A Comprehensive User-level Checkpointing Strategy for MPI Applications *

Technical Report # 2007-1, Department of Computer Science and Engineering, University at Buffalo, SUNY

John Paul Walters
Department of Computer Science
Wayne State University
Detroit, MI 48201
jwalters@wayne.edu

Vipin Chaudhary
Computer Science & Engg
University at Buffalo, SUNY
Buffalo, NY 14260
vipin@buffalo.edu

Abstract

As computational clusters increase in size, their mean-time-to-failure reduces drastically. After a failure, most MPI checkpointing solutions require a restart with the same number of nodes. This necessitates the availability of multiple spare nodes, leading to poor resource utilization. Moreover, most techniques require a central storage for storing checkpoints. This results in a bottleneck and severely limits the scalability of checkpointing.

We propose a scalable fault-tolerant MPI based on LAM/MPI which supports user-level checkpointing, migration, and replication. Our contributions extend the existing state of fault-tolerant MPI with asynchronous replication, eliminating the need for central or network storage. We evaluate both centralized storage and SAN-based solutions and show that they are not scalable, particularly after 64 CPUs. Our migration strategy is the first to make no assumptions on restart topologies, eliminating the need for spare nodes. We demonstrate the low overhead of our checkpointing and replication scheme with the NAS Parallel Benchmarks and the High Performance LINPACK benchmark with tests up to 256 nodes. We show that checkpointing and replication can be achieved with much lower overhead than current techniques and near transparency to the end user while still providing fault resilience.

*This research was funded in part by NSF IGERT grant 9987598, NSF ITR grant 0081696, MEDC Michigan Life Science Corridor Grant, Institute for Scientific Computing at Wayne State University, Sun Microsystems, and New York State Office of Science, Technology and Academic Research.

1 Introduction

Computational clusters with hundreds and thousands of processors are fast-becoming ubiquitous in large-scale scientific computing leading to lower mean-time-to-failure. This forces the systems to effectively deal with the possibility of arbitrary and unexpected node failure. Since MPI [19] provides no mechanism to recover from a failure, a single node failure will halt the execution of the entire MPI world. Thus, there exists great interest in the research community for a truly fault-tolerant MPI implementation that is transparent and does not depend on a heavy-weight grid/adaptive middleware ([22, 39]).

Several groups have included checkpointing within various MPI implementations. MVAPICH2 now includes support for kernel-level checkpointing of Infiniband MPI processes [20]. Sankaran et al. also describe a kernel-level checkpointing strategy within LAM/MPI [8, 32, 33].

However, such implementations suffer from two major drawbacks: (1) a reliance on a common network file system or dedicated checkpoint servers. (2) Failure recovery requires either the insertion of a replacement node (to maintain topology) or an adaptive grid/middleware (e.g. Charm++/AMPI [25, 21, 22]).

We consider both the reliance on a network file system/checkpoint servers and the insistence on a common restart topology to be a fundamental limitation of existing checkpoint systems. Storing directly to network storage incurs too great an overhead. Using dedicated checkpoint servers saturates the network links of a few machines, resulting in degraded performance. And the reliance on a common restart topology requires a set of spare nodes that may not be available.

Because most fault-tolerance implementations shown in the literature consider clusters of inadequate size (typically fewer than 64 nodes, more often 8-16 nodes) or inadequately small benchmarks (class B sizes of the NAS Parallel Benchmarks [38]) scalability issues regarding checkpointing are rarely considered. We specifically test the scalability of our implementation up to 256 nodes with individual node memory footprints of 1 GB each.

We show that the overhead of checkpointing either to network storage or to dedicated checkpoint servers is too severe for large-scale computational clusters. As such, we make two contributions in this paper:

- We propose and implement a checkpoint replication system, which distributes the overhead of checkpointing evenly over all nodes participating in the computation. This significantly reduces the impact of heavy I/O on network storage.

- We introduce our migration/restart technique which allows us to restart any checkpoint on any machine, regardless of its original location without requiring any spare nodes. In so doing, we are able to effectively move the computation to the data. This provides for drastic improvements in restart times over strategies that require data movement over the network.

The remainder of this paper is outlined as follows: in Section 2 we provide a brief introduction to LAM/MPI and checkpointing. In Section 3 we discuss our user-level checkpointing solution within the context of the LAM/MPI implementation. In Section 4 we compare existing checkpoint storage strategies

and evaluate our proposed replication technique. In Section 5 we provide a brief overview of the work related to this project. Finally, in Section 6 we present our conclusions.

2 Background

2.1 LAM/MPI

LAM/MPI [8] is a research implementation of the MPI-1.2 standard [19] with portions of the MPI-2 standard. LAM uses a layered software approach in its construction [34]. In doing so, various modules are available to the programmer that tune LAM/MPI's runtime functionality including TCP, Infiniband [23], Myrinet [28], and shared memory communication.

The most commonly used module is the TCP module which provides basic TCP communication between LAM processes. A modification of this module, CRTCP, provides a bookmark mechanism for checkpointing libraries to ensure that a message channel is clear. LAM uses the CRTCP module for its built-in checkpointing capabilities.

2.2 Checkpointing Distributed Systems

Checkpointing itself can be performed at several levels. In kernel-level checkpointing [20, 16, 8, 32, 33], the checkpointer is implemented as a kernel module, making checkpointing fairly straightforward. However, the checkpoint itself is heavily reliant on the operating system (kernel version, process IDs, etc.). User-level checkpointing [40, 41] performs checkpointing using a checkpointing library, enabling a more portable checkpointing implementation at the cost of limited access to kernel-specific attributes (e.g. user-level checkpointers cannot restore process IDs). At the highest level is application-level checkpointing [7, 6] where code is instrumented with checkpointing primitives. The advantage to this approach is that checkpoints can often be restored to arbitrary architectures. However, application-level checkpointers require access to a user's source code and do not support arbitrary checkpointing. Thus, a user's code must be instrumented with checkpoint locations (often inserted by hand by the programmer) after which a preprocessor adds in portable code to save the application's state. Unlike kernel-level and user-level checkpointing, application-level checkpointers are not signal driven. Thus, in order for a checkpoint to be taken (in the application-level case), a checkpoint location must first be reached. Such is not the case with kernel-level or user-level strategies.

There are two major checkpointing/rollback recovery techniques: coordinated checkpointing and message logging. Coordinated checkpointing requires that all processes come to an agreement on a consistent state before a checkpoint is taken. Upon failure, all processes are rolled back to the most recent checkpoint/consistent state.

Message logging requires distributed systems to keep track of interprocess messages in order to bring a checkpoint up-to-date. Checkpoints can be taken in a non-coordinated manner, but the overhead of logging the interprocess messages can limit its utility. Elnozahy et al. provide a detailed survey of the various rollback recovery protocols that are in use today [17].

3 User-level LAM Checkpointing with Arbitrary Restart Structure

We are not the first group to implement checkpointing within the LAM/MPI system. Three others [41, 33, 37] have added checkpointing support. We begin with an overview of the existing LAM/MPI checkpointing implementations.

3.1 Existing implementations

Sankaran et al. first added checkpointing support within the LAM system [33] by implementing a lightweight coordinated blocking module to replace LAM's existing TCP module. The protocol begins when *mpirun* instructs each LAM daemon (*lamd*) to checkpoint its MPI processes (see Figure 1). When a checkpoint signal is delivered to an MPI process, each process exchanges bookmark information with all other MPI processes. These bookmarks contain the number of bytes sent to/received from every other MPI process. With this information, any in-flight messages can be waited on and received before the checkpoint occurs.

After acquiescing the network channels, the MPI library is locked and a checkpointing thread assumes control. The Berkeley Linux Checkpoint/Restart library (BLCR) [16] is used as a kernel-level checkpointing engine. Each process checkpoints itself using BLCR (including *mpirun*) and the computation resumes.

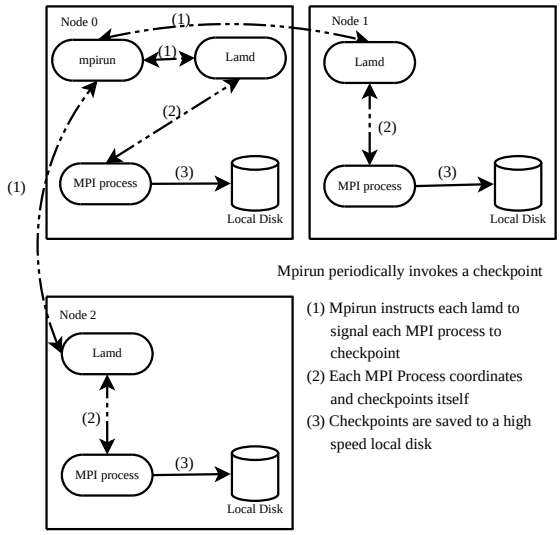


Figure 1. Overview of the checkpointing module. A dashed line indicates asynchronous LAM communication, while a solid line indicates synchronous communication as well as checkpointing to disk.

When a node fails, the user restarts the checkpointed *mpirun* which automatically restarts the application using an *application schema* to maintain the original topology. The MPI library is then reinitialized and computation resumes.

Zhang et al. describe a user-level checkpointing solution, also implemented within LAM [41]. Their checkpointer (*libcsm*) is signal-based rather than thread-based. But otherwise, their implementation is identical to that of Sankaran's. Neither implementation includes support for automatic/periodic checkpointing, limiting their use in batch environments.

A second problem with the above solutions is that both require exactly the same restart topologies. If, for example, a compute node fails, the system cannot restart by remapping checkpoints to existing nodes. Instead, a new node would have to be inserted into the cluster to force the restart topology into consistency with the original checkpoint topology. This requires the existence of spare nodes that can be inserted into the MPI world to replace failed nodes. If no spare nodes are available, the computation cannot be restarted.

Two previous groups have attempted to solve the problem of migrating LAM checkpoint images [10]. Cao et al. propose a migration scheme based on the BLCR work [33] by Sankaran et al. Their technique towards migrating LAM checkpoints requires a tool to parse through the binary checkpoint images, find the MPI process location information, and update the node IDs.

Wang, et al. propose a pause/migrate solution where spare nodes are used for migration purposes when a LAM daemon discovers an unresponsive node [37]. Upon detecting a failure, their system migrates the failed processes via a network file system to the replacement nodes before continuing the computation. Their solution incorporates periodic checkpointing, but still requires compatible restart topologies.

Our solution diverges from the above in that: (1) we make no assumptions as to the restart topology and, consequently, we do not require the use of replacement nodes. (2) We add support for automatic periodic checkpointing through a checkpointing daemon. (3) Most importantly, we do not require the use of a network storage facility for checkpointing or migration/restart.

3.2 Enhancements to LAM's Checkpointing

Our checkpointing solution uses the same coordinated blocking approach as Sankaran and Zhang's techniques described above. To perform the actual checkpointing, we use Victor Zandy's *Ckpt*, a 32-bit user-level checkpointer [40].

In Figure 2 we provide an overview of the steps taken within each MPI process in order to ensure a consistent checkpoint. Figure 2(a) is the same basic checkpoint strategy that is used in [33, 41]. Where our implementation differs is in the "restart base" and "GPS coordinate update" portions of Figure 2(b) where process-specific data is updated before continuation.

In the "restart base" portion of Figure 2(b) we update all process-specific information that has changed due to a restart. This includes updating process IDs, environment variables, and rebuilding messaging channels. In order to support process migration, we added a phase to LAM's restart sequence. We depict this as "update GPS coordinates" in Figure 2(b). LAM uses a GPS structure (Figure 3) to maintain vital identification for each MPI process. Thus, for every process in the MPI world, there exists a GPS entry, and each process maintains a local copy of the entire GPS array. A migration that varies the MPI topology will alter elements of the GPS array as processes are assigned to alternate nodes (the *gps_node* field in Figure 3). Our strategy is to mimic an *MPI_Init* upon restart in order to trigger a GPS cache update from

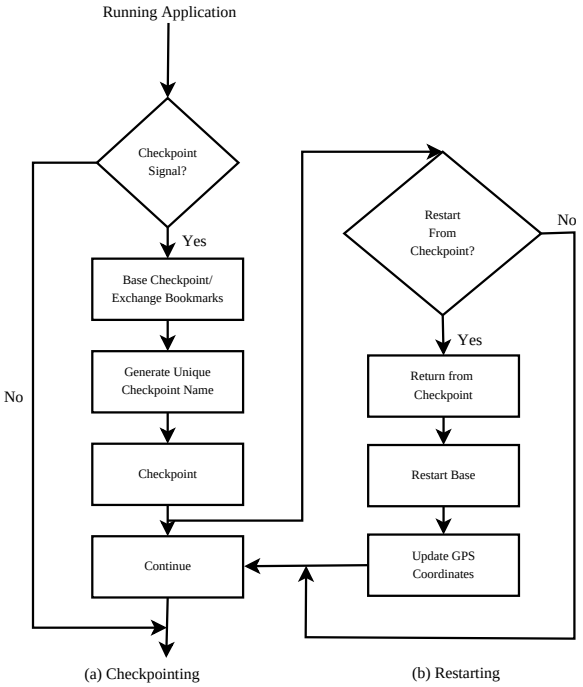


Figure 2. (a) Checkpointing a LAM/MPI process, and (b) Restarting

```
struct _gps {
        int4 gps_node;    /* node ID */
        int4 gps_pid;     /* process ID */
        int4 gps_idx;     /*process index */
        int4 gps_grank;   /*glob. rank in loc. world */
};
```

Figure 3. LAM GPS struct for process info

mpirun. This facilitates the arbitrary migration of MPI processes without relying on a tool to manually parse checkpoints.

In allowing for proper checkpoint migration, we facilitate faster restart solutions than have previously been possible. Because our solution does not rely on checkpointing to network storage, all checkpoints can be read directly from the host's faster local hard disk. Further, by indexing which checkpoints are stored on which nodes, we can build a customized *application schema* that is based on the current location of checkpoints within the cluster. After querying each node, we build an index of the checkpoints contained on each node which ultimately forms the *application schema*. The *application schema* specifies that each node should first start its previous checkpoint. Afterwards, any checkpoints that have not been started are mapped onto the node with the (currently) lowest restart load. When all checkpoints have been accounted for, the computation resumes. In the coming sections, we describe how our replication technique helps to ensure that checkpoints will survive through multiple simultaneous node failures without the need to write checkpoint images to network storage.

4 Checkpoint Storage, Resilience, and Performance

In order to enhance the resiliency of our checkpointing we include data replication. While not typically stated explicitly, nearly all checkpoint/restart methods rely on the existence of network storage that is accessible to the entire cluster. Such strategies suffer from two major drawbacks in that they create a single point of failure and also incur massive overhead when compared to checkpointing to local disks.

A cluster that utilizes a network file system-based checkpoint/restart mechanism may sit idle should the file system experience an outage. This leads not only to wasteful downtime, but also may lead to lost data should the computation fail without the ability to checkpoint. However, even with fault-tolerant network storage, simply writing large amounts of data to such storage represents an unnecessary overhead to the application. In the sections to follow, we examine two replication strategies: a dedicated server technique, and a distributed implementation.

We evaluate both centralized-server and network storage-based storage techniques and compare them against our proposed replication strategy using the SP, LU, and BT benchmarks from the NAS Parallel Benchmarks (NPB) suite [38] and the High Performance LINPACK (HPL) [15] benchmark. The remaining benchmarks within the NPB suite represent computational kernels, rather than mini-applications. We limited our NPB benchmarking focus to the mini-applications. The tests were performed with a single MPI process per node. To gauge the performance of our checkpointing library using the NPB tests, we used exclusively "Class C" benchmarks. For our HPL benchmark, we selected a matrix with a problem size of 28,000. These configurations resulted in checkpoints that were 106MB, 194MB, 500MB, and 797MB for the LU, SP, BT, and HPL benchmarks, respectively. The LU and HPL benchmarks consisted of 8 CPUs each, while the BT and SP benchmarks required 9 CPUs.

For our implementation testing we used a university owned cluster consisting of 16 dual 2.66 GHz Pentium IV Xeon processors with 2.5 GB RAM, a 10,000 RPM Ultra SCSI hard disk and gigabit ethernet. A 1 TB IBM DS4400 SAN was also used for the network storage tests. To test for scalability we tested our implementation up to 256 nodes on a second university cluster consisting of 1600 3.2 GHz Intel Xeon processors, with 2 processors per node (800 total nodes) and a 30 TB EMC Storage Area Network. The network is connected by both gigabit ethernet and Myrinet. Gigabit ethernet was used for our tests.

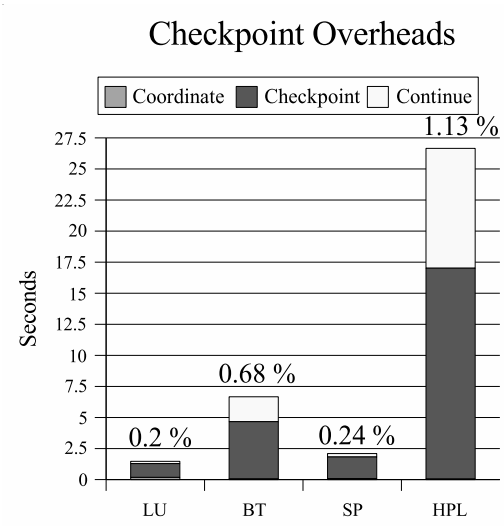
As our baseline, we compare the three storage techniques against the checkpoint data shown in Figure 4. Here we show the result of periodically checkpointing the NAS Parallel Benchmarks as well as the HPL benchmark along with the time taken to perform a single checkpoint. Our implementation shows very little overhead even when checkpointed at 1 minute intervals. The major source of the overhead of our checkpointing scheme lies in the time taken in writing the checkpoint images to the local file system.

In Figure 4(a) we break the checkpointing overhead down by coordination time, checkpointing time, and continue time. The percentages listed above each column indicate the overhead of a checkpoint when compared to the baseline running time of Figure 4(b). The coordination phase corresponds to the "exchange bookmarks/checkpoint base" segment of Figure 2(a). The checkpoint time consists of the time needed to checkpoint the entire memory footprint of a single process and write it to stable storage. Finally,

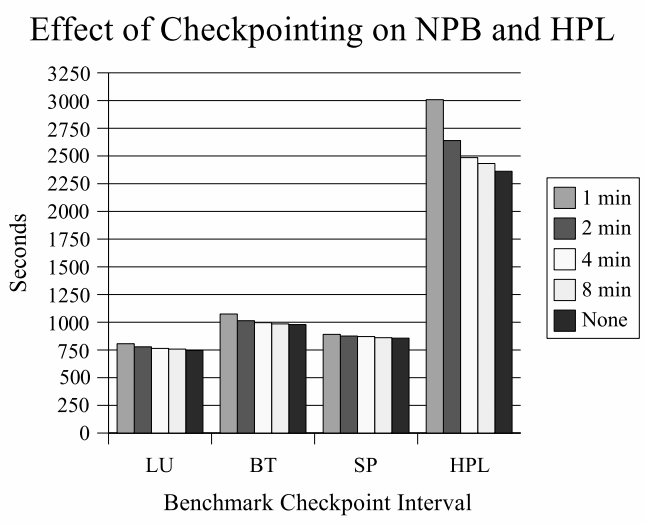
the continue phase corresponds to the "continue" portion of Figure 2(a). The coordination and continue phases require barriers to ensure application synchronization, while each process performs the checkpoint phase independently.

As confirmed in Figure 4(a), the time required to checkpoint the entire system is largely dependent on the time needed to checkpoint the individual nodes. Writing the checkpoint file to disk represents the single largest time in the entire checkpoint process and dwarfs the coordination phase. Thus, as the memory footprint of an application grows, so too does the time needed to checkpoint. This can also affect the time needed to perform the *continue* barrier as faster nodes are forced to wait for slower nodes to write their checkpoints to disk. This suggests that the key to fast and light-weight checkpointing is in minimizing the checkpoint/disk-writing time.

From Figure 4(b) we confirm that the memory footprint of an application is the dominating factor in the overall checkpointing overhead. Nevertheless, checkpointing the 500MB BT benchmark every minute resulted in less than 10% additional overhead. Reducing the checkpoint interval to 2 minutes further reduced the overhead of the BT benchmark by more than half.



(a) % indicates the contribution of checkpointing (in terms of overhead) over the base timings without checkpointing (from Figure 4(b)).



(b)

Figure 4. A breakdown of checkpointing overheads.

4.1 Dedicated Checkpoint Servers versus Checkpointing to Network Storage

The two most common checkpoint storage techniques presented in the literature are the dedicated server(s) [14] and storing checkpoints directly to network storage [32, 20]. We begin our evaluation with a comparison of these two common strategies.

For our dedicated checkpoint server implementation we use the LAM daemons (*lamd*) to move checkpoints from individual nodes to a dedicated checkpoint server. Each *lamd* was extended with an additional daemon that is used to both collect checkpoint information from each of its MPI processes, and

asynchronously propagate the data to the dedicated server. We have also extended *mpirun* to include a checkpointing daemon responsible for scheduling and receiving checkpoints.

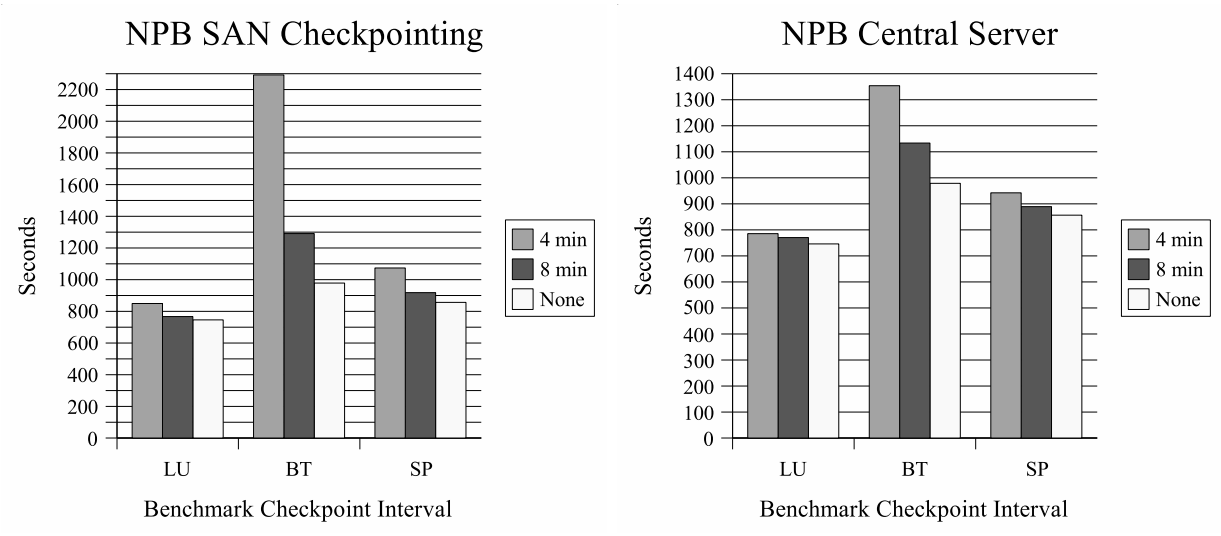


Figure 5. Runtime of NPB with checkpoints streamed to central checkpoint server vs. saving to SAN.

In Figure 5 we show the results of checkpointing the NAS Parallel Benchmarks with the added cost of streaming the checkpoints to a centralized server or storing the checkpoints to a SAN. In the case of the LU benchmark, we notice a marked reduction in overhead when comparing the SAN data in Figure 5(a) to the checkpoint server data also presented in Figure 5(b). Indeed, the overhead incurred by streaming an LU checkpoint every 4 minutes is less than 6% – a dramatic improvement over saving checkpoints to shared storage, which results in an overhead of nearly 14% for LU and 25% for SP. The situation is even worse for the BT benchmark which incurs an overhead of 134% at 4 minute checkpointing intervals.

However, we can also see that as the size of the checkpoint increases, so too does the overhead incurred by streaming all checkpoints to a centralized server. At 8 minutes checkpointing intervals the SP benchmark incurs an overhead of approximately 4% while the overhead of BT jumps to nearly 16%. The increase in overhead is due to individual *lamds* overwhelming the checkpoint server, thereby creating too much network and disk congestion for a centralized approach to handle.

Nevertheless, the use of a dedicated checkpoint server shows a distinct cost-advantage over the SAN-based solution. However, dedicated servers still suffer from being a single point of failure as well as being network bottlenecks. Techniques using multiple checkpoint servers have been proposed to mitigate such bottlenecks [14]. However, their efficacy in the presence of large checkpoint files has not been demonstrated in the literature (NPB class B results are shown in [14]). Furthermore, if each checkpoint server is collecting data from a distinct set of nodes, as is the case in [14], the computation will not be resilient to failed checkpoint servers. Finally, as we show in Figure 5(b), dedicated server strategies simply do not scale. With as few as 8 nodes checkpointing to a single server, overheads as high 16% are observed for 8 minute checkpointing intervals. A technique to alleviate the impact of checkpointing directly to SANs

is proposed in [37]. Wang, et al. combine local checkpointing with asynchronous checkpoint propagation to network storage. However, they require multiple levels of scheduling in order to prevent the SAN from being overwhelmed by the network traffic. The overhead of their scheduling has not yet been demonstrated in the literature, nor has the scalability of their approach, where their tests are limited to 16 nodes.

4.2 Checkpoint Replication

To address the scalability issues shown in Section 4.1, we implemented an asynchronous replication strategy that amortizes the cost of checkpoint storage over all nodes within the MPI world. Again we extended LAM's *lamds*. This time we used a peer-to-peer strategy between each *lamd* to replicate checkpoints to multiple nodes. This addresses both the resiliency of checkpoints to node failure as well as the bottlenecks incurred by transferring data to dedicated servers.

A variety of replication strategies have been used in peer-to-peer systems. Typically, such strategies must take into account the relative popularity of individual files within the network in order to ascertain the optimal replication strategy. Common techniques include the square-root, proportional, and uniform distributions [27]. While the uniform distribution is not used within peer-to-peer networks because it does not account for a file's query probability, our checkpoint/restart system relies on each checkpoint's availability within the network. Thus, each checkpoint object has an equal query probability/popularity and we feel that a uniform distribution is justified for this specific case.

We opted to randomly distribute the checkpoints in order to provide a higher resilience to network failures. For example, a solution that replicates to a node's nearest neighbors would likely fail in the presence of a switch failure. Also, nodes may not fail independently and instead cause the failure of additional nodes within their vicinity. Thus, we feel that randomly replicating the nodes throughout the network provides the greatest possible chance of survival.

To ensure an even distribution of checkpoints throughout the MPI world, we extended the *lamboot* command with a parameter representing the degree of replication (number of replicas). *lamboot* is responsible for computing and informing each *lamd* of its replicas.

Figure 6(a) shows the results of distributing a single replica throughout the cluster with NPB. As can be seen, the overhead in Figure 6(a) is substantially lower than that of the centralized server shown in Figure 5(b). In each of the three NAS benchmarks, we are able to reduce the overhead of distributing a checkpoint by at least 50% when compared to streaming all checkpoints to a single server. For the most expensive checkpoint (BT), we are able to reduce the overhead of checkpointing to 9% at 4 minute intervals and 5.5% at 8 minute intervals (compared to 38% and 16% at 4 minute and 8 minute intervals, respectively).

In Figure 6(b) we show the results of distributing a single replica every 4, 8, 16, and 32 minutes for the HPL benchmark. We found that our network was unable to handle checkpoint distribution of HPL at intervals shorter than 4 minutes, due to the size of the checkpoint files. We notice a steady decrease in overhead as the checkpoint interval increases to typical values with a single checkpoint resulting in an overhead of only 2.2%.

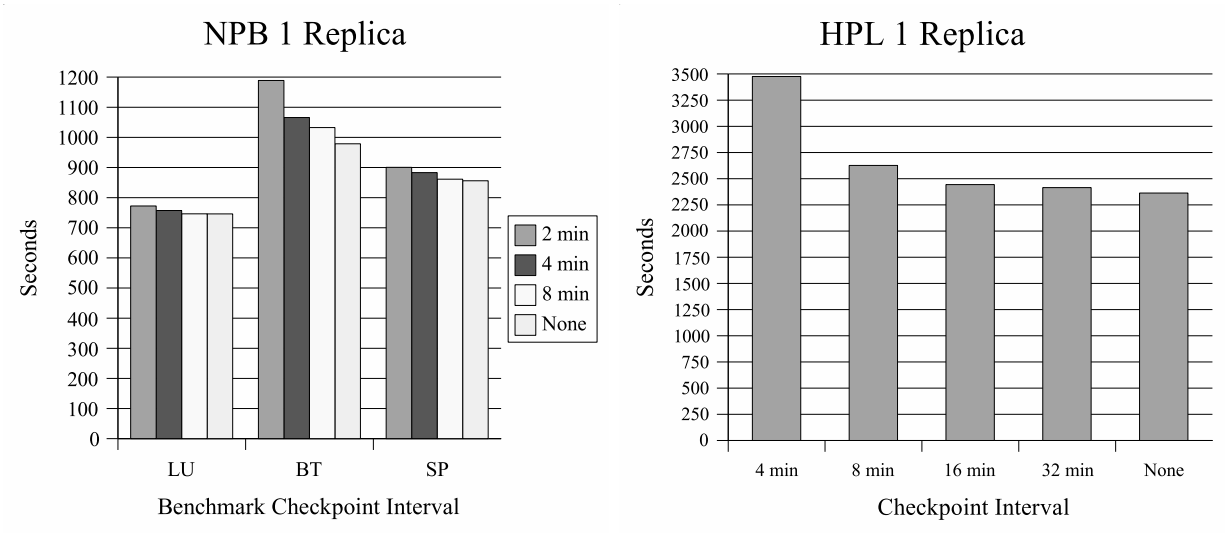


Figure 6. Benchmark timings with one replica.

To address the resiliency of checkpoint replication in the presence of node failure we insert multiple checkpoint replicas into the system. In Figure 7 we compare the overheads of distributing 1, 2, and 3 replicas of each of our three NAS benchmarks every 4 minutes. As we would expect, the overhead incurred is proportional to the size of the checkpoint that is distributed. For smaller checkpoints such as LU and SP, distributing the 3 replicas represents minimal overhead. As the size of the checkpoint increases, however, so too does the overhead as the BT data shows in Figure 7. Even so, we incur comparatively little overhead given the amount of data that is exchanged at every checkpoint, with the LU and SP benchmarks adding only 2% and 6% overhead respectively. On the other hand, the BT overhead for 3 checkpoints jumps to nearly 21%. However, as we show in the scalability tests to follow (Section 4.5), such overhead drops considerably as more typical checkpoint intervals are used.

4.3 The Degree of Replication

While the replication strategy that we have described has clear advantages in terms of reducing the overhead on a running application, an important question that remains is the number of replicas necessary to achieve a high probability of restart. To help answer this question, we developed a simple simulator capable of replicating node failures, given inputs of the network size and the number of replicas.

From Table 1 we can see that our replication strategy enables a high probability of restart with seemingly few replicas needed in the system. Furthermore, this strategy is *ideal* as supercomputing approaches the petascale stage. As can be seen, our replication technique scales quite well with the number of CPUs. With 2048 processors, we estimate that 111 *simultaneous* failures could occur while maintaining at least a 99.9% probability of successful restart and requiring only 4 replicas of each checkpoint. Combining this with our migration system, we can simply absorb any node failures *without requiring any replacement nodes*.

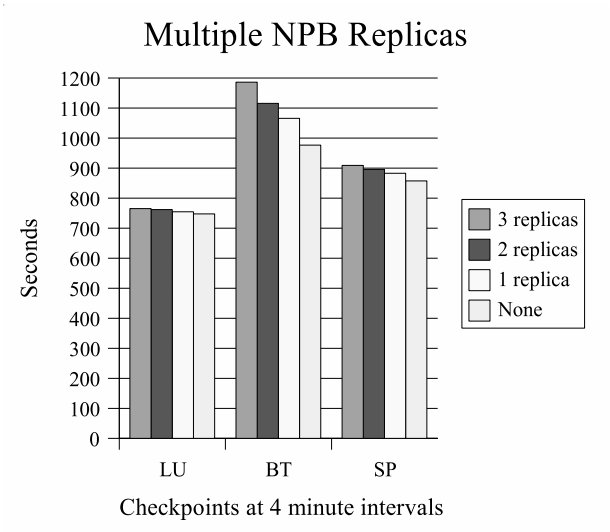


Figure 7. NPB with multiple replicas.

	1 Replica			2 Replicas			3 Replicas			4 Replicas		
	Allowed Failures for			Allowed Failures for			Allowed Failures for			Allowed Failures for		
Nodes	90%	99%	99.9%	90%	99%	99.9%	90%	99%	99.9%	90%	99%	99.9%
8	1	1	1	2	2	2	3	3	3	4	4	4
16	1	1	1	2	2	2	5	4	3	7	5	4
32	2	1	1	5	3	2	8	5	4	11	8	6
64	3	1	1	8	4	2	14	8	4	19	12	8
128	4	1	1	12	6	3	22	13	8	32	21	14
256	5	2	1	19	9	5	37	21	13	55	35	23
512	7	2	1	31	14	7	62	35	20	95	60	38
1024	10	3	1	48	22	11	104	58	33	165	103	67
2048	15	5	2	76	35	17	174	97	55	285	178	111

Table 1. Probability of successful restart with 1-4 replicas.

4.4 Restarting Computation

We provide two mechanisms for restarting the computation. The first, based on the use of an *application schema* was described in Section 3.2. For comparison, we also provide a peer-to-peer lookup implementation. The advantage of the peer-to-peer lookup strategy over the *application schema* strategy is we can guarantee a load-balanced restart. However, the disadvantage is that the lookup scheme is more time consuming.

From the analysis performed by Lv et al. [27] we can describe the average search size to find a replica within our cluster as:

$$Size = \sum_{i=1}^{m} q_i \frac{m}{\rho} = \frac{m}{\rho}$$

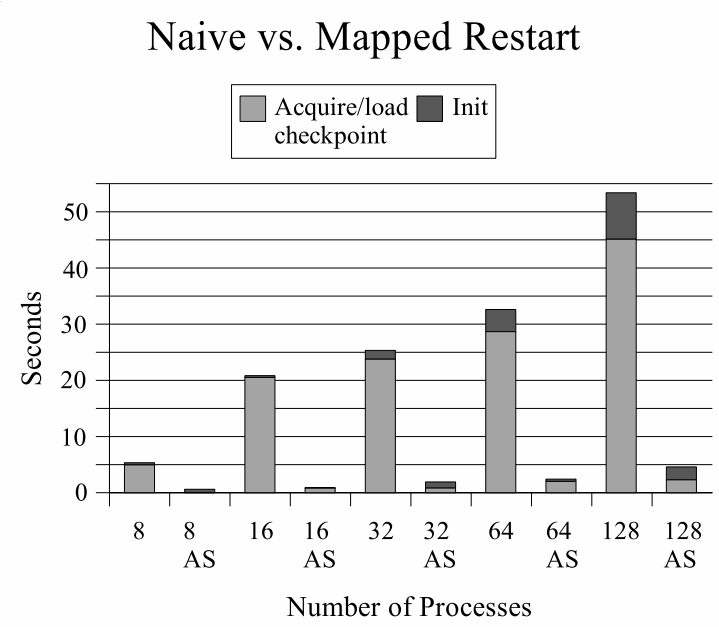


Figure 8. Restart overheads with *application schema* and peer-to-peer lookup. AS = Application Schema.

Where m is the number of checkpoints (one per MPI process) in the cluster, ρ is the average number of replicas of each object per *lamd*, and q_i is the popularity of a given object. In a uniform distribution, objects are uniformly popular. So we can replace q_i with $\frac{1}{m}$. Summing over all m nodes, we are left simply with $\frac{m}{\rho}$ [27].

Using this formula, a user can choose to either optimize for the lowest overhead during the runtime of an MPI computation (ρ near zero), or choose a more fault-tolerant number of replicas which will also reduce the time to restart.

The restart times described in Figure 8 consist of two metrics: the acquisition/loading of the checkpoint image (labeled "acquire/load checkpoint" in Figure 8) and a mock-MPI_Init (labeled "init" in Figure 8). The acquisition/loading phase includes the time necessary to find and transfer any checkpoint images, while the "init" phase consists of reinitializing the MPI library.

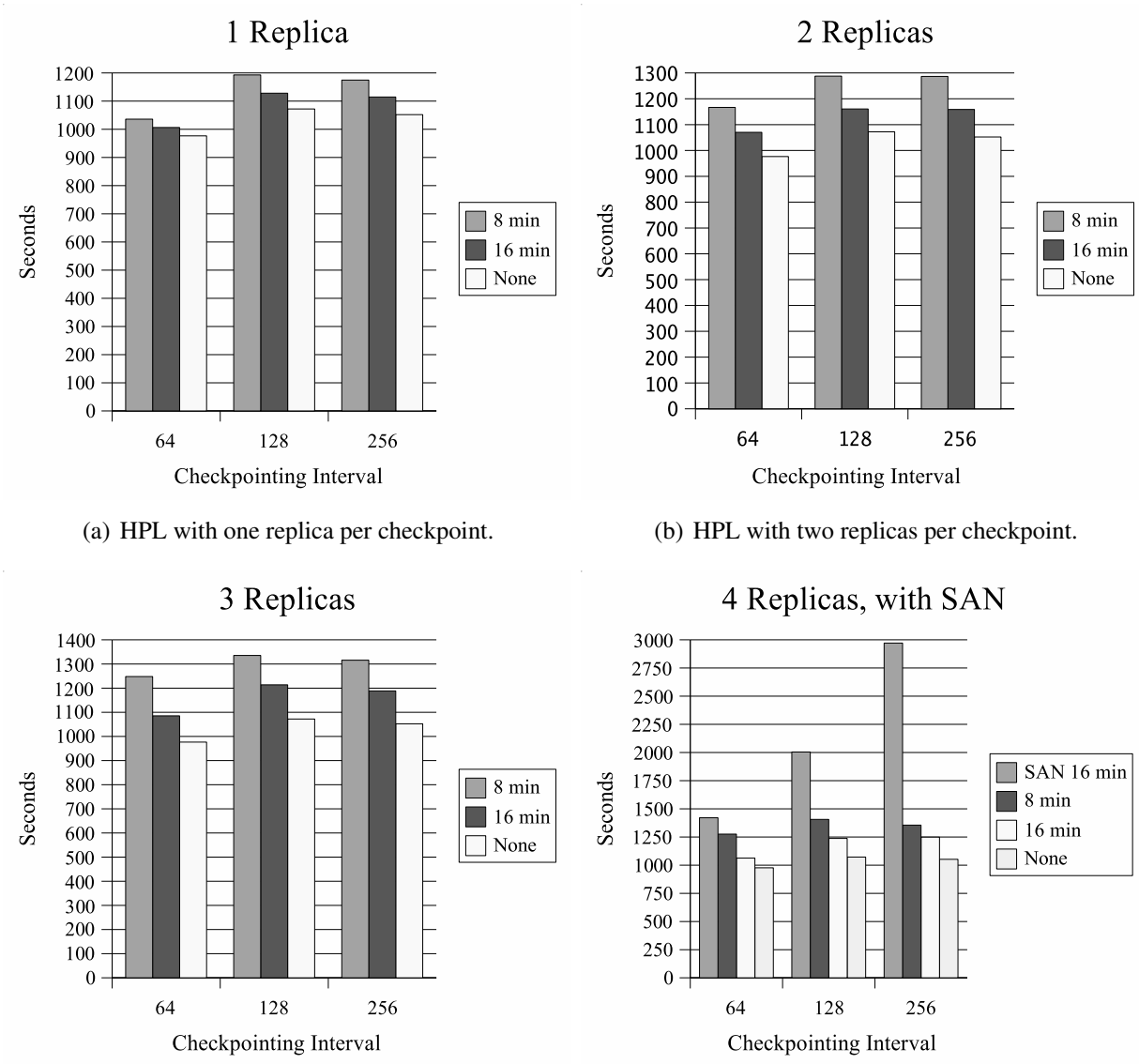
In Figure 8 we demonstrate the time to restart a failed computation using the LU benchmark. We compare both our peer-to-peer lookup as well as our restart *application schema*-based strategy (AS in Figure 8) with up to 128 processes. To simulate a failure/restart, we checkpointed the application, killed the job and measured the time necessary to both find the correct checkpoint and restart it. In all cases, the MPI universe is reduced from 8 nodes to 4 nodes, simulating a 4 node failure and restarting with multiple processes per node[1]. Clearly, our strategy of using an *application schema* results in much faster restarts - checkpoints need not be moved around the network, and instead can simply be restarted. However, as we noted earlier, some applications may tolerate an increased restart time in favor of more ideal resource

[1]Limited resources required the use of only 8 nodes for all tests, up to 128 processes

allocation. As such, we include both mechanisms.

4.5 Scalability Studies

Because our checkpointing engine, *Ckpt* [40], is only 32 bit while the second cluster's Xeon processors are each 64 bit, we simulated the mechanics of checkpointing with an artificial 1 GB file that is created and written to local disk at each checkpoint interval. Aside from this slight modification, the remaining portions of our checkpointing system remain intact (coordination, continue, file writing, and replication).



(a) HPL with one replica per checkpoint.

(b) HPL with two replicas per checkpoint.

(c) HPL with three replicas per checkpoint.

(d) HPL with four replicas per checkpoint compared with EMC SAN.

Figure 9. Scalability tests using the HPL benchmark.

In Figure 9 we demonstrate the impact of our checkpointing scheme. Each number of nodes (64, 128, and 256) operates on a unique data set to maintain a run time of approximately 1000 seconds. For comparison, we also present the overhead of checkpointing to the EMC SAN in Figure 9(d). Large scale

evaluation of the centralized server technique was not possible due to the limited disk capacity of the compute nodes. We chose to evaluate our system for up to 4 checkpoints as the results of our failure simulation (see Table 1) suggest that 4 replicas achieves an excellent restart probability.

The individual figures in Figure 9 all represent the total run time of the HPL benchmark at each cluster size. Thus, comparing the run times at each replication level against the base run time without checkpointing gives us a measure of the overhead involved for each replication level. From Figure 9(a) we can see that the replication overhead is quite low - only approximately 6% for 256 nodes or 3% for 64 nodes (at 16 minute checkpoint intervals). Similar results can be seen at 2, 3, and 4 replicas with only a minimal increase in overhead for each replication increase.

The most important results, however, are those shown in Figure 9(d). Here we include the overhead data with 4 replicas as well as with checkpointing directly to the SAN (a common strategy in nearly all MPI checkpointing literature). As can be seen, the overhead of checkpointing directly to a SAN not only dwarfs that of our distributed replication strategy but also nullifies the efficacy of additional processors for large clusters.

5 Related Work

Checkpointing at both the user-level and kernel-level has been extensively studied [29, 16]. The official LAM/MPI implementation includes support for checkpointing [32] using the BLCR kernel-level checkpointing library [16]. A more recent implementation by Zhang et al. duplicates the functionality of LAM's checkpointing, but implements the checkpointing at the user-level [41]. Wang, et al. have implemented additional pause/restart functionality in LAM [37]. However, all current LAM implementations rely on network storage and a similar process topology.

Other MPI implementations have been similarly enhanced with checkpointing support. MPICH-GM, a Myrinet specific implementation of MPICH has been extended to support user-level checkpointing [24]. Similarly, Gao et al. [20] demonstrate a kernel-level checkpointing scheme for Infiniband (MVAPICH2) that is based on the BLCR kernel module [16].

DejaVu [30] implements an incremental checkpoint/migration scheme that is able to incrementally capture the differences between two checkpoints to minimize the size of an individual checkpoint. However, DejaVu is currently unavailable and results presented in the literature are limited to 16 nodes making the scalability of their system unknown.

MPICH-V [5] uses an uncoordinated message logging strategy with checkpointing provided by the Condor checkpointing library [26]. CoCheck [35] also uses the Condor checkpointing library as the basis for the tuMPI checkpointing system. More recent work has demonstrated a blocking coordinated protocol within MPICH2 [14]. Their observations suggested that for high speed computational clusters blocking approaches achieve the best performance for sensible checkpoint frequencies. Our scalability results from Section 4.5 lend additional evidence supporting their claim.

Using Charm++ [25] and Adaptive-MPI [21, 22], Chakravorty et al. add fault tolerance via task mi-

gration to the Adaptive-MPI system [11, 13, 31, 12]. Their system relies on processor virtualization to achieve migration transparency.

MPICH-GF [39] is a grid-enabled version of MPICH that runs on top of the Globus grid-middleware. It uses a user-level coordinated checkpointing strategy to ensure consistent recovery in the case of node failure. MPICH-GF supports task migration; however, such migration relies on the Globus toolkit.

Other strategies such as application-level checkpointing have also been extended to MPI checkpointing, particularly the C^3 [7, 6] system. Application-level checkpointing carries advantages over kernel-level or user-level in that it is more portable and often allows for restart on varying architectures. However they do not allow for arbitrary code checkpointing and require access to a user's source code.

Network-specific fault-tolerance schemes have also been addressed [2, 3]. These address the delivery of messages in the face of node failures. FT-MPI [18] implements these network fault-tolerance protocols, but requires that the user supply an application-level checkpointing package in order to facilitate the restart of computations. LA-MPI [4] similarly requires that users perform self-checkpointing.

Scalable MPI job initializing has also been studied [36, 9]. While typical MPI implementations, including LAM/MPI, are unable to quickly start jobs where the number of processes number in the hundreds or thousands, work has been proposed in the literature to facilitate such scalability. Our work goes hand-in-hand with such enhancements.

Our work differs from the above in that we provide task migration and redistribution on the standard LAM/MPI implementation. We handle checkpoint redundancy for added resiliency in the presence of node failures. Our checkpointing solution does not rely on the existence of network storage for checkpointing. The absence of network storage allows for improved scalability and also reduced checkpoint intervals (where desired). Unlike others [24, 20], we focus on ethernet based communication due to its prevalence in large computational cluster [1].

6 Conclusions and Future Work

We have shown that it is possible to effectively checkpoint MPI applications using the LAM/MPI implementation with very low overhead. Previous checkpointing implementations have typically neglected the issue of checkpoint replication and process migration. We addressed both of these issues with an emphasis on transparency to the end-user and low-cost/low-overhead. Our checkpointing implementation has proven to be highly effective and resilient to node failures.

Our migration system allows a computation to be restarted with varying topologies, allowing for the elimination of "spare" nodes for checkpoint restarts. Further, we showed that our replication strategy is highly scalable. Where previous work discussed within the literature typically tests scalability up to 16 nodes, we have demonstrated low overhead up to 256 nodes with more realistic checkpoint image sizes of 1 GB per node. Our work enables more effective use of resources without any reliance on network storage.

References

[1] Top500 list `http://www.top500.org`, 2006.

[2] T. Angskun, G. Fagg, G. Bosilca, J. Pjesivac-Grbovic, and J. Dongarra. Scalable fault tolerant protocol for parallel runtime environments. In *Euro PVM/MPI*, 2006.

[3] T. Angskun, G. Fagg, G. Bosilca, J. Pjesivac-Grbovic, and J. Dongarra. Self-healing network for scalable fault tolerant runtime environments. In *Austrialian-Hungarian Workshop on Distributed and Parallel Systems*, 2006.

[4] R. Aulwes, D. Daniel, N. Desai, R. Graham, L. Risinger, M. Taylor, R. Woodall, and M. Sukalski. Architecture of la-mpi, a network-fault-tolerant mpi. In *International Parallel and Distributed Processing Symposium*, 2004.

[5] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. Mpich-v: toward a scalable fault tolerant mpi for volatile nodes. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[6] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated application-level checkpointing of mpi programs. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 84–94. ACM Press, 2003.

[7] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Collective operations in application-level fault-tolerant mpi. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 234–243. ACM Press, 2003.

[8] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[9] R. Butler, W. Gropp, and E. L. Lusk. A scalable process-management environment for parallel programs. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 168–175, London, UK, 2000. Springer-Verlag.

[10] J. Cao, Y. Li, and M. Guo. Process migration for mpi applications based on coordinated checkpoint. In *ICPADS '05: Proceedings of the 11th International Conference on Parallel and Distributed Systems (ICPADS'05)*, pages 306–312, Washington, DC, USA, 2005. IEEE Computer Society.

[11] S. Chakravorty and L. Kalé. A fault tolerant protocol for massively parallel systems. In *Proceedings. 18th International Parallel and Distributed Processing Symposium.*, page 212, 2004.

[12] S. Chakravorty and L. Kalé. A fault tolerance protocol with fast fault recovery. In *Proceedings of* 21^{st} *IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2007*, Long Beach, CA, USA, Mar. 26-30, 2007. To appear.

[13] S. Chakravorty, C. Mendes, and L. Kalé. Proactive fault tolerance in large systems. In *Workshop on High Performance Computing Reliability Issues*, 2005.

[14] C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello. Mpi tools and performance studies—blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant mpi. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 127, New York, NY, USA, 2006. ACM Press.

[15] J. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience*, 15:1–18, 2003.

[16] J. Duell. The design and implementation of berkeley lab's linux checkpoint/restart, 2003. http://old-www.nersc.gov/research/FTG/ checkpoint/reports.html.

[17] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[18] G. E. Fagg and J. Dongarra. Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 346–353, London, UK, 2000. Springer-Verlag.

[19] T. M. Forum. MPI: A message passing interface. *Proceedings of the Supercomputing Conference*, pages 878–883, 1993.

[20] Q. Gao, W. Yu, W. Huang, and D. K. Panda. Application-transparent checkpoint/restart for mpi programs over infiniband. In *ICPP'06: Proceedings of the 35th International Conference on Parallel Processing*, Columbus, OH, October 2006.

[21] C. Huang, O. Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*, pages 306–322, College Station, Texas, October 2003.

[22] C. Huang, G. Zheng, S. Kumar, and L. V. Kalé. Performance evaluation of adaptive MPI. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, March 2006.

[23] Infiniband Trade Association. Infiniband. `http://www.infinibandta.org/home`.

[24] H. Jung, D. Shin, H. Han, J. W. Kim, H. Y. Yeom, and J. Lee. Design and implementation of multiple fault-tolerant mpi over myrinet (m^3). In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 32, Washington, DC, USA, 2005. IEEE Computer Society.

[25] L. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[26] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of unix processes in the condor distributed processing system. Technical Report 1346,University of Wisconsin-Madison, 1997.

[27] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *ICS '02: Proceedings of the 16th international conference on Supercomputing*, pages 84–95, New York, NY, USA, 2002. ACM Press.

[28] Myricom. Myrinet. `http://www.myricom.com/`.

[29] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under unix. Technical Report UT-CS-94-242, 1994.

[30] J. Ruscio, M. Heffner, and S. Varadarajan. Dejavu: Transparent user-level checkpointing, migration, and recovery for distributed systems. In *Proceedings of* 21^{st} *IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2007*, Long Beach, CA, USA, Mar. 26-30, 2007. To appear.

[31] S. Chakravorty and C. Mendes and L. V. Kalé. Proactive fault tolerance in mpi applications via task migration. In *to be presented at HIPC 2006*, 2006.

[32] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. In *Proceedings, LACSI Symposium*, Sante Fe, New Mexico, USA, October 2003.

[33] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. *International Journal of High Performance Computing Applications*, 19(4):479–493, Winter 2005.

[34] J. M. Squyres and A. Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, pages 379–387, Venice, Italy, September / October 2003. Springer-Verlag.

[35] G. Stellner. Cocheck: Checkpointing and process migration for mpi. In *IPPS '96: Proceedings of the 10th International Parallel Processing Symposium*, pages 526–531, Washington, DC, USA, 1996. IEEE Computer Society.

[36] J. Varma, C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Scalable, fault tolerant membership for mpi tasks on hpc systems. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 219–228, New York, NY, USA, 2006. ACM Press.

[37] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. A job pause service under LAM/MPI+BLCR for transparent fault tolerance. In *Proceedings of* 21^{st} *IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2007*, Long Beach, CA, USA, Mar. 26-30, 2007. To appear.

[38] F. C. Wong, R. P. Martin, R. H. Arpaci-Dusseau, and D. E. Culler. Architectural requirements and scalability of the nas parallel benchmarks. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 41, New York, NY, USA, 1999. ACM Press.

[39] N. Woo, H. Y. Yeom, and T. Park. Mpich-gf: Transparent checkpointing and rollback-recovery for grid-enabled mpi processes. *IEICE TRANSACTIONS on Information and Systems*, E87-D(7):1820–1828, 2004.

[40] V. Zandy. Ckpt: user-level checkpointing. `http://www.cs.wisc.edu/~zandy/ckpt/`.

[41] Y. Zhang, D. Wong, and W. Zheng. User-level checkpoint and recovery for lam/mpi. *SIGOPS Oper. Syst. Rev.*, 39(3):72–81, 2005.