MPIWiz: Subgroup Reproducible Replay of MPI Applications

Ruini Xue[†], Xuezheng Liu[‡], Ming Wu[‡], Zhengyu Guo[‡], Wenguang Chen[†]
Weimin Zheng[†], Zheng Zhang[‡], Geoffrey M. Voelker[¶]
[†]Tsinghua University [‡]Microsoft Research Asia [¶]University of California, San Diego

ABSTRACT

Message Passing Interface (MPI) is a widely used standard for managing coarse-grained concurrency on distributed computers. Debugging parallel MPI applications, however, has always been a particularly challenging task due to their high degree of concurrent execution and non-deterministic behavior. Deterministic replay is a potentially powerful technique for addressing these challenges, with existing MPI replay tools adopting either data-replay or order-replay approaches. Unfortunately, each approach has its tradeoffs. Data-replay generates substantial log sizes by recording every communication message. Order-replay generates small logs, but requires all processes to be replayed together. We believe that these drawbacks are the primary reasons that inhibit the wide adoption of deterministic replay as the critical enabler of cyclic debugging of MPI applications.

This paper describes *subgroup reproducible replay* (SRR), a hybrid deterministic replay method that provides the benefits of both data-replay and order-replay while balancing their trade-offs. SRR divides all processes into disjoint groups. It records the contents of messages crossing group boundaries as in data-replay, but records just message orderings for communication within a group as in order-replay. In this way, SRR can exploit the communication locality of traffic patterns in MPI applications. During replay, developers can then replay each group individually. SRR reduces recording overhead by not recording intra-group communication, and at the same time reduces replay overhead by limiting the size of each replay group. Exposing these tradeoffs gives the user the necessary control for making deterministic replay practical for MPI applications.

We have implemented a prototype, MPIWiz, to demonstrate and evaluate SRR. MPIWiz employs a replay framework that allows transparent binary instrumentation of both library and system calls. As a result, MPIWiz replays MPI applications with no source code modification and relinking, and handles non-determinism in both MPI and OS system calls. Our preliminary results show that MPIWiz can reduce recording overhead by over a factor of four relative to data-replay, yet without requiring the entire application to be replayed as in order-replay. Recording increases execution time by 27% while the application can be replayed in just 53% of its base execution time.

1. INTRODUCTION

Software bugs remain a key factor impacting the reliability of high-performance computing (HPC) applications. A recent study of more than 20 HPC systems, for instance, found that software bugs accounted for 24% of system failures [32]. Debugging HPC applications has always been a particularly challenging task [13] due to their high degree of concurrent execution, distributed communication across multiple nodes in contemporary cluster environments, and non-deterministic behavior [5]. These characteristics conspire to make subtle bugs difficult to reproduce and debug.

Deterministic replay is a potentially powerful technique for debugging HPC applications. When an application executes, the replay tool records application inputs, such as messages, during the recording phase. Then when developers want to track debug the application, in the replay phase they can replay the faulty processes to any state of a recorded execution and investigate how these processes reached that state. Replay tools for HPC applications typically fall into two categories [22]. *Data-replay* tools record all incoming messages to each process during program execution, and provide the recorded messages to processes during replay and debugging. With this approach, developers can replay just faulty processes rather than having to replay the entire parallel application. In contrast, *order-replay* tools only record the outcome of non-deterministic events in inter-process communication during program execution — for instance, `MPI_Recv` with `MPI_ANY_SOURCE` for MPI applications — and lets sending processes reproduce the actual message contents during replay. Since order-replay only records the ordering of non-deterministic events, it records far less data than data-replay.

Despite their benefits, however, existing replay approaches for HPC applications impose substantial overhead either at recording or replay time. These overheads, unfortunately, limit their current utility. With data-replay, the system must record the contents of all inter-process communication to make every process replayable. As a result, the replay log size scales directly with the amount of inter-process communication, and becomes prohibitively large for even moderate-scale applications. The NPB kernel LU with 64 processes in our experiments, for example, logs data at the rate of nearly 14 GB per minute. While order-replay dramatically reduces recording overhead, it imposes overhead during the replay phase. All processes must be replayed together, even if the developer only needs to investigate just a few processes. This requirement is impractical when an application has a large number of processes but a developer only has limited resources for debugging, a common situation in HPC settings. In general, these two approaches represent different trade-offs between introducing overhead in the recording vs. replay phases, and it remains a challenge to find a balance between them and make deterministic replay applicable for large HPC applications.

In this paper we propose a hybrid approach called *subgroup reproducible replay* (SRR) that provides the benefits of both data-replay and order-replay while balancing their trade-offs. SRR divides all processes into disjoint *replay groups*. During the recording phase, SRR records the contents of messages crossing group

boundaries as in data-replay, but records just message orderings for communication within a group as in order-replay. During replay, developers can replay each group independently of the others. SRR reproduces messages from outside the group directly from the logs, and reproduces messages from within the group through direct execution. It uses the recorded outcome of non-deterministic events to make the replay deterministic with the original execution. SRR therefore reduces recording overhead by not recording intra-group communication, and at the same time constrains the replay overhead by limiting the size of each replay group.

SRR is able to dramatically reduce recording overhead because it exploits communication locality within HPC applications [8, 39, 18, 15]. Developers often structure communication patterns such that processes typically exchange messages within a group to avoid global synchronization and therefore improve overall application performance. By design, such intra-group messages are the dominant form of communication in an application. By making replay groups consistent with these communication patterns, most messages therefore become internal to a replay group and SRR avoids having to record them.

As a result, the size of the replay group is the critical parameter that fundamentally determines the overhead of the SRR approach. We therefore developed a graph partitioning algorithm to discover the communication locality of a running application, and automatically determine the appropriate group size that best captures this locality. With SRR, though, developers are still free to choose a group size according to their needs. In fact, for an application with n processes, group sizes of 1 and n make SRR behave exactly like traditional data-replay and order-replay approaches, respectively.

We have implemented a prototype of SRR for MPI applications called MPIWiz. MPIWiz uses a flexible library-based replay framework called R2 [11] that employs binary instrumentation to transparently make any MPI application replayable without recompilation. We apply MPIWiz to several common MPI applications to demonstrate its benefits compared to data-replay and order-replay approaches alone. The extent of these benefits of SRR depends upon the communication patterns of applications. For example, for an application (NPB kernel CG) with good communication locality, MPIWiz only generates 22% of the data-replay log size. Even for applications that have no communication locality (e.g., NPB kernel FT, which uses all-to-all communication), MPIWiz is still able to reduce log size by about 13%. Across a suite of applications, the average recording and replay overheads of MPIWiz naturally fall in between that of data-replay and order-replay.

Furthermore, by building on the R2 framework, MPIWiz provides two additional practical features not found in existing MPI replay tools. First, in addition to non-determinism in communication, MPIWiz also captures non-determinism in operating system calls (e.g., `gettimeofday`, `random`) invoked by MPI applications. All of the applications in the NPB benchmarks, for example, use non-deterministic system calls (`MPI_Wtime`), and capturing the full extent of non-determinism is necessary for their accurate replay. Second, MPIWiz guarantees that the memory footprints of the replayed processes are *identical* to those of the processes in recording execution — all application memory locations at user-level have the same values during both the record and replay phases. Ensuring identical memory values further aids developers in debugging applications by removing inconsistencies between deployment and debugging environments as a source of uncertainty.

The rest of the paper is organized as follows: Section 2 presents related work. Section 3 discusses the design of SRR. Section 4 describes our approach for determining replay groups, and Section 5 describes the MPIWiz replay framework. Section 6 details how MPIWiz deals with the various sources of non-determinism in MPI applications. We evaluate SRR relative to data-replay and order-replay in Section 7. Finally, Section 8 summarizes our work and concludes.

2. RELATED WORK

Deterministic replay is just one of many approaches that have been proposed for debugging MPI applications. This section discusses how subgroup reproducible replay relates to existing replay approaches, and places it in the larger context of MPI debugging approaches.

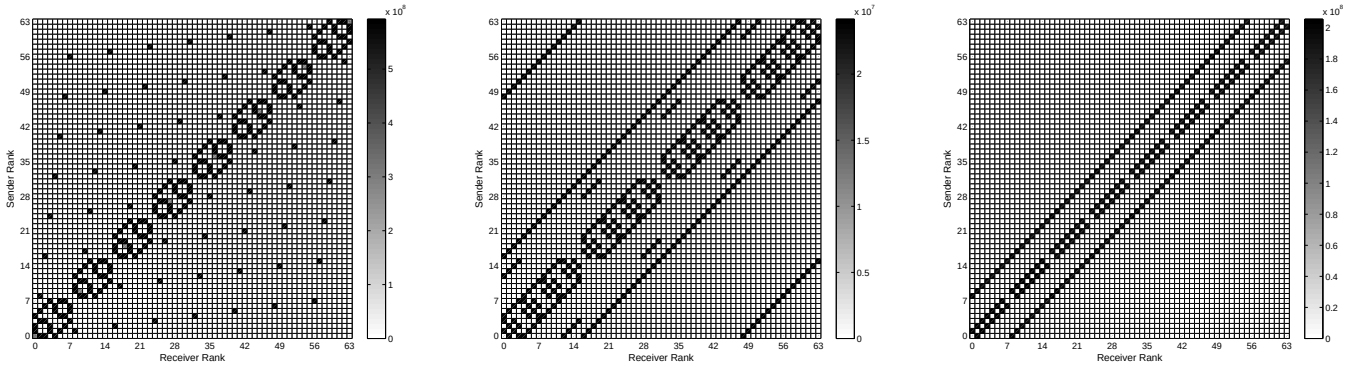
Replay-based debug tools adopt either data-replay [25, 4, 2] or order-replay [5, 21, 20] approaches to debug MPI applications. Each approach has tradeoffs. Data-replay tools generate massive logs, while order-replay tools require all processes to be replayed together. Both of them are impractical for large-scale applications. SRR is a balance between data-replay and order-replay. It only requires replaying a group of processes, and users can adjust the number of processes in the group to match the resources of their debugging environment. Further, by exploiting locality in the communication patterns of an application, SRR can substantially reduce the size of logs generated during the recording phase. As a result, SRR makes it possible to debug large-scale applications with limited resources in the development environment.

Most MPI replay systems are implemented via the MPI profiling interface. While convenient, unfortunately this approach does not handle non-deterministic system calls, thereby making it difficult to guarantee a completely faithful replay. We have implemented SRR in MPIWiz on a general record and replay platform [11], enabling MPIWiz to capture all forms of non-determinism in MPI applications. Other MPI replay systems are implemented by changing the source code of the MPI distribution, which limits its portability. MPIWiz employs binary instrumentation to transparently replay applications without the need to recompile or relink, and does not depend on the MPI distribution.

More generally, deterministic replay is just one of many approaches that have been proposed for debugging MPI applications. MPI-CHECK [24] uses static analysis to check the source code at compile time against the programming rules specified by the MPI standard. Although useful for identifying some classes of errors, static analysis also suffers from false negatives since many parameters are not known until the application executes. Parallel debuggers operate similarly as sequential debuggers [36, 31, 30, 28, 4], but can be difficult to use effectively when there are hundreds of processes. Automatic checking tools address the drawbacks of manual checking in parallel debuggers [38, 12, 19, 7, 37]. These tools use similar rules as static analysis, but they verify the rules at runtime rather than compile time. IMC records communication during execution and checks the trace to identify predefined errors [6]. Several recent efforts have also explored the use of model checking to verify MPI applications [23, 29, 33, 35, 34] to verify MPI codes. Though it is difficult for these tools to handle bugs due to non-determinism, they are helpful in application understanding and deterministic bug tracking. We view SRR as complementary to these efforts, and it can be used in conjunction with all of them.

3. DESIGN OVERVIEW

This section presents an overview of our design of subgroup reproducible replay. We first explain how SRR divides all processes into the replay groups and exploits communication locality to reduce recording overhead. Then we describe SRR record and replay for an MPI application.



(a) CG (b) MG (c) LU

Figure 1: Communication traffic in CG, MG and LU (CLASS=C, NPROCS=64). The cell at (i, j) represents the communication volume (in bytes) between process i and j using shades of gray. With replay groups in sets of 8 sequentially numbered processes, intra-group messages account for about 77%, 55% and 50% of application communication traffic, respectively.

3.1 Communication Locality

The key inspiration underlying SRR is that HPC applications typically exhibit strong communication locality. A number of studies have shown that HPC applications, by design, have structured communication patterns where processes predominantly exchange messages within a group [8, 39, 18, 15]. Such communication patterns increase parallel application performance by improving their scalability. For example, Figure 1 shows the communication patterns of three widely-used MPI benchmarks — CG, MG and LU from the NAS Parallel Benchmarks (NPB) [1]) — generated from a trace-driven simulation using SIM-MPI [27]. In the figure, the gray level of a cell at the i^{th} row and j^{th} column represents the communication volume between two processes i and j. The figure shows distinct group patterns, where processes can be divided into small groups (e.g., a group of size eight for CG) in which intra-group communication comprises the majority of overall communication traffic. By organizing application processes into appropriate replay groups, SRR can dramatically reduce recording overhead and limit the resources required during replay.

Based on these observations, a desirable assignment of processes into replay groups should satisfy two conditions. First, each group should have a moderate size so that replay requires only moderate hardware resources, i.e., MPIWiz can replay a group of processes reasonably fast with fewer computing resources than required for the entire application. MPIWiz allows developers to specify an upper-bound of the group size, and ensures this bound when partitioning processes. By this means developers have the freedom to choose different trade-offs between recording and replay overhead. Second, given the constraint on group size, the processes assigned to each group should reduce inter-group communication as much as possible. In Figure 1a, for example, it is much more effective to assign processes to replay groups in sets of eight sequentially ranked processes rather than eight randomly selected processes. We will describe how MPIWiz satisfies the two conditions in Section 4, and Section 7.3 experimentally quantifies the tradeoff of recording overhead and replay group size, and the benefits of making informed group membership assignments.

3.2 SRR Record

During the recording phase, SRR records only the contents of incoming inter-group messages, and records the order of messages only if an operation is non-deterministic (for any message, no matter whether the message is intra-group or not). Table 1 outlines the recording and replay mechanism for different operations.

In the spirit of order-replay, since the replay phase executes the processes within a replay group, SRR does not have to record any data corresponding to intra-group deterministic MPI communication operations. Replay naturally reproduces message order and contents. For non-deterministic intra-group communication, SRR records the order of the messages but does not record their contents; execution during replay will faithfully reproduce the contents of messages as long as it preserves the original ordering. Messages sent out of the replay group do not affect the replay of the group, and can be safely ignored.

In the spirit of data-replay, SRR records the full contents of inter-group messages received from outside the replay group, as well as their order if the receive operation is non-deterministic. Since only processes within a group execute during the replay phase, messages from outside the group have to be recorded during the recording phase so that they can be faithfully emulated during replay.

Collective communication involves messages sent among a set of processes. As a result, during the recording phase MPIWiz needs to determine the process membership of a collective communication to determine what information to record, if any. A collective communication specifies the set of processes involved in the operation, albeit indirectly. To determine whether the current process is involved, MPIWiz uses two steps. First, it determines the MPI group associated with the collective communication's communicator via `MPI_Comm_group`. It then translates its global rank (the rank in `MPI_COMM_WORLD`) into the context of this MPI group via `MPI_Group_translate_ranks`. If the result is `MPI_UNDEFINED`, then the process is not in the group, otherwise it is participating in the collective communication. Since the process membership of the collective communication cannot be recalculated during the replay phase, MPIWiz records this information during the record phase.

Finally, when non-deterministic system calls are used directly by the application, MPIWiz always records their results in the log.

3.3 SRR Replay

In the replay phase, SRR replays all of the processes of only one replay group. The replayed processes generate intra-group messages directly, and the contents of incoming inter-group messages are emulated using the recorded logs. Since the message orders of non-deterministic operations have been recorded, SRR can

Table 1: Summary of record and replay mechanisms for the MPI API and system calls.

Category	API Example	Record&Replay mechanism
Point-to-Point Communication	`MPI_Send`, `MPI_Recv`	During recording, log inter-group communication, ignore intra-group communication. During replay, emulate inter-group communication using the log, and reproduce intra-group communication. For non-blocking operations, log the request type (send or receive) and buffer information. (Section 3.2 & Section 3.3)
Collective Communication	`MPI_Bcast`, `MPI_Gather`	Record members involved, handle message contents as with point-to-point communication. Replace with point-to-point communication during replay. (Section 3.2 & Section 3.3)
MPI Environment API	`MPI_Init`, `MPI_Comm_rank`	Record parameters and return value. Emulate them using the log during replay. (Section 3.2 & Section 3.3)
Non-determinism in MPI	`MPI_ANY_SOURCE`, `MPI_ANY_TAG`	Wildcard receives. Record the real values for source and tag fields. Replace them with real values during replay. (Section 6.1.2)
	`MPI_Waitany`, `MPI_Testsome`	Record returned request indices, and handle corresponding messages buffer according to point-to-point communication. During replay, check the request type and handle corresponding message buffer according to point-to-point communication. (Section 6.1.3)
	`MPI_Probe`	Record the parameters and returned value. Emulate them using the log during replay. (Section 6.1.3)
Non-determinism in OS	`GetTickCount`	Record the outcome according to the semantics of the routine. Emulate them using the log during replay. (Section 6.2)

```
/* MPI_Bcast() replay code */
load MPI_Bcast rank_list from log
if (I am root) { /* for data sender */
  foreach rank in rank_list:
    if (rank is in replay group)
      send message to rank
} else { /* for data receiver */
  if (root is in group)
    recv message from root
  else
    load message from log
}
```

Figure 2: Pseudo code for `MPI_Bcast` during replay.

guarantee a deterministic replay by enforcing this order in replay, as follows. For non-deterministic point-to-point operations, SRR replaces the parameters introducing non-determinism (e.g., wildcards) with their real values. For collective operations, SRR replays them using multiple point-to-point operations because some of the participants might be outside of the group. For instance, Figure 2 illustrates how it replays `MPI_Bcast`. If the replayed process is the broadcast root, it generates messages to only those processes in the replay group (the others do not execute during replay). If the replayed process is a recipient, it receives the message as with point-to-point communication replay if the root is in the group, otherwise, the recipient loads the message from the log.

A key difference between SRR and order-relay is how SRR delivers intra-group messages. In order-replay, the message is delivered to the receiver by the sender through the same channel as in the recording phase, e.g., through a socket. However, in SRR replay, the original MPI initialization routine which constructs the MPI parallel computing environment cannot execute identically as in the recording phase because only the processes in the replay group execute. To address this problem, SRR replay skips the construction of the full computing environment (similar to data-replay), and as a result does not establish the communication channels among the replayed processes as during the recording phase. As a result, normal MPI communication functions (e.g., `MPI_Send`) cannot deliver the message in SRR replay. Therefore, SRR needs to emulate the communication channels and deliver messages itself. The emulated communication channels also enable SRR to control the order of message delivery, a necessary condition for reproducing non-determinism in MPI message orders. In our current implementation of MPIWiz, we use a dedicated replayer process as a message relay, i.e., the wrapper of `MPI_Send` sends messages to the replayer, and `MPI_Recv` receives messages from the replayer.

4. REPLAY GROUPS

The size and membership of replay groups are key parameters that determine the overhead and performance of SRR. A straightforward way to determine these parameters is to utilize expert knowledge and manually specify them. This approach is reasonable when the developer knows much about the communication flow of the MPI application and the communication pattern presents good locality.

In general, though, it is more practical to have MPIWiz automatically determine replay groups. First, we need to find out the group size constraint. Then, for a given group size, we determine an efficient membership of all processes to replay groups. Finally, we search in a range of group sizes below a bound provided by the user to find one that provides a near-optimal result, i.e., results in the smallest inter-group communication volume.

For the group size constraint, in practice we imagine users setting it to a small multiple ($1--4$) of the number of processor cores in their debugging environment both to limit the replay execution time overhead, as well as to fit the working set of the replayed processes within memory constraints. As with data-replay, a goal of SRR is to enable users to debug MPI applications on a single machine. To keep the execution time of replay reasonable, the group size should reflect the resources available in the debugging environment. Having the group size reflect the number of processors minimizes replay execution overhead. With the advent of multi-core architectures, we believe this rule of thumb matches hardware trends well.

After setting the group size constraint, we formalize the group membership problem as a k-way graph partitioning problem. We represent the communication pattern of an application with a graph, in which each vertex represents a process and the weight of each edge is the aggregate message traffic between the two corresponding processes of the two vertices. This communication graph can be obtained, for example, by profiling the execution of the application (e.g., with a tool like SIM-MPI [27] as in Figure 1). Our goal is to partition this graph into k partitions with roughly equal numbers of vertices, and where the sum of the weights of edges crossing the partition boundary is minimized.

Although MPIWiz can handle replay groups of different sizes, we argue that partitioning the process graph into nearly equal sizes is desirable for the following reasons. First, the replay group size

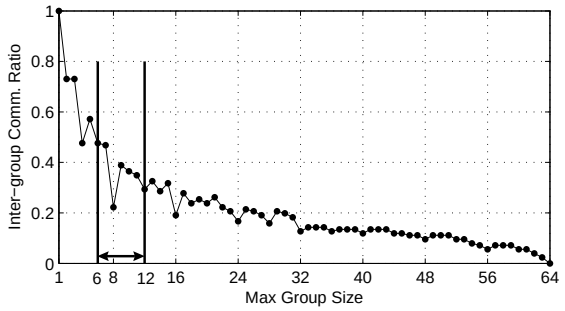


Figure 3: The ratio of inter-group message size relative to total communication volume vs. group size for CG (NPROCS=64, CLASS=C). If the group size upper bound is set to $S = 12$, MPIWiz would choose the best group size in the range $[6, 12]$.

constraint should be applied to the largest partition, otherwise MPIWiz cannot replay that partition with reasonable overhead. On the other hand, if the total size of two partitions is still smaller than the specified upper bound, they should be merged together. Then the inter-group communication between them becomes intra-group communication and does not need to be recorded. Therefore, the merging of the two partitions will certainly not increase the recording overhead, but more than likely will decrease it. Creating nearly equal-sized groups tends to generate a balanced result which is more efficient than the original unbalanced one.

Although this kind of graph partitioning problem is NP-complete, many algorithms have been proposed to find reasonably good partitioning using heuristic methods [9, 3, 26]. MPIWiz employs a multilevel k-way partitioning algorithm MLkP [16] to partition the process communication graph. We chose this algorithm since it can generate a high-quality partitioning in linear time complexity proportional to the number of edges (Section 7.3 shows the effectiveness of MLkP). Let S be the upper bound of the partition size, n be the number of vertices in the communication graph, and k the number of partitions. Because the result of MLkP is nearly balanced, MPIWiz can limit the size of a partition to be lower than S by guaranteeing that $n/k < S$.

Given an upper bound S of the partition size, the largest value of k that satisfies this upper bound may still not produce the minimal amount of inter-group communication traffic. The reason is that a group size of eight exactly matches the natural group communication boundaries in the application (Figure 1). Slightly larger groups will include processes that place them outside of their natural communication group, causing substantially more inter-group communication that MPIWiz needs to record in the log. For example, Figure 3 shows the inter-group communication traffic for the NPB benchmark CG, as collected by SIM-MPI [27], for a range of replay group sizes. In general, the inter-group communication volume decreases with larger group sizes, but there still exist some local optimal points (e.g., at 4, 8, 16, etc.).

The next step is to automatically discover a local optimum near the upper bound S on replay group size. We do so by applying MLkP iteratively across a range of values of k to discover the value that generates the optimal result. For each group size, MLkP identifies the replay group and we use SIM-MPI to collect the aggregate inter-group communication volume. Fortunately, the number of iterations is reasonably small — we show that MPIWiz only needs to search for group sizes in the range $S/2 < n/k < S$. Figure 3 shows the results of this process when $S = 12$.

Let s be the replay group size of one local optimal point. The group size $2s$ should also be a local optimal point since, in this case, groups with size $2s$ can be formed by merging pairs of groups with size s. These merging operations do not increase the inter-group communication volume while maintaining its local optimum at the same time. Then given S as the group size upper bound, if there is no local optimal point with group size less than S, MPIWiz can find the optimal partitioning result when group size equals to S. If there are local optimal points with group size less than S, let s be the local optimal group size which is less than and nearest to S. Then we must have $2s > S$ since $2s$ is also a local optimal point, which we can rewrite as $s > S/2$. Hence, since s by definition is less than S, we have $S/2 < s < S$. Therefore, by searching in the group size range from $S/2$ to S, MPIWiz can find a global optimal group size satisfying the constraint.

Table 2: Group sizes grow slower as applications scale.

Proc. #		16	32	64	128	256
Group Size	CG	4	8	8	16	16
	MG	4	8	8	16	16

A final observation is that, in scalable MPI applications, the size of a communication group does not scale as quickly as the overall application size. As the application scales up, the number of groups increases accordingly, while the number of processes within each group increases more slowly. Table 2 shows that the group size grows more slowly as applications scale to larger numbers of processes for two NPB applications. As a result, even with applications using a large number of processes, MPIWiz can replay the application using relatively small replay groups. In addition, since group size grows slower than application size, the larger the application, the more SRR will reduce recording overhead relative to data-replay.

5. REPLAY FRAMEWORK

Deterministic replay requires that all MPI routines are both replayable and deterministic. In this section we describe our approach for making MPI routines replayable, and in Section 6 we describe the techniques we use to ensure that all MPI and system routines are deterministic.

MPIWiz takes advantage of a replay platform called R2 that we previously developed for multi-threaded, distributed applications [11]. R2 uses binary instrumentation to transparently interpose wrappers on API routines for both runtime environments as well as system calls. For the MPI library, MPIWiz transparently interposes a wrapper routine around each MPI routine in the library interface.

Under MPIWiz, when applications call into the MPI library they instead invoke the wrapper. The wrapper implements the record and replay functionality, and invokes the actual MPI library routine when necessary. For example, when an application calls `MPI_Recv`, it will instead call a wrapper for the function. During the record phase, the wrapper will call `MPI_Recv`, record the contents of the received message to the log if appropriate, and then return to the application. During the replay phase, the wrapper may emulate `MPI_Recv` by returning the contents of the message from the log rather than invoking the routine.

Implementing the recording and replay functionality for the entire MPI API can be tedious because it requires wrapping nearly 300 API functions. Thanks to R2, which provides a flexible code generation mechanism, we only need to write several general code templates as annotations on API parameters instead of manually programming recording and replay wrapper functions for every API routine. MPIWiz currently supports 191 of the most commonly-

```
int
[reproducible]
MPI_Recv (
  [out, bsize("GetSize(datatype, count)"), force] void* buf,
  [in] int count,
  [in] MPI_Datatype datatype,
  [in] int src,
  [in] int tag,
  [in] MPI_Comm comm,
  [out, opt(MPI_STATUS_IGNORE)] MPI_Status* status
  );
```

Figure 4: The annotation of `MPI_Recv`. `reproducible` means this function may be reproduced if it is called by a process in a replay group. `in` means the parameter is not modified, and no logging is needed, while `out` indicates the parameter is changed by the routine and it is recorded automatically by generated code. `bsize` indicates how to obtain the length of the buffer, and `force` means the length itself should be saved since the length can not be calculated during replay. `opt` means the parameter can be null or some special values, in which cases it does not need to be saved.

used MPI functions (MPI-2.0 has 284 functions in total). Functions not supported include remote memory access, MPI I/O, and dynamic process creation. Expanding the set of supported functions with further annotations is ongoing work.

For example, Figure 4 shows the signature of `MPI_Recv`. To generate its wrapper functions, we only need to annotate its input and output parameters as shown in the figure. The generator in R2 will then parse the annotations and generate code that logs input parameters during the recording phase and returns output parameters during the replay phase. Compared with the manual approach, automatic code generation is more convenient and avoids many potential errors in manual programming. Compared with previous MPI replay tools, which use customized MPI libraries rather than binary instrumentation, this approach has the benefit that it is transparently applicable to different MPI distributions.

6. HANDLING NON-DETERMINISM

Roughly speaking, for a replay tool everything that cannot be deterministically reproduced during the replay phase needs to be logged during the recording phase. This section describes our approach for handling non-determinism in both the MPI API and in system calls.

6.1 Non-determinisms in MPI API

MPIWiz needs to accomodate three sources of non-determinism in the MPI API: inter-group messages, the use of wildcard parameters when receiving messages, and the use of wait, test, and probe operations.

6.1.1 Inter-group Message Content

During the recording phase, when a process receives a message MPIWiz needs to determine whether or not the message came from a sender outside the replay group. At initialization time, MPIWiz reads the membership of replay groups from a configuration file which stores the ranks of processes in each group. When a process receives a message MPIWiz checks the membership of the sender process, and records the received message if it is from a different replay group than the current process.

MPIWiz retrieves the message from the receiving buffer, typically provided as parameters to MPI routines. In addition to plain buffers, MPI allows applications to specify derived data types for which the sender can transmit a data trunk which is later split and placed into non-contiguous positions of the receiving buffer at the receiver. Currently, MPIWiz records the entire buffer used with derived data types. In such cases, recording the entire receive buffer ensures the correctness of replay, but it may be inefficient because only a subset of the buffer may actually be used. A more efficient solution is for MPIWiz to process the definitions of these data types and record only the transmitted data. Our experience with MPI applications suggests that the use of derived data types is uncommon, however, so we have left optimizing derived data types for future work.

6.1.2 Wildcard Receives

Another source of non-determinism is the order of messages received using wildcard parameters. For a receive operation, an application typically specifies the source, the communicator, and a conventional tag. However, both the source and the tag can be specified using wildcards, e.g., `MPI_ANY_SOURCE` and tag `MPI_ANY_TAG`. A wildcard as the source (or tag) identifier allows a message from any process to be accepted. When wildcards are used to receive messages, the arrival order of messages is non-deterministic. Different orders may change the application's behavior because the execution after the receive operation may depend on the order of messages.

Since MPIWiz handles messages from inside and outside of the replay group differently, it needs to determine the actual message origin when the source is set to `MPI_ANY_SOURCE`. Typically, MPIWiz can retrieve the actual source from the status parameter. Unfortunately, the MPI standard allows applications to ignore the status parameter by setting it to a special value (`MPI_STATUS_IGNORE` or `MPI_STATUSES_IGNORE`). To successfully record such receive operations, we transparently replace the special status value provided by the application with an allocated private variable provided by the MPIWiz runtime to ensure the MPI routine returns the necessary information. After retrieving the actual source of the message, MPIWiz records it and restores the special status value as provided by the application. MPIWiz performs similar steps during replay. This method makes use of MPI's functionality, and does not change the application's control flow and semantics.

Non-blocking receive operations can also use these wildcard tags, and therefore also need special treatment to determine the source process of a message. Non-blocking operations return immediately without waiting until messages are received or delivered (e.g., `MPI_Irecv/MPI_Isend`). Instead, an application uses test operations (e.g., `MPI_Wait/MPI_Test`) to check for the arrival of messages or to check if a send operation has finished. Non-blocking receive operations return MPI request objects, which can subsequently be used as handles by test operations. A test operation returns a status parameter for the related request. Again, the status can be ignored according to MPI standard. We adopt the same techniques described above to determine the actual sources of the non-blocking receive operations, and record and replay them accordingly.

6.1.3 Waits, Tests and Probes

The MPI routines `MPI_Waitsome`, `MPI_Waitany`, `MPI_Testsome` and `MPI_Testany` can also introduce non-determinism. These procedures operate on a set of requests posted by previous non-blocking operations, and return if any or some of the requests have completed in blocking or non-blocking manners for waits and tests, respectively.

To address the non-determinism introduced by wait and test operations, MPIWiz maintains a table tracking the requests posted by non-blocking operations and their corresponding buffer information. To help illustrate how MPIWiz handles such operations,

```
MPI_Request requests[2];
MPI_Irecv(buf1, cnt1, type1, src1, tag1, com1, requests[0]);
MPI_Irecv(buf2, cnt2, type2, src2, tag2, com2, requests[1]);
...
/* wait until either buf1 or buf2 is ready */
MPI_Waitany(2, requests, index, status);
```

Figure 5: An example of non-determinism caused by `MPI_Waitany`. When `MPI_Waitany` returns, either `buf1` or `buf2` is ready, depending on the actual execution.

Figure 5 illustrates the use of `MPI_Waitany`. During the record phase, `MPI_Irecv` just inserts the request and buffer information into the table. After `MPI_Waitany` returns, it records the index of the returned request and the status structure. Since the receive buffer contains the message at this point, MPIWiz uses this information to index into the table and decide whether it needs to record the message contents based on the message source.

During the replay phase `MPI_Irecv` also only inserts the request and buffer information into the table, and the buffer contents are backfilled by `MPI_Waitany`. If the request is bound to a send operation, the process is similar. Indeed, all other wait and test functions (`MPI_Wait`, `MPI_Waitall`, `MPI_Test` and `MPI_Testall`) are handled in the same way.

Blocking `MPI_Probe` and non-blocking `MPI_Iprobe` are analogous to `MPI_Wait` and `MPI_Test`, respectively. Both of them can also accept `MPI_ANY_SOURCE` and `MPI_ANY_TAG` as source and tag parameters. The difference between probes and the wait and test operations is that, after a successful probe, the corresponding message is not copied to the application buffer. Therefore, unlike waits and tests, it is impossible to handle the message at the time of probe operation. A common programming convention is to invoke a receive operation after a probe. Therefore, MPIWiz records and replays probe operations as normal operations without special treatment. Instead, it is the responsibility of the subsequent receive operation to properly handle the message. During replay, MPIWiz loads the return value of the probe operation from the log. Doing so directs the application to follow the same execution path as during the replay phase, and the corresponding receive operation takes over.

6.2 System Calls

Some applications may directly call some system calls provided by the operating system, or indirectly through the MPI runtime. These system calls can depend on the execution environment, and therefore are non-deterministic when the replay environment differs from the recording environment. For example, random number generators will produce inconsistent numbers, and `gettimeofday` (on which `MPI_Wtime` depends) returns different values at different times. These system calls fall into a wide range of categories, including I/O operations (e.g., reading a file), callback functions of signals and interrupts, etc., and must be carefully dealt with so that they can be replayed deterministically. Further, ideally the memory footprints of each process should also be identical for both the record and replay phases. Having identical memory footprints is very useful when debugging memory-related bugs like buffer overflow. To make both system calls and memory footprints deterministic across the record and replay phases, MPIWiz relies upon the functionality implemented in the R2 framework.

7. EVALUATION

In this section, we evaluate MPIWiz using NPB benchmarks and real-world applications with a variety of communication patterns to demonstrate the benefits SRR provides over data-replay and order-replay approaches alone. All of the applications also use some form of non-deterministic operations, and we demonstrate that MPIWiz is able to correctly handle such cases.

Table 3: Application characteristics: *All-to-All*: all-to-all communication pattern; *Locality*: group communication locality; *M/S*: master/slave pattern; *Non-determ. MPI* and *Non-determ. Sys*: whether non-deterministic MPI and operating system calls are used, respectively; *Coll. Operation*: whether collective operations are used.

	Communication Patterns						
	Locality			All-to-All			M/S
Operations	CG	MG	LU	FT	GE	ASP	PU
Non-determ. MPI		√	√		√	√	√
Non-determ. Sys	√	√	√	√	√		
Coll. Operation	√	√	√	√	√	√	

This sections answers the following questions:

- What is the record and replay overhead and performance of MPIWiz compared to data-replay and order-replay alone?
- What is the sensitivity of the record log size to replay group size and membership?

We start by describing our methodology.

7.1 Methodology and Applications

We conducted our experiments on a cluster of eight nodes totalling 64 cores. Each node is equipped with two Quad-Core Intel Xeon 2.33 GHz CPUs, 8 GB RAM, and a 140 GB hard disk. We run MPI applications within the MPICH2-1.0.7 environment on Windows Server 2003 Enterprise Edition SP1. All machines are connected through a switched 1Gbps Ethernet LAN. Each process writes its log to local disk without compression.

We evaluate MPIWiz using the following set of applications with 64 processes:

- CG, MG, LU, FT: NAS Parallel Benchmarks (NPB) kernels version 2.4 compiled in Class C [1].
- GE [14]: A message passing implementation of Gaussian Elimination.
- ASP [17]: A parallel application that solves the all-pairs-shortest-path problem with the Floyd-Warshall algorithm.
- probe-unexp (PU) [10]: A test program that validates the correctness of non-deterministic MPI probe operations, and stress-tests communication primitives using many messages and a range of messages sizes.

Table 3 summarizes the characteristics of the applications we use in terms of their communication patterns and their use of non-deterministic operations.

The applications fall into three distinctive communication patterns. (1) CG, MG and LU have communication locality as illustrated in Figure 1, and every eight successively ranked processes form a natural replay group. (2) For FT, GE and ASP, communication is uniformly distributed across all processes. This all-to-all style has no communication locality, and represents a less-than-ideal case for SRRin terms of reducing recording overhead. Though these applications do not benefit as much as CG and MG from SRR, our experiments show that SRR remains helpful even when no communication locality exists. As with the previous applications, we place eight processes with successive ranks in a replay group (Section 7.3 shows that results are insensitive to the particular replay group membership). (3) PU uses a master/slave pattern:

the master sends messages to slaves, and slaves only communicate with the master. Since replay groups are disjoint, the master can only be in one group. Consequently, we organize the master (rank 0) and a slave (rank 1) as one replay group, and each other slave as its own independent replay group. Since nearly every process is in its own replay group (as the case for traditional data-replay), this pattern represents a worst-case for SRR.

The applications also use different forms of non-determinism, as described in Section 6.1. In terms of non-deterministic MPI communication operations, MG, LU, GE and PU use receive operations with the `MPI_ANY_SOURCE` source wildcard; PU also uses probe operations with the wildcard `MPI_ANY_TAG` message type. In terms of non-deterministic system calls, GE uses the Windows system call `GetTickCount` directly, and all NPB kernels call `MPI_Wtime`. The prevalence of non-deterministic operations in these common applications show that handling non-determinism in both communication and system is important for any replay tool.

Finally, the table also shows that all applications except PU use collective communication operations, which MPIWiz handles using the techniques described in Section 3.2 and Section 3.3.

7.2 Record and Replay Overhead

This section presents the record and replay overhead of MPIWiz compared to data-replay and order-replay in terms of execution time (for both record and replay phases) and record log size. MPIWiz implements data-replay by considering each process in its own replay group, and order-replay by including all processes in one replay group.

7.2.1 Execution Time

Figure 6 compares the execution time of MPIWiz for the various record and replay scenarios relative to the baseline execution time of the original application. It shows the execution time of both the record and replay phases for data-replay and SRR with replay groups of eight processes, and the execution time of the record phase for order-replay.[1]

The execution times of the record phases of all approaches are slower than the baseline due to the overhead of capturing logs. Since SRR is a balance between data-replay and order-replay, its execution performance falls roughly halfway between those two approaches.

The execution times of the replay phases of data-replay and SRR are faster than the baseline because messages in both cases are taken from the log. As expected, the replay time of SRR is slower than data-replay. Data-replay emulates all communication by reading from the log without the need to wait and synchronize, while SRR needs to reproduce and exchange intra-group messages. Also, since the current implementation of MPIWiz uses a relay process for all communication — in particular, to simplify the handling of collective communication — this level of indirection adds additional overhead during replay.

Recall from Section 4 that we expect users to constrain replay group sizes to fit within the computing resources of their debugging environments, which is typically limited in HPC environments. The replay execution times in Figure 6 show the benefits of having enough processor cores to replay all processes in the replay group

[1]MPIWiz currently replays all processes on a single node for ease of implementation and debugging. Since it is misleading to measure the execution time of order-replay for all 64 processes on a single node, we do not report replay execution time for order-replay. Since order-replay restarts all processes and executes all computation and communication as with normal execution, its replay execution time is nearly the same as the base execution time [25].

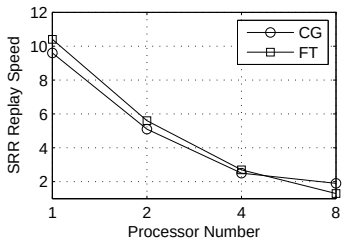


Figure 7: SRR replay speed relative to data-replay for different number of processors.

(eight cores in that experiment). What is the impact on replay execution time if a user has fewer processor resources than the size of a replay group? Figure 7 answers this question for the CG and FT applications with a replay group size of eight. When there are sufficient processors (eight), replay execution time is faster than data-replay. With fewer processors, the replay time correspondingly increases. Note that, with fewer processors, not only does replay have to execute the replayed processes, but it also has to handle all intra-group communication operations as well. Increased replay execution time may be acceptable for debugging; if not, users can always decrease the size of the replay group at the cost of increasing the size of the log. We view this flexibility as an important feature of SRR.

7.2.2 Log Size

Figure 8 compares the size of the logs generated during the record phase of MPIWiz for data-replay, order-replay, and SRR. Since order-replay only needs to record ordering information about non-deterministic operations, as expected its log sizes are negligible compared to data-replay and SRR. However, since the goal of SRR is to retain the replay execution benefits of data-replay (replay only a subset of the original application processes), the more interesting comparison is between SRR and data-replay.

Since SRR only records inter-group messages, whereas data-replay has to record all messages, SRR log sizes are strictly lower than data-replay. The degree to which SRR improves log size overhead, though, depends upon the communication pattern of the application. For the applications with group communication locality, the SRR log size is only 38% of data-replay on average, or just a third of the log size required by data-replay.

With all-to-all applications, whose communications are uniformly distributed across all processes, SRRstill provides some benefit. If the replay group contains k of the total n processes, the expected log size reduction with SRR is k/n relative to data-replay. For the FT, GE and ASP applications, we therefore expect SRR to reduce the log size by $1/8 \approx 12.5\%$ relative to data-replay, where $k = 8, n = 64$ in our experiments. Figure 8 shows that SRR logs for FT, GE and ASP are 13%, 12.8%, and 13.1% smaller than data-log, respectively, or 13% on average. These results match the theoretical analysis very well.

PU benefits least from SRR, whose log only decreases about 1.6%. Since almost every process is in its own replay group, this communication pattern makes SRR behave almost identically with data-replay.

7.3 Replay Group Size and Membership

Finally, we evaluate the sensitivity of the record log size generated by MPIWiz to the replay group size and membership. The size of the replay group represents a tradeoff to the application developer: a larger group size produces smaller logs (less inter-group traffic needs to be recorded), but requires more resources during

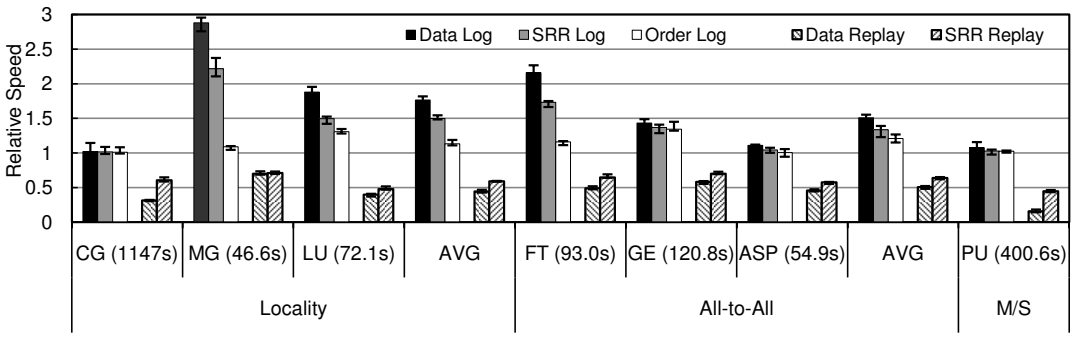


Figure 6: Record and replay speeds relative to base execution time (beside the application name on the x-axes). *Data Log*, *SRR log* and *Order Log* mean the record phases for data-replay, SRR and order-replay respectively. *AVG:* geometric mean for applications of the current communication pattern.

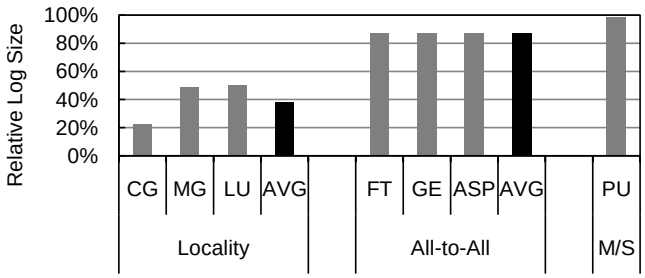
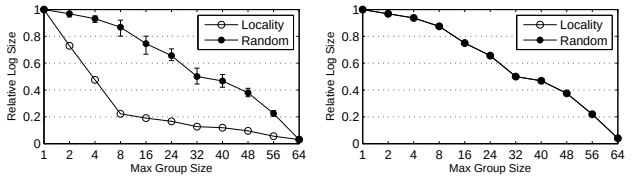


Figure 8: SRR's log size compared to data-log. *AVG:* geometric mean for applications of the current communication pattern.



(a) CG (b) FT

Figure 9: Log size as a function of replay group size and membership relative to log size of data-replay. *Locality:* organize replay groups according to communication locality; *Random:* organize replay groups randomly. With *Random*, we show the average and standard deviation from 5 trials.

replay (more processes have to execute again during replay). Log size can also be sensitive to process membership since processes that exhibit communication locality will generate more inter-group traffic if they are not placed in the same replay group.

We explore these issues with the following experiment. For a given application, we vary the size of a replay group from one process (equivalent to data-replay) to 64 processes (equivalent to order-replay). For each replay group size, we determine the process membership of the group using two methods: according to group communication locality (Section 4), and uniform random selection as a baseline.

Figure 9 shows the results of this experiment for two applications, one with communication locality (CG) and another without locality (FT), in two graphs. The x-axes show the size of the replay group, and the y-axes show the log size relative to a group size of one process (data-replay). The two curves correspond to group membership based on locality and using a random assignment; for random, we performed 3 trials and show the average and standard deviation of the trials.

The results in Figure 9 confirm that larger groups produce smaller logs. For random group assignment, the log size decreases roughly linearly with the size of the group. For applications with locality, however, MPIWiz can do much better.

Figure 9a shows the benefits of making informed group membership assignments for applications that exhibit communication locality. Relative to a random assignment, exploiting communication locality substantially reduces the log size. For a group size of eight, for example, the log size using locality is 2.8 times smaller than with random. Figure 9b, however, shows that applications lacking communication locality are insensitive to group membership. The group membership assignment using graph partitioning results in the same log sizes as using a random assignment.

8. CONCLUSION

This paper proposes a new deterministic replay method, subgroup reproducible replay (SRR), for making deterministic replay practical for MPI applications. SRR balances the tradeoffs of both data-replay and order-replay. It partitions processes into disjoint *replay groups* and allows any subset of these groups to be recorded and replayed. We have implemented a prototype, MPIWiz, to demonstrate and evaluate the SRR approach to deterministic replay.

By partitioning processes into replay groups, SRR can fully exploit communication locality in MPI applications to further reduce recording overhead. When using MPIWiz on popular MPI benchmarks, for example, SRR can reduce recording overhead by over a factor of four relative to data-replay.

Replay groups also make the replay phase more feasible relative to order-replay. An important advantage of data-replay is that it can replay any process of an application individually on a single machine, whereas order-replay requires all processes of the original application to be replayed together. SRR strikes a balance between the two. It only requires replaying the processes of a single replay group, enabling practical replay on one multi-core machine rather than an entire cluster as with order-replay.

MPIWiz provides two additional benefits not found in existing MPI replay tools. In addition to handling the non-determinism in MPI operations, it also handles non-determinism due to system calls like `gettimeofday`. And it guarantees that the memory footprints of the replayed processes are identical to those of the original processes. These features further increase the practicality

of deterministic replay for MPI applications.

Finally, although MPIWizis a fully functional replay tool, it currently does not support checkpointing. Checkpointing was not critical for evaluating SRR relative to data-replay and order-replay, but we recognize that checkpoint and restart is important for many applications and are implementing them as future work.

9. REFERENCES

[1] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report Report NAS-95-020, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, Mail Stop T 27 A-1, Moffett Field, CA 94035- 1000, USA, Dec. 05 1995.

[2] A. Bouteiller, G. Bosilca, and J. Dongarra. Retrospect: Deterministic Replay of MPI Applications for Interactive Distributed Debugging. In *14th European PVM/MPI User's Group Meeting*, pages 297–306, 2007.

[3] C.-K. Cheng and Y.-C. A. Wei. An improved two-way partitioning algorithm with stable performance. *IEEE Transactions on Computer Aided Design*, 10(12):1502–1511, 1991.

[4] C. Clémençon, J. Fritscher, M. J. Meehan, and R. Rühl. An implementation of race detection and deterministic replay with MPI. In *EuroPar'95*, pages 155–166, Aug. 1995.

[5] J. C. de Kergommeaux, M. Ronsse, and K. D. Bosschere. MPL*: Efficient Record/Play of Nondeterministic Features of Message Passing Libraries. In *6th European PVM/MPI Users' Group Meeting*, pages 141–148, 1999.

[6] J. DeSouza, B. Kuhn, B. R. de Supinski, V. Samofalov, S. Zheltov, and S. Bratanov. Automated, scalable debugging of MPI programs with Intel Message Checker. In *SE-HPCS'05*, pages 78–82, 2005.

[7] C. Falzone, A. Chan, E. L. Lusk, and W. Gropp. Collective Error Detection for MPI Collective Operations. In *PVM/MPI'05*, pages 138–147, 2005.

[8] A. Faraj and X. Yuan. Communication Characteristics in the NAS Parallel Benchmarks. In *PDCS'02*, pages 724–729, 2002.

[9] J. Garbers, H. J. Prömel, and A. Steger. Finding clusters in vlsi circuits. In *IEEE International Conference on Computer Aided Design*, pages 520–523, Nov. 1990.

[10] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[11] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An Application-Level Kernel for Record and Replay. In *OSDI'08, To appear*, 2008.

[12] W. Haque. Concurrent deadlock detection in parallel programs. *Int. J. Comput. Appl.*, 28(1):19–25, 2006.

[13] HPCC. Hpcc 1998 blue book (computing, information, and communications: Technologies for the 21st century). Computing, Information, and Communications (CIC) R&D Subcommittee of the National Science and Technology Council's Committee on Computing, Information, and Communications (CCIC), 1998.

[14] Z. Huang, M. K. Purvis, and P. Werstein. Performance Evaluation of View-Oriented Parallel Programming. In *ICPP'05*, pages 251–258, 2005.

[15] NAS Parallel Benchmarks: ProActive implementations. `http://proactive.inria.fr/index.php?page=nas_benchmarks`.

[16] G. Karypis and V. Kumar. Multilevel k-way Partitioning Scheme for Irregular Graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.

[17] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. *ACM SIGPLAN Notices*, 34(8):131–140, 1999.

[18] J. Kim and D. J. Lilja. Characterization of Communication Patterns in Message-Passing Parallel Scientific Application Programs. In *CANPC'98*, pages 202–216, 1998.

[19] B. Krammer and M. S. Müller. MPI Application Development with MARMOT. In *ParCo'05*, pages 893–900, 2005.

[20] D. Kranzlmüller, C. Schaubschläger, and J. Volkert. An Integrated Record&Replay Mechanism for Nondeterministic Message Passing Programs. In *8th European PVM/MPI Users' Group Meeting*, pages 192–200, 2001.

[21] D. Kranzlmüller and J. Volkert. NOPE: A Nondeterministic Program Evaluator. In *ACPC'99*, pages 490–499, 1999.

[22] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Trans. Computers*, 36(4):471–482, 1987.

[23] R. Lovas and P. Kacsuk. Correctness Debugging of Message Passing Programs Using Model Verification Techniques. In *14th European PVM/MPI User's Group Meeting*, pages 335–343, 2007.

[24] G. R. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, and Y. Zou. MPI-CHECK: a tool for checking Fortran 90 MPI programs. *Concurrency and Computation: Practice and Experience*, 15(2):93–100, 2003.

[25] M. Maruyama, T. Tsumura, and H. Nakashima. Parallel Program Debugging based on Data-Replay. In *PDCS'05*, pages 151–156, 2005.

[26] G. L. Miller, S.-H. Teng, and S. A. Vavasis. A unified geometric approach to graph separators. In *32th Annual Symposium on Foundations of Computer Science*, pages 538–547, Oct. 1991.

[27] SIM-MPI Library. `http://www.hpctest.org.cn/resources/sim-mpi.tgz`.

[28] N. Neophytou and P. Evripidou. Net-dbx: A Web-Based Debugger of MPI Programs Over Low-Bandwidth Lines. *IEEE Trans. Parallel Distrib. Syst.*, 12(9):986–995, 2001.

[29] S. Pervez, G. Gopalakrishnan, R. M. Kirby, R. Palmer, R. Thakur, and W. Gropp. Practical Model-Checking Method for Verifying Correctness of MPI Programs. In *14th European PVM/MPI User's Group Meeting*, 2007.

[30] PGDBG Graphical Symbolic Debugger. `http://www.pgroup.com/products/pgdbg.htm`.

[31] M. Rudgyard. Novel techniques for debugging and optimizing parallel applications. In *SC'06*, page 281, 2006.

[32] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. In *International Conference on Dependable Systems and Networks (DSN 2006)*, pages 249–258, 2006.

[33] S. F. Siegel. Model Checking Nonblocking MPI Programs. In *8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2007)*, pages 44–58, 2007.

[34] S. F. Siegel and G. S. Avrunin. Modeling wildcard-free mpi programs for verification. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP 2005)*, pages 95–106, 2005.

[35] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2006)*, pages 157–168, 2006.

[36] Totalview. `http://www.totalviewtech.com/`.

[37] J. L. Träff and J. Worringen. Verifying Collective MPI Calls. In *11th European PVM/MPI Users' Group Meeting*, pages 18–27, 2004.

[38] J. S. Vetter and B. R. de Supinski. Dynamic Software Testing of MPI Applications with Umpire. In *SC'00*, pages 70–70, November, 4–10 2000.

[39] J. S. Vetter and F. Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. *J. Parallel Distrib. Comput.*, 63(9):853–865, 2003.