A Scalable Asynchronous Replication-Based Strategy for Fault Tolerant MPI Applications*

John Paul Walters and Vipin Chaudhary

Department of Computer Science and Engineering
University at Buffalo, The State University of New York
Buffalo, NY 14260, USA
{waltersj,vipin}@buffalo.edu

Abstract. As computational clusters increase in size, their mean-time-to-failure reduces. Typically checkpointing is used to minimize the loss of computation. Most checkpointing techniques, however, require a central storage for storing checkpoints. This severely limits the scalability of checkpointing. We propose a scalable replication-based MPI checkpointing facility that is based on LAM/MPI. We extend the existing state of fault-tolerant MPI with asynchronous replication, eliminating the need for central or network storage. We evaluate centralized storage, SAN-based solutions, and a commercial parallel file system, and show that they are not scalable, particularly beyond 64 CPUs. We demonstrate the low overhead of our replication scheme with the NAS Parallel Benchmarks and the High Performance LINPACK benchmark with tests up to 256 nodes while demonstrating that checkpointing and replication can be achieved with much lower overhead than that provided by current techniques.

1 Introduction

Computational clusters with hundreds and thousands of processors are fast-becoming ubiquitous in large-scale scientific computing. This is leading to lower mean-time-to-failure and forces the system software to deal with the possibility of arbitrary and unexpected node failure. Since MPI 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 and transparent MPI implementation.

Several groups have included checkpointing within various MPI implementations. MVAPICH2 now includes support for kernel-level checkpointing of Infiniband MPI processes [1]. Sankaran et al. also describe a kernel-level checkpointing strategy within LAM/MPI [2,3]. However, these implementations suffer from a major drawback: a reliance on a common network file system or dedicated checkpoint servers.

We consider the reliance on network file systems, parallel file systems, and/or checkpoint servers to be a fundamental limitation of existing fault-tolerant systems. Storing checkpoints directly to network storage incurs too great an overhead. Using dedicated

* This research was supported in part by NSF IGERT grant 9987598, the Institute for Scientific Computing at Wayne State University, MEDC/Michigan Life Science Corridor, and NYSTAR.

checkpoint servers saturates the network links of a few machines, resulting in degraded performance. Even parallel file systems are quickly saturated. As such, we make the following contributions in this paper:

1. We propose and implement a checkpoint replication system that 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.
2. We show that common existing strategies, including the use of dedicated checkpoint servers, storage area networks (SANs), and parallel file systems, are inadequate for even moderately-sized computations.

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 existing LAM/MPI checkpointing strategies. 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 [4] is a research implementation of the MPI-1.2 standard with portions of the MPI-2 standard. LAM uses a layered software approach in its construction [5]. In doing so, various modules are available to the programmer that tune LAM/MPI's runtime functionality including TCP, Infiniband, Myrinet, and shared memory communication. The most commonly used module, however, 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, 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 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 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.

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 [6].

3 LAM/MPI Checkpointing

We are not the first group to implement checkpointing within the LAM/MPI system. Three others [7,3,8] have added basic checkpoint/restart support. Because of the previous work in LAM/MPI checkpointing, the basic checkpointing/restart building blocks were already present within LAM's source code. This provided an ideal environment for testing our replication strategy. We begin with a brief overview of checkpointing with LAM/MPI.

Sankaran et al. first added checkpointing support within the LAM system [3] 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. 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) [9] is used as a kernel-level checkpointing engine. Each process checkpoints itself using BLCR (including *mpirun*) and the computation resumes.

A problem with the above solution is that it requires identical 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. Cao et al. propose a migration scheme that parses the binary checkpoint images, finds the MPI process location information, and updates the node IDs [10]. Wang, et al. propose a pause/migrate solution where spare nodes are used for migration purposes when a LAM daemon discovers an unresponsive node [8]. Upon detecting a failure, their system migrates the failed processes via a network file system to the replacement nodes before continuing the computation.

We use the same coordinated blocking approach as Sankaran's technique described above. To perform the checkpointing, we use Victor Zandy's *Ckpt* checkpointer [11]. Unlike previous solutions, we allow for arbitrary restart topologies without relying on any shared storage or checkpoint parsing. Instead, we reinitialize the MPI library and update node and process-specific attributes in order to restore a computation on varying topologies. Due to space limitations and our focus on the replication portion of our implementation, we omit the details of the basic checkpoint/migrate/restart solution. A more detailed description is available in our extended work [12].

4 Checkpoint Storage, Resilience, and Performance

In order to enhance the resiliency of checkpointing while simultaneously reducing its overhead, 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 acknowledge that arguments can be made in support of the use of SANs or parallel file systems for the storage of checkpoints. The most powerful supercomputers, such as the IBM Bluegene/L, have no per-node local storage. Instead, parallel file systems are used for persistent data storage in order to reduce the number of disk related node failures. We do not position our implementation for use on such massive supercomputers. Instead, we target clusters consisting of hundreds or thousands of commodity nodes, each with its own local storage.

For our implementation testing we used a Wayne State 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. We evaluate both centralized-server and SAN-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 [13] and the High Performance LINPACK (HPL) [14] benchmark.

To gauge the performance of our checkpointing library using the NPB tests, we used exclusively “Class C” benchmarks. Our HPL benchmark tests used 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. We describe the scalability tests and configuration in Section 4.4.

In order to test the overhead of our implementation we chose to checkpoint the benchmarks with much greater frequency than would otherwise be used. By checkpointing at frequencies as short as 1 minute, we are better able to demonstrate the individual components of the overhead. In a real application, users would likely checkpoint an application at intervals of several hours (or more).

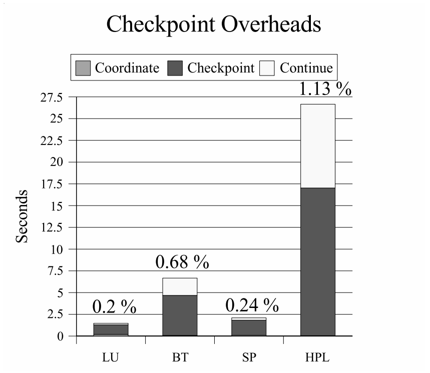
As a baseline, we compare the SAN storage, dedicated server storage, and replication storage techniques against the local disk checkpoint data shown in Figure 1. Here we show the result of periodically checkpointing the NAS Parallel Benchmarks as well as the HPL benchmark along with the time taken to 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 1(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 1(b). The coordination phase includes the time needed to acquiesce the network channels/exchange bookmarks (see Section 3). 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 includes the time needed to synchronize the resumption of computation. The coordination and continue phases require barriers to ensure application synchronization, while each process performs the checkpoint phase independently.

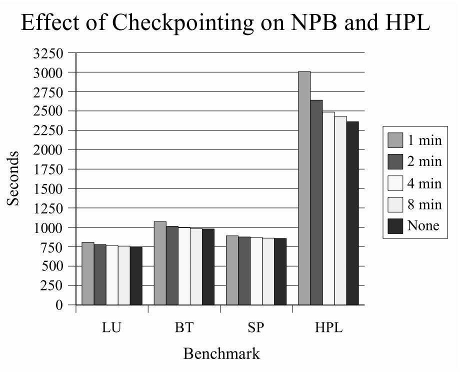
As confirmed in Figure 1(a), the time required to checkpoint the entire system is largely dependent on the time needed to checkpoint the individual nodes. Writing the checkpoint file to disk represents the single largest time in the entire checkpoint process and dwarfs the coordination phase. Thus, as the memory footprint of an application grows, so too does the time needed to checkpoint. This can also impact the time needed to perform the *continue* barrier as faster nodes are forced to wait for slower nodes to write their checkpoints to disk.

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) [15] and storing checkpoints directly to network storage [2,1].



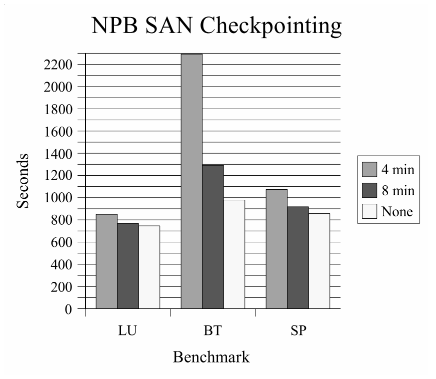
(a) % indicates the contribution of checkpointing (in terms of overhead) at 8 minute intervals over the base timings without checkpointing (from Figure 1(b)).



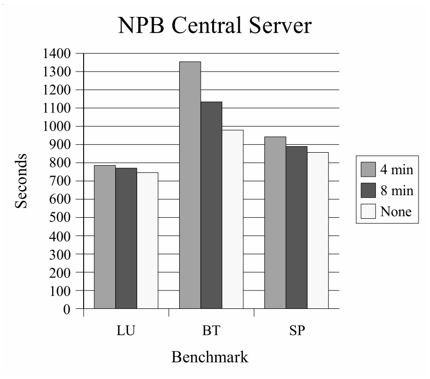
(b) Multiple checkpointing intervals.

Fig. 1. A breakdown of overheads when checkpointing to local disks

We begin our evaluation with a comparison of these two common strategies. To do so, we implemented both a dedicated checkpoint server solution as well as a SAN-based checkpoint storage solution by extending the LAM daemons and *mpirun* to collect and propagate checkpoints.



(a) Checkpointing to SAN-based storage.



(b) Checkpointing to a central server.

Fig. 2. Runtime of NPB with checkpoints streamed to central checkpoint server vs. saving to SAN

In Figure 2 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 2(a) to the checkpoint server data presented in Figure 2(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 minute checkpointing intervals the SP benchmark incurs an overhead of approximately 4% while the overhead of BT is 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 despite suffering 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 [15]. However, their efficacy in the presence of large checkpoint files has not been demonstrated in the literature (NPB class B results are shown).

Wang et al. propose a technique to alleviate the impact of checkpointing directly to SANs [8]. Their technique combines local checkpointing with asynchronous checkpoint propagation to network storage. However, their solution requires 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.

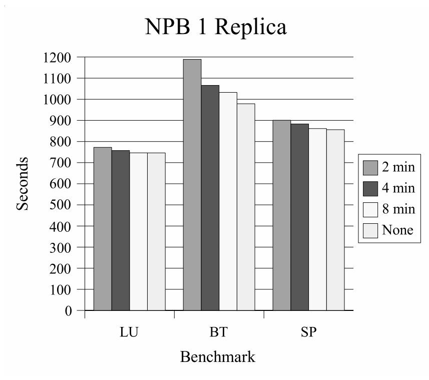
4.2 Checkpoint Replication

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 using a peer-to-peer strategy 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.

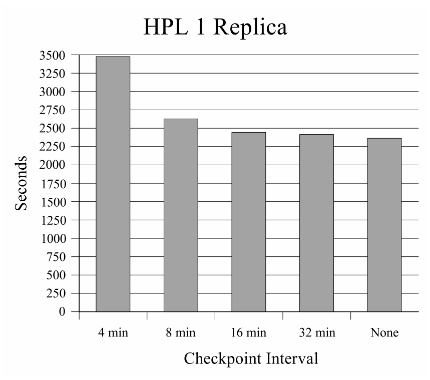
A variety of replication strategies have been used in peer-to-peer systems. Typically, such strategies must take into account the relative popularity of individual files within the network in order to ascertain the optimal replication strategy. Common techniques include the square-root, proportional, and uniform distributions [16]. 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 the availability of each checkpoint 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 distribute the checkpoints randomly 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 checkpoints throughout the network provides the greatest possible survivability.

Figure 3(a) shows the results of distributing a single replica throughout the cluster with NPB. As can be seen, the overhead in Figure 3(a) is substantially lower than that of the centralized server shown in Figure 2(b). In each of the three NAS benchmarks, we



(a) Periodic NPB checkpointing with a single replica.



(b) Periodic HPL checkpointing with a single replica.

Fig. 3. Benchmark timings with one replica

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 3(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%.

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 simulator capable of replicating node failures, given inputs of the network size and the number of replicas.

Table 1. Maximum number of allowed failures with 90, 99, and 99.9% restart probability

	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

From Table 1 we can see that our replication strategy enables a high probability of restart with seemingly few replicas needed in the system. Further, our replication technique scales quite well with the number of CPUs. With 2048 processors, for example, 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.

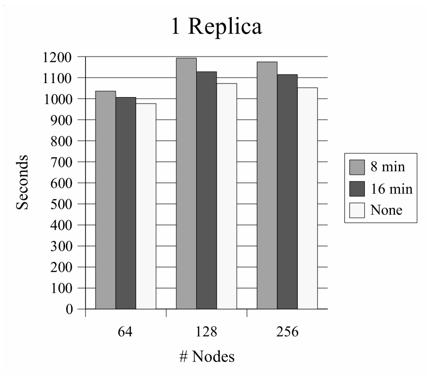
4.4 Scalability Studies

To evaluate for scalability we tested our implementation with up to 256 nodes on a University at Buffalo Center for Computation Research owned cluster consisting of 1600 3.0/3.2 GHz Intel Xeon processors, with 2 processors per node (800 total nodes), a 30 TB EMC SAN as well as a high performance Ibrix parallel file system. The network is connected by both gigabit ethernet and Myrinet. Gigabit ethernet was used for our tests. 21 active Ibrix segment servers are in use and connect to the existing EMC SAN.

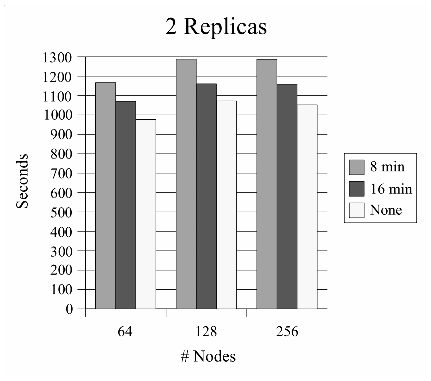
Because our checkpointing engine, *Ckpt* [11], is only 32 bit while the University at Buffalo's Xeon processors are each 64 bit, we simulated the mechanics of checkpointing with an artificial 1 GB file that is created and written to local disk at each checkpoint interval. Aside from this slight modification, the remaining portions of our checkpointing system remain intact (coordination, continue, file writing, and replication).

In Figure 4 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 and Ibrix parallel file system in Figure 4(d). 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 with high node failures.

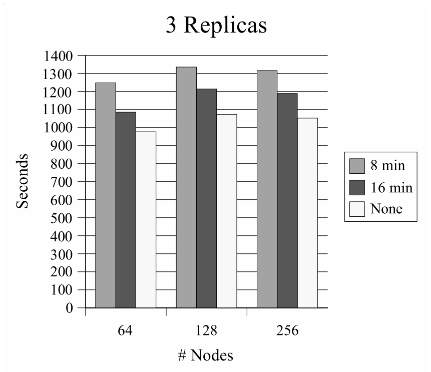
The individual figures in Figure 4 all represent the total run time of the HPL benchmark at each cluster size. Thus, comparing the run times at each replication level against



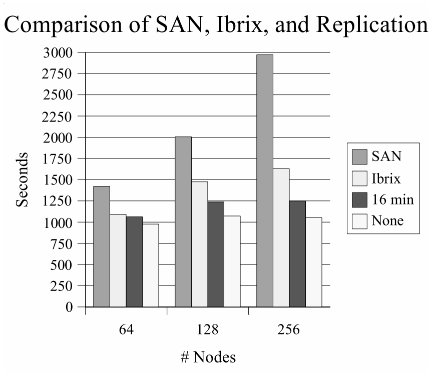
(a) HPL with one replica per checkpoint.



(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 and Ibrix PFS.

Fig. 4. Scalability tests using the HPL benchmark

the base run time without checkpointing provides a measure of the overhead involved at each replication level. From Figure 4(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, and 3 replicas with only a minimal increase in overhead for each replication increase.

The most important results, however, are those shown in Figure 4(d). Here we include the overhead data with 4 replicas (labeled "16 min" in Figure 4(d)) as well as with checkpointing directly to the SAN (a common strategy in nearly all MPI checkpointing literature) and the Ibrix parallel file system. In every case, checkpoints are taken at 16 minute intervals. 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. The Ibrix file system, while scaling much better than the EMC SAN, is quickly overwhelmed as the ratio of compute nodes to segment servers increases. Indeed, the overhead of saving checkpoints to the Ibrix parallel file systems for cluster sizes of 128 and 256 nodes is 37.5% and 55% respectively, while our replication strategy results in overheads of only 15.4% and 18.7%.

5 Related Work

Other MPI implementations aside from LAM/MPI have been extended with checkpointing support. MPICH-GM, a Myrinet specific implementation of MPICH has been extended to support user-level checkpointing [17]. Similarly, Gao et al. [1] demonstrate a kernel-level checkpointing scheme for Infiniband (MVAPICH2) that is based on the BLCR kernel module [9]. DejaVu [18] 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.

Coti, et al. implemented a blocking coordinated protocol within MPICH2 [15]. Their observations suggested that for high speed computational clusters blocking approaches achieve the best performance (compared to non-blocking/message-logging approaches) for sensible checkpoint frequencies. Our scalability results from Section 4.4 lend additional evidence supporting their claim.

Using Charm++ and Adaptive-MPI, Chakravorty et al. add fault tolerance via task migration to the Adaptive-MPI system [19,20]. Zheng, et al. discuss a minimal replication strategy within Charm++ to save each checkpoint to two "buddy" processors [21]. Their work, however, is limited in that it only provides a minimal amount of resiliency and is vulnerable to multiple node failures.

Other strategies such as application-level checkpointing have also been extended to MPI checkpointing, particularly the C^3 [22] 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 periodic checkpointing and require access to a user's source code.

Our work differs from the above in that 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).

6 Conclusions

We have shown that it is possible to effectively checkpoint MPI applications using the LAM/MPI implementation with low overhead. Previous checkpointing implementations have typically neglected the issue of checkpoint replication. We comprehensively addressed this issue with a comparison against all major storage techniques, including commercial SAN strategies and a commercial parallel file system. Our replication implementation has proven to be highly effective and resilient to node failures.

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. We hope to continue this work with a greater interest in applying our replication strategies toward fault-tolerant HPC. By combining our checkpoint/restart and migration system with a fully fault-tolerant MPI, even greater resource utilization would be possible while still maintaining user-transparency.

References

1. Gao, Q., Yu, W., Huang, W., Panda, D.K.: Application-Transparent Checkpoint/Restart for MPI Programs over InfiniBand. In: ICPP 2006. Proceedings of the 35th International Conference on Parallel Processing, Columbus, OH (2006)
2. Sankaran, S., Squyres, J.M., Barrett, B., Lumsdaine, A., Duell, J., Hargrove, P., Roman, E.: The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. In: Proceedings, LACSI Symposium, Sante Fe, New Mexico, USA (2003)
3. Sankaran, S., Squyres, J.M., Barrett, B., Lumsdaine, A., Duell, J., Hargrove, P., Roman, E.: The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. International Journal of High Performance Computing Applications 19(4), 479–493 (2005)
4. Burns, G., Daoud, R., Vaigl, J.: LAM: An Open Cluster Environment for MPI. In: Proceedings of Supercomputing Symposium, pp. 379–386 (1994)
5. Squyres, J.M., Lumsdaine, A.: A Component Architecture for LAM/MPI. In: Dongarra, J.J., Laforenza, D., Orlando, S. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 2840, pp. 379–387. Springer, Heidelberg (2003)
6. Elnozahy, E.N.M., Alvisi, L., Wang, Y.M., Johnson, D.B.: A Survey of Rollback-Recovery Protocols in Message-Passing Systems. ACM Comput. Surv. 34(3), 375–408 (2002)
7. Zhang, Y., Wong, D., Zheng, W.: User-Level Checkpoint and Recovery for LAM/MPI. SIGOPS Oper. Syst. Rev. 39(3), 72–81 (2005)
8. Wang, C., Mueller, F., Engelmann, C., Scott, S.L.: A Job Pause Service under LAM/MPI+BLCR for Transparent Fault Tolerance. In: IPDPS 2007. Proceedings of 21^{st} IEEE International Parallel and Distributed Processing Symposium, Long Beach, CA, USA (2007), Long Beach, CA, USA (2007)
9. Duell, J.: The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart (2003), `http://old-www.nersc.gov/research/FTG/checkpoint/reports.html`
10. Cao, J., Li, Y., Guo, M.: Process Migration for MPI Applications based on Coordinated Checkpoint. In: ICPADS 2005. Proceedings of the 11th International Conference on Parallel and Distributed Systems, pp. 306–312. IEEE Computer Society Press, Los Alamitos (2005)

11. Zandy, V.: Ckpt: User-Level Checkpointing (2005), `http://www.cs.wisc.edu/~zandy/ckpt/`
12. Walters, J., Chaudhary, V.: A Comprehensive User-level Checkpointing Strategy for MPI Applications. Technical Report 2007-1, University at Buffalo, The State University of New York, Buffalo, NY (2007)
13. Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V., Weeratunga, S.: The NAS Parallel Benchmarks. International Journal of High Performance Computing Applications 5(3), 63–73 (1991)
14. Dongarra, J.J., Luszczek, P., Petitet, A.: The LINPACK Benchmark: Past, Present, and Future. Concurrency and Computation: Practice and Experience 15, 1–18 (2003)
15. Coti, C., Herault, T., Lemarinier, P., Pilard, L., Rezmerita, A., Rodriguez, E., Cappello, F.: MPI Tools and Performance Studies—Blocking vs. Non-Blocking Coordinated Checkpointing for Large-Scale Fault Tolerant MPI. In: Löwe, W., Südholt, M. (eds.) SC 2006. LNCS, vol. 4089, Springer, Heidelberg (2006)
16. Lv, Q., Cao, P., Cohen, E., Li, K., Shenker, S.: Search and Replication in Unstructured Peer-to-Peer Networks. In: ICS 2002. Proceedings of the 16th international conference on Supercomputing, pp. 84–95. ACM Press, New York (2002)
17. Jung, H., Shin, D., Han, H., Kim, J.W., Yeom, H.Y., Lee, J.: Design and Implementation of Multiple Fault-Tolerant MPI over Myrinet (M^3). In: Gschwind, T., Aßmann, U., Nierstrasz, O. (eds.) SC 2005. LNCS, vol. 3628, p. 32. Springer, Heidelberg (2005)
18. Ruscio, J., Heffner, M., Varadarajan, S.: DejaVu: Transparent User-Level Checkpointing, Migration, and Recovery for Distributed Systems. In: Proceedings of the 21^{st} IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2007, Long Beach, CA, USA (2007)
19. Chakravorty, S., Mendes, C., Kalé, L.V.: Proactive Fault Tolerance in MPI Applications via Task Migration. In: Robert, Y., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2006. LNCS, vol. 4297, Springer, Heidelberg (2006)
20. Chakravorty, S., Kalé, L.: 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 (2007)
21. Zheng, G., Shi, L., Kalé, L.V.: FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI. In: CLUSTER, pp. 93–103 (2004)
22. Bronevetsky, G., Marques, D., Pingali, K., Stodghill, P.: Collective Operations in Application-Level Fault-Tolerant MPI. In: ICS 2003. Proceedings of the 17th annual international conference on Supercomputing, pp. 234–243. ACM Press, New York (2003)