Replication-Based Fault-Tolerance for MPI Applications

John Paul Walters and Vipin Chaudhary, *Member, IEEE*

Abstract—As computational clusters increase in size, their mean-time-to-failure reduces drastically. Typically, checkpointing is used to minimize the loss of computation. Most checkpointing techniques, however, require central storage for storing checkpoints. This results in a bottleneck and severely limits the scalability of checkpointing, while also proving to be too expensive for dedicated checkpointing networks and storage systems.

We propose a scalable replication-based MPI checkpointing facility. Our reference implementation is based on LAM/MPI, however, it is directly applicable to any MPI implementation. We extend the existing state of fault-tolerant MPI with asynchronous replication, eliminating the need for central or network storage. We evaluate centralized storage, a Sun X4500-based solution, an EMC SAN, and the Ibrix commercial parallel file system and show that they are not scalable, particularly after 64 CPUs. 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 while demonstrating that checkpointing and replication can be achieved with much lower overhead than that provided by current techniques. Finally, we show that the monetary cost of our solution is as low as 25% of that of a typical SAN/parallel file system-equipped storage system.

Index Terms—fault-tolerance, checkpointing, MPI, file systems.

1 Introduction

Computational clusters with hundreds or 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 [1] provides no mechanism to recover from such failures, a single node failure will halt the execution of the entire computation. Thus, there exists great interest in the research community for a truly fault-tolerant MPI implementation.

Several groups have included checkpointing within various MPI implementations. MVAPICH2 now includes support for kernel-level checkpointing of InfiniBand MPI processes [2]. Sankaran et al. also describe a kernel-level checkpointing strategy within LAM/MPI [3], [4].

However, such 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 checkpoint systems. While SAN's and network file systems are certainly useful for general computation, writing checkpoints 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. Even parallel file systems are easily saturated. In this article we focus specifically on reducing checkpointing overhead by eliminating the reliance on shared storage. For completeness we develop a user-level checkpointing solution within the LAM MPI implementation. With our user-level checkpointing solution, initially described in our previous work [5], we investigate several existing checkpoint storage solutions and demonstrate the low overhead of our proposed replication-based approach.

Because most checkpointing 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 [6]) scalability issues regarding checkpointing are rarely considered. We specifically test the scalability of our implementation with class D versions of the NAS Parallel Benchmark suite. Further, we provide performance results for the HPL benchmark with up to 256 nodes and individual node memory footprints of up to 1 GB each.

We show that the overhead of checkpointing to network storage, parallel file systems, or dedicated checkpoint servers is too severe for even moderately sized computational clusters. As such, we make the following contributions in this article:

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. In addition, our solution is significantly less expensive, costing as little as 25% of the cost of a dedicated checkpointing SAN.

• *J.P. Walters and V. Chaudhary are with the Department of Computer Science and Engineering, University at Buffalo, Buffalo, NY, 14260. Email: {waltersj, vipin}@buffalo.edu.*

The remainder of this article is outlined as follows: in Section 2 we provide a brief introduction to LAM/MPI and checkpointing. In Section 3 we describe the work related to this project. In Section 4 we discuss existing LAM/MPI checkpointing strategies. In Section 5 we compare existing checkpoint storage strategies and evaluate our proposed replication technique. Finally, in Section 6 we present our conclusions.

2 Background

2.1 LAM/MPI

LAM/MPI [3] is a research implementation of the MPI-1.2 standard [1] with portions of the MPI-2 standard. LAM uses a layered software approach in its construction [7]. In doing so, various modules are available to the programmer that tune LAM/MPI's runtime functionality including TCP, InfiniBand [8], Myrinet [9], 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.

More recently, checkpoint/restart support has been added to the Open MPI implementation [10]. Its implementation mirrors the kernel-level checkpoint/restart work already present in LAM/MPI. For our work we chose LAM over Open MPI due to LAM's maturity and widespread academic use. However, this work is directly applicable to any MPI implementation as the replication work itself is implementation-neutral.

2.2 Checkpointing Distributed Systems

Checkpointing is commonly performed at one of three levels: the kernel-level, user-level, and application-level (others, including language-level, hardware-level and virtual machine-level also exist). 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 at the library level, 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. 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. Thus, in order for a checkpoint to be taken 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 [11].

Regardless of the level at which a checkpoint is taken (user-level, kernel-level, or application-level) and the mechanism used to achieve a consistent state (message-logging, or coordination) a checkpoint must ultimately be stored such that it can be retrieved in the event of a node failure. Checkpoints may be stored in memory or directly to disk. The most common strategy is to store checkpoints directly to stable storage, such as a SAN or parallel file system. This has advantages in terms of simplicity and ease of implementation, but largely ignores the overhead created by such an approach. Our focus in this article is to reduce the overhead of checkpointing through the use of local disk replication.

3 Related Work

Checkpointing at both the user-level and kernel-level has been extensively studied [12]–[14]. The official LAM/MPI implementation includes support for checkpointing using the BLCR kernel-level checkpointing library [4]. An implementation by Zhang et al. duplicates the functionality of LAM's checkpointing, but implements the checkpointing at the user-level [15]. As described earlier, Open MPI also includes support for LAM-like BLCR checkpointing [10]. All current LAM and Open MPI implementations rely on network storage.

Scheduling-based approaches have been used to help alleviate the impact of saving checkpoints to centralized storage. Wang et al. have implemented pause/restart functionality in LAM [16] that moves checkpoint data to centralized storage in smaller groups. However, their results are limited to 16 nodes, making the scalability of their solution unknown. Similarly MPICH-GM, a Myrinet-specific implementation of MPICH, has been extended by Jung et al. to support user-level checkpointing [17]. They show that the overhead of SAN-based checkpoint storage may be partially mitigated by first storing checkpoints locally before serializing their transfer to a SAN. We repeat and compare against their solution in Section 5.5.

Gao et al. demonstrate a kernel-level checkpointing scheme for InfiniBand on MVAPICH2 that is based on the BLCR kernel module [2]. To help improve the performance

of checkpointing, particularly the checkpointing delay due to shared storage, they propose a group-based checkpointing solution that divides the MPI processes into multiple (smaller) checkpointing groups. Each group checkpoints individually in an effort to reduce checkpointing overhead. In order to achieve low overhead, however, their solution requires that non-checkpointing groups make computational progress during other groups' checkpoints. This requires that the MPI processes be divided into groups according to their communication patterns, which in turn requires information from the user. Further, their tests are limited to 32 nodes, making the scalability of their system unknown.

Using Charm++ [18] and Adaptive-MPI [19], Chakravorty et al. add fault tolerance via task migration to the Adaptive-MPI system [20], [21]. Their system relies on processor virtualization to achieve migration transparency. Zheng et al. discuss a minimal replication strategy within Charm++ to save each checkpoint to two "buddy" processors [22]. Their work, however, is limited in that it provides a minimal amount of resiliency and may only recover from a single node failure.

Coding strategies have recently reappeared as a common technique for providing fault-tolerance to checkpoint data. While RAID-like coding strategies have been previously described for parallel systems [23], [24], the massive amounts of data generated in today's checkpoints is necessitating a reexamination of such techniques. Plank investigated the performance implications of centralized storage with the use of RAID-like checkpointing [23]. His work, however, does not account for the use of SANs or parallel file systems, but instead relies on a traditional NFS server for checkpoint storage. Ren et al. use information dispersal algorithms [25] along with checkpoint storage repositories in a fine-grained cycle sharing system [26]. Chen et al. similarly discuss a coding approach to checkpoint storage in FT-MPI that requires users to insert the coding scheme directly into their self-written checkpointing code [27].

Nath et al. describe a content addressable storage-based (CAS) solution that may be used to reduce the impact of checkpointing on storage systems (either shared or local storage). Their approach reduces or eliminates duplication of data within the CAS file system by storing common data once and referencing it as-needed [28]. Any common data that exists from one checkpoint period to the next can be stored once, thereby reducing disk IO and disk space consumed. However, this approach introduces a single point of failure in that corruption of highly common data chunks may result in unreadable checkpoints spanning multiple checkpoint periods. Nath et al. suggest that a replication system may be used to alleviate this single point of failure [28].

Our work differs from the previous work in checkpoint storage in that we handle checkpoint redundancy for added resiliency in the presence of multiple simultaneous 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 shorter checkpoint intervals (where desired). This work may be used in conjunction with a coding or CAS-based approach, as both may decrease the size of checkpoint fragments while providing fault-tolerance. Combining these strategies with checkpoint replication may yield further improvements over the replication-only results we present in this article. However, the overhead of encoding and decoding large checkpoints is a concern that we are currently investigating.

4 LAM/MPI Checkpointing

We are not the first group to implement checkpointing within the LAM/MPI system. Three others [4], [15], [16] 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 [4] 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 quiescing the network channels, the MPI library is locked and a checkpointing thread assumes control. The BLCR Library (Berkeley Linux Checkpoint/Restart) is used as a kernel-level checkpointing engine [13]. Each process checkpoints itself using BLCR (including *mpirun*) and the computation resumes.

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 that is also implemented within LAM [15]. Their checkpointer (*libcsm*) is signal-based rather than thread-based. But otherwise, their implementation is identical to that of Sankaran's.

A problem with the above solutions is that both require identical restart topologies. If, for example, a compute node fails, the system cannot restart by remapping checkpoints to existing nodes. Instead, a new node must be inserted into the cluster to force the restart topology into consistency with the original 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. [29] propose a migration scheme based on the BLCR work [4] by Sankaran et al. Their technique towards migrating LAM

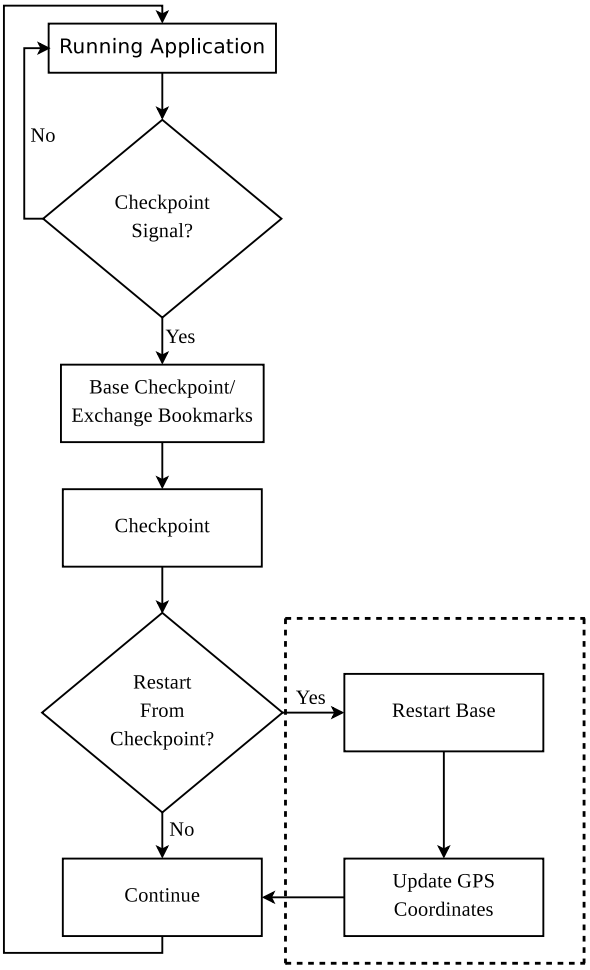

Fig. 1. Checkpointing and restarting a LAM/MPI process. The boxed portion represents the majority of our checkpoint/migration enhancements.

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 [16]. Upon detecting a failure, their system migrates the failed processes via a network file system to the replacement nodes before continuing the computation. However, the scalability of their solution is not yet known as their results were limited to 16 nodes

In Fig. 1 we provide a brief sketch of the LAM/MPI checkpoint/restart mechanism, including our enhancements. Our checkpointing solution uses the same coordinated blocking approach as Sankaran and Zhang's techniques described above. To perform the checkpointing, we use Victor Zandy's *Ckpt* checkpointer [30]. Unlike previous solutions, we allow for arbitrary restart topologies without relying on any shared storage or checkpoint parsing.

The majority of Fig. 1 is the same basic checkpoint strategy that is used by Sankaran et al. [4]. Where our implementation differs is in the "restart base" and "GPS coordinate update" portions of Fig. 1 where process-specific data is updated before continuation (noted by the box). The primary differences (compared to LAM's BLCR checkpoint implementation) are that user-level checkpointing does not provide the same restart capabilities of kernel-level checkpointing. For example, process IDs are restored when using a kernel-level checkpointer such as BLCR [13], but cannot be restored at the user-level. Thus our user-level checkpointing solution requires a more in-depth restart procedure that both restores the computation and resynchronizes the process-specific attributes that would otherwise be handled by the kernel-level checkpointing library.

```
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 */
};
```

Fig. 2. LAM GPS (global positioning system) structure. This maintains node-specific data that must be updated prior to migration.

In the "restart base" portion of Fig. 1 we update all process-specific information that has changed due to a restart. This includes updating process IDs, environment variables, and restarting messaging channels. In order to support process migration, we added a phase to LAM's restart sequence. We depict this as "update GPS coordinates." Internally, LAM uses a GPS (global positioning system) structure (Fig. 2) to maintain vital identification for each MPI process. The GPS structure defines location-specific information for each MPI process. It provides a mechanism to easily determine the node on which a process is executing as well a process' rank within its node. Thus, for every process in the MPI world, there exists a GPS entry, and each process caches 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 Fig. 2). Our strategy is to mimic an *MPI_Init* upon restart in order to trigger a GPS cache update from *mpirun*. This facilitates the arbitrary migration of MPI processes without relying on a tool to manually parse the checkpoints. It further simplifies the restarting of messaging channels as it allows us to use LAM's existing message channel initialization code to start/restart its underlying network connections.

5 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 excessive overhead when compared to checkpointing to local disks.

Moreover, a cluster that utilizes a network file system-based checkpoint/restart mechanism may sit idle should the file system experience a failure. 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. To help mitigate this overhead we examine two replication strategies in the sections to follow: a dedicated server technique, and a distributed replication strategy.

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 node failures due to disk failures. However, the majority of today's computational clusters already posses some form of local storage. Our goal is to better utilize such storage for fault-tolerance. Further, as supercomputing becomes more data-intensive, the need for local storage, even in the largest supercomputers, is becoming even more necessary [31]. With active work in the search and analysis of petabytes of data, high-speed, local, and "intelligent" disks are becoming mandatory [32]. We would argue that the next generation of supercomputers must contain some form of high speed local storage, despite its potential shortcomings.

Our implementation is built on top of TCP using ethernet. Other networking fabrics could be used, as evidenced by the previous work in InfiniBand and Myrinet. However, our goal was to target the most widely used cluster interconnect in an effort to be more widely applicable. Accordingly, we chose gigabit ethernet due to its popularity within the Top500 supercomputer list (56.8% as of June 2008) [33].

We evaluate both centralized-server and network storage-based techniques and compare them against our proposed replication strategy using the SP, LU, and BT benchmarks from the NAS Parallel Benchmarks (NPB) suite [6] and the High Performance LINPACK (HPL) [34] benchmark. The remaining benchmarks within the NPB suite represent synthetic computational kernels, rather than mini-applications. We limited our NPB benchmarking focus to the mini-applications as we found them to be more representative of typical computational workloads. The tests were performed with a single MPI process per node. To measure the performance of our checkpointing implementation and replication using the NPB tests, we used exclusively "Class D" benchmarks. For the HPL benchmark, we selected a matrix with a problem size of 53,000. These configurations resulted in checkpoint sizes and runtimes that are listed in Table 1, with all tests conducted on a single set of 64 nodes. Later, in Section 5.5, we test our solution for scalability.

TABLE 1
Benchmark runtime and checkpoint sizes.

	Runtime (s)	Size (MB)
LU	2240.68	204
BT	3982.69	1020
SP	3962.61	447
HPL	2698.74	374

These tests were performed using a Wayne State University cluster consisting of 92 dual 2.66 GHz Pentium IV Xeon processors with 2.5 GB RAM, a 10,000 RPM Ultra SCSI hard disk and fast ethernet. A 36 TB Sun X4500 storage unit was also used for the network storage tests. The 92 node cluster accesses the X4500 via a single gigabit fibre connection. Tests are performed on 64 nodes of the 92 node cluster, with checkpoints taken at intervals ranging from 4 minutes to 16 minutes. We utilize short checkpointing frequencies in order to exaggerate the impact of our checkpoint strategy on the benchmarks. In a production environment, checkpoints would be taken at intervals of several hours or more. By exaggerating the checkpoint intervals, we are able to gain a more accurate sense of the worst-case overheads present in our system.

As a baseline, we compare storage to a dedicated server, storage to the X4500, and replication storage techniques against the checkpoint data shown in Fig. 3. Here we show the result of periodically checkpointing the NAS Parallel Benchmarks as well as the HPL benchmark along with a breakdown of the time taken to perform a single checkpoint. Each node stores its checkpoints directly to its own local disk. This is done in order to gain a sense of the overhead incurred from checkpointing without stable storage of any kind.

In Fig. 3(a) we break the checkpointing overhead down by coordination time, memory write time, and continue time. The coordination phase includes the time needed to quiesce the network channels/exchange bookmarks (see Section 4). The memory write time consists of the time needed to checkpoint the entire memory footprint of a single process and write it to a local disk. Finally, the continue phase includes the time needed to synchronize and resume the computation. On occasion, particularly with large memory footprints, the continue phase can seem disproportionately long. This is due to some nodes' slower checkpoint/file writing performance, forcing the faster nodes to wait. As we show in Fig. 3(a), this is the case with the BT benchmark. Such waiting ultimately drives up the continue phase timings, as they are simply an average of all nodes' waiting time. We note carefully that one should not conclude that the continue phase dominates the overhead in such cases. Instead, it is more accurately interpreted as a measure of the slowest nodes' checkpoint writing performance.

Thus, the time required to checkpoint the entire system is largely dependent on the time needed to write the memory footprint of the individual nodes. Writing the checkpoint file to disk represents the single largest time in the entire

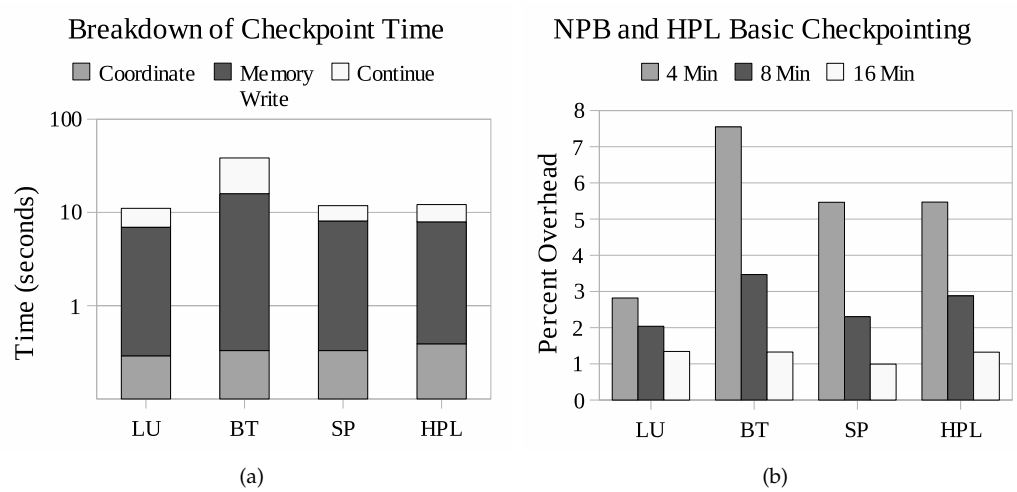

Fig. 3. A breakdown of checkpointing overheads of 64 node computations written directly to local disk: (a) Overhead of checkpointing at 8 minute intervals; (b) total overhead with varying checkpointing frequencies.

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.

In Fig. 3(b) we provide the total overheads for all benchmarks at 4, 8, and 16 minute checkpointing intervals. Our user-level checkpointing implementation shows very little overhead even when checkpointing the 1.02 GB BT benchmark at 4 minute intervals. Indeed none of the benchmarks exhibit overhead of more than 8% despite being checkpointed as much as 17 times (in the case of BT). As the checkpointing frequency is reduced to 8 and 16 minutes, the overhead drops to less than 2% in all cases. 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.

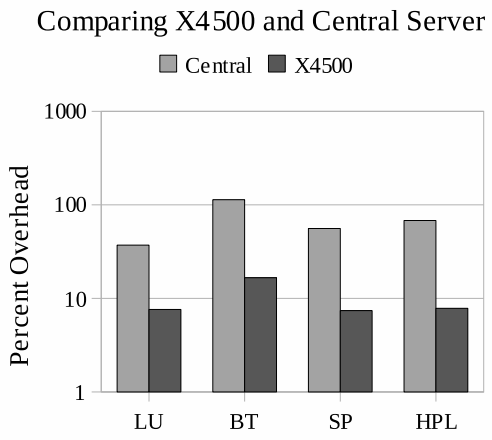

Fig. 4. Overhead of NPB and HPL using X4500 and centralized server, 64 nodes.

5.1 Dedicated Checkpoint Servers versus Checkpointing to Network Storage

The two most common checkpoint storage techniques presented in the literature are the use of dedicated server(s) [35] and storing checkpoints directly to network storage [2], [4]. We begin our evaluation with a comparison of these two 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.

In Fig. 4 we show the overhead of checkpointing with the added cost of streaming the checkpoints to a centralized server or storing the checkpoints to the Sun X4500. Only one checkpoint is taken for both the dedicated checkpoint server as well as the X4500. As Fig. 4 shows, the overhead of storing checkpoints to the X4500 far outperforms that of streaming the checkpoints to a centralized server. Indeed, the overhead of streaming the largest checkpoints, BT, to a dedicated server result in overhead of 113.5%. Storing checkpoints directly to the X4500, however, results in an overhead of only 16.62%. This is to be expected, given that the single dedicated server is collecting checkpoints over a fast ethernet connection.

However, we can also see that as the size of the checkpoint decreases, so too does the overhead incurred by streaming all checkpoints to a centralized server. A single checkpoint incurs an overhead of 37%, 56%, and 68% for the LU, SP, and HPL benchmarks respectively. Similar reductions are seen when saving checkpoints directly to the X4500, resulting in overheads of 7.6%, 7.4%, and 7.8% for LU, SP, and HPL. We note that the overhead incurred by HPL is higher than both SP and LU due to its shorter runtime, and larger checkpoints. With a runtime

that approximates LU, and checkpoint sizes approximating SP, the HPL benchmark spends proportionally more time checkpointing than either LU or SP, hence the slightly higher overhead.

Nevertheless, the X4500-based solution shows a distinct advantage over centralized storage, as would be expected for moderately-sized clusters. To help mitigate the bottleneck of a centralized server, techniques using multiple checkpoint servers have been proposed [35]. However, their efficacy in the presence of large checkpoint files has not been demonstrated in the literature (NPB class B results are shown). Furthermore, if each checkpoint server is collecting data from a distinct set of nodes, as Coti et al. imply [35], the computation will not be resilient to failed checkpoint servers. Finally, as we show in Fig. 4, dedicated server strategies simply do not scale, particularly not at cluster sizes of 64 nodes. Thus, a solution that uses dedicated servers would require a large number of servers to accomodate large computations.

Wang et al. propose a technique to alleviate the impact of checkpointing directly to centralized storage such as SAN or NAS devices by combining local checkpointing with asynchronous checkpoint propagation to network storage [16]. 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. Other strategies, such as serializing checkpoint writes to the SAN, have also been demonstrated in the literature [17]. In Section 5.5 we evaluate the Jung et al. implementation of serializing checkpoints.

5.2 Checkpoint Replication

To address the scalability issues shown in Section 5.1, we implemented an asynchronous replication strategy that distributes the checkpoint/replication overhead over all nodes participating in the computation. 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 [36]. While the uniform distribution is not often 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 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 pseudo-randomly in order to provide a higher resilience to network failures. For example, a solution that replicates to a node's nearest (or farthest) neighbors may fail in the presence of a switch failure. A strategy that forces nodes to replicate to their farthest neighbors may also exhibit lower performance in the presence of inadequate network bisection bandwidth. Further, nodes may not fail independently and instead may cause the failure of additional nodes within their local vicinity. Thus, we use random replication in order to probabilistically spread checkpoints onto multiple switches in multiple locations.

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.

5.2.1 Random Node Selection

Intuitively, the generation of random locations for replica placement is quite simple. The MPI computation is described by a set of nodes, with one or more compute processes executing per node. For each node there is a single *lamd* process per user, independent of the number of MPI processes executing on any particular node. Since replication is provided at the *node* level, not the MPI process level, the computation can be described strictly in terms of the number of nodes. Then $N = node_0, node_1, \ldots, node_{n-1}$ is the set of nodes participating in a user's MPI computation, and r is the number of replicas, decided by the user.

The goal is to randomly generate r replicas per node, subject to the following constraints:

1. A node should not replicate to itself.
2. A node should replicate to exactly r nodes, each of which may only store r replicas.
3. Each of a node's r replicas should be unique.

Constraints 1 and 3 are quite straightforward. Number 2, however, represents the subtle difference between our balanced replication approach, and an approach that simply utilizes multiple checkpoint repositories. Without constraint 2 an imbalance can occur within the replica storage. An extreme, though unlikely, consequence of this would be one in which many nodes are assigned the same replicas, thereby overly burdening a few nodes that act as defacto checkpoint servers. Our approach eliminates this possibility.

Perhaps the most obvious solution is to proceed in order from $node_0$ through $node_{n-1}$, and randomly generate replicas per the above constraints. In practice, however, this will not work due to the gradual narrowing of the replica candidates as the algorithm proceeds. Eventually one is left with only a few replica candidates, none of which satisfy the constraints.

In Algorithm 1 we describe our solution to the problem of random node selection. Similar work has been studied in the context of expander graphs [37]. The key to our algorithm is to begin with an initial state that satisfies the above replica constraints, and incrementally refines the replica choices through swapping. We achieve an initial

```
0 goes to 0 0 0
1 goes to 1 1 1
2 goes to 2 2 2
...
```

Fig. 5. Initial assignment where each node is first initialized with its own replicas.

```
0 goes to 1 2 3
1 goes to 2 3 4
2 goes to 3 4 5
...
```

Fig. 6. After shifts where each replica has been circular shifted by $column\ index - 1$.

```
0 goes to 31 14 5
1 goes to 11 3 61
2 goes to 53 33 36
...
```

Fig. 7. Final assignments where replicas have been randomly swapped within columns, assumes 64 nodes.

Algorithm 1 Compute random replica placements.

Input: Integer r, the number of replicas
Input: Integer n, the number of nodes
Output: Replica array, $Replicas[0...n-1][0...r-1]$

1: **for all** i such that $0 \leq i < n$ **do**
2: Preload node i's replicas with i
3: **end for**
4: Circular-shift each column (j index) of $Replicas$ by $column\_index - 1$
5: **for all** i such that $0 \leq i < n$ **do**
6: **for all** j such that $0 \leq j < r$ **do**
7: **repeat**
8: $z =$ random node, s.t. $0 \leq z < n$
9: $v = Replicas[z][j]$
10: **until** $z \neq i$
11: **if** $v \neq i$ and $Replicas[i][j] \neq z$ **then**
12: $valid\_replica = 1$
13: **for all** k such that $0 \leq k < r$ **do**
14: **if** $Replicas[i][k] == v$ or $Replicas[i][j] == Replicas[z][k]$ **then**
15: $valid\_replica = 0$
16: **end if**
17: **end for**
18: **if** $valid\_replica$ **then**
19: $Replicas[z][j] = Replicas[i][j]$
20: $Replicas[i][j] = v$
21: **end if**
22: **end if**
23: **end for**
24: **end for**

state in lines 1–4. The only requirement of the circular-shift referenced on line 4 is that the final state of replicas should be valid according to the constraints. This means that each column (j index in Algorithm 1) should be circular-shifted such that no two rows (i index) contains duplicate replicas and no row should be assigned itself (if there are n nodes, one should not shift by either 0 or n). The shift can be random or could be as simple as rotating the jth column by $j \pm 1$.

In Fig. 5 and 6 we provide an example of an initial state and post-shift state, assuming 64 nodes with 3 replicas. Clearly, rotating the columns as shown in Fig. 6 is far from ideal. Nevertheless, it is a valid assignment that provides an adequate initial state. Once the initial state is achieved, the algorithm proceeds through each node and each replica and swaps replicas with randomly chosen nodes.

Because we began the swapping with a valid initial state, we need only maintain the constraints while introducing an element of randomness into the replica generation. Two nodes may swap replicas as long as neither node already contains a copy of their potential replicas, and provided that the swap would not result in either node replicating to itself.

In lines 7–10 we generate the candidate node that may later be used for swapping. The only constraint that is enforced is that z, the random node, should not be i. That is, node i should not attempt to swap with itself. At the end of the repeat-until loop, z will contain the node that node i may swap with, and v contains the actual replica that may be swapped.

Once a valid replica candidate has been generated, we simply check to ensure that swapping would maintain a valid state. In line 11 we confirm that the exchange would not result in self-replication. That is, v, the actual replica to be swapped, should not be i, and vice versa. In lines 13–16 we confirm that for both nodes i and the candidate node, the swap would not result in a duplicate replica being added to either node's replica set. Finally, in lines 18–21 we perform the actual replica swap. We provide an example of a possible final replica assignment for a 64 node cluster with 3 replicas in Fig. 7.

5.2.2 Basic Replication Overhead

Fig. 8 shows the results of distributing one and two replicas of both NPB and HPL throughout the cluster. As can be seen, the overhead of a single checkpoint in Fig. 8(a) is substantially lower than that of either the centralized server or X4500, shown previously in Fig. 4. In each case, we are able to reduce the overhead of a single checkpoint to below 2%, where the overhead of a single checkpoint is computed as $\frac{overhead}{num\_checkpoints}$ for the 8 minute checkpoint interval. We observe that even at 8 minute checkpointing intervals the aggregate overhead typically remains below 10%, with only the BT benchmark exhibiting a higher overhead of 12.6%. After doubling the checkpointing interval to 16 minutes, the overhead reduces to 2.9%–4.3% for all benchmarks.

To address the resiliency of checkpoint replication in the presence of node failure we insert multiple checkpoint replicas into the system. In Fig. 8(b) we compare the overheads of distributing 2 replicas of each checkpoint. As we would expect, the overhead incurred is proportional to the size of the checkpoint that is distributed. For smaller

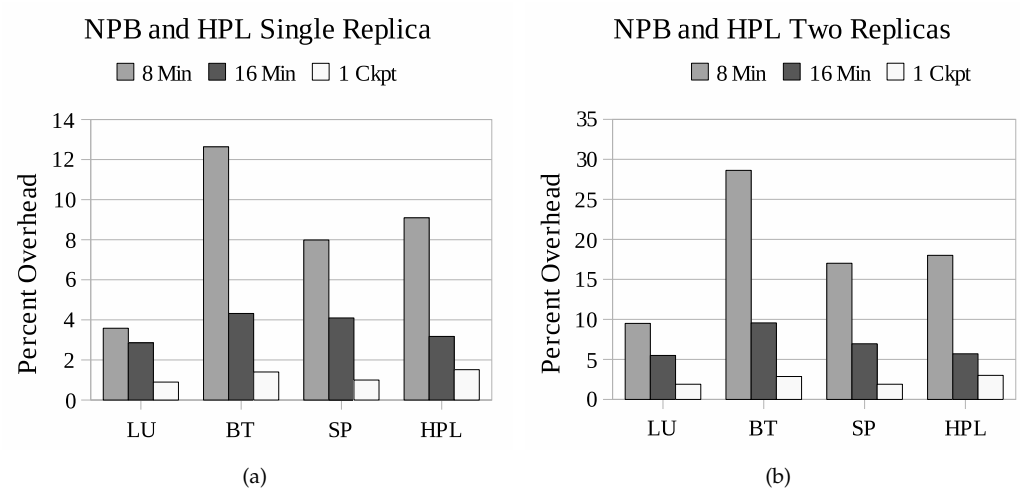

Fig. 8. Overhead of one and two replicas for 64 nodes. Both NPB and HPL results are shown.

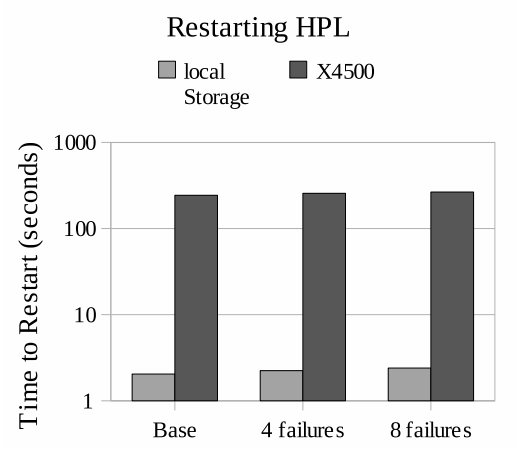

Fig. 9. HPL Recovering from up to 8 failures. The initial computation utilized 64 nodes. Up to 8 nodes have been failed in these tests.

checkpoints such as LU, distributing the 2 replicas represents minimal overhead. As the size of the checkpoint increases, however, so too does the overhead. The BT data of Fig. 8(b) clearly exhibits much higher overall overhead than the single replica data of Fig. 8(a). Even so, the overhead per checkpoint remains quite low as evidenced by BT's single checkpoint overhead of 2.86%. Once the checkpointing interval increases to 16 minutes, the overall overheads drop to less than 10%.

5.3 Restarting/Recovering from Failure

One potential caveat to the use of distributed local storage/replication for checkpointing lies in the restart of failed computations. Storing checkpoints to a SAN or other central repository makes restarting and rebalancing computation quite straightforward. Indeed, one can simply restart the failed computation in a similar manner as running a standard MPI application. Since all checkpoints are stored in a single place, *mpirun* can easily find and start any checkpoint on any node, provided that the checkpoints are already migrateable.

Restarting a computation that uses local storage, however, is slightly more involved. We use the fact that all *lamds* are allocated their replicas at the *lamboot* stage. Therefore, all checkpoints and their replica locations are known a priori. At startup, the *lamboot* command simply records all replica locations and writes them to a file (replica schema). One would expect even the largest cluster's replica schema to total less than several megabytes, with typical clusters averaging less than 100 kilobytes. As such, this file can be replicated like any other checkpoint. For added resiliency, the replica schema may be broadcast to all nodes in the computation.

When restarting a failed computation due to node failure, *mpirun* uses the replica placement data along with the current machine file to remap computation onto existing nodes. First, all surviving nodes are assigned their own checkpoints to restart. If the computation is restarted on a subset of the original nodes, then any outstanding checkpoints are remapped to a previous replica location. If additional replacement nodes are available, *mpirun* initializes a parallel file transfer to copy unclaimed checkpoints to the new node(s).

In Fig. 9 we present the results of restarting the HPL benchmark in the presence of 0, 4, and 8 failures, where 0 failures indicates a simple restart with no missing nodes. We compare our mapped restart to the typical strategy of restarting from a shared storage device. As Fig. 9 shows, our local storage strategy exhibits clear benefits to restarting computations, reducing the X4500-based restart times from 4.4 minutes to 2.4 seconds with 8 failures.

Both solutions exhibit nearly constant restart times regardless of the number of failures. This is to be expected, as oversubscribing computations to nodes incurs overhead primarily in runtime rather than startup time, whereas in restarts the overhead is due to lack of bandwidth. Nevertheless, we see a slight increase in startup times, with the X4500 restart times increasing from 243.4 seconds to 265.7

TABLE 2
Number of allowed failures to maintain 90,99, and 99.9% probability of restart with 1-4 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	5	19	12	8
128	4	1	1	12	6	3	22	13	8	32	21	14
256	5	2	1	20	9	5	37	21	13	55	35	23
512	7	2	1	31	14	7	62	35	21	95	60	38
1024	10	3	1	48	23	11	104	59	33	165	103	67
2048	15	5	2	76	35	17	174	97	56	286	179	112

seconds, and the local storage restart times increasing from 2.0 seconds to 2.4 seconds as additional failures are added.

5.4 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 2 lists the number of failures that our replication scheme will probabilistically tolerate, given 1-4 replicas. We provide failure data for both the number of replicas as well as the desired probability of a successful restart up to 99.9%. Our goal with Table 2 is not to demonstrate that our system is capable of surviving a node failure with 2 or more replicas, but rather to show the number of *simultaneous* node failures that our replication system is capable of surviving, for example in the case of a switch failure.

From Table 2 we can see that our replication strategy enables a high probability of restart with seemingly few replicas needed in the system. With 2048 processors, for example, we estimate that 112 *simultaneous* failures could occur while maintaining at least a 99.9% probability of successful restart and requiring only 4 replicas of each checkpoint. This amounts to 5.5% of the total cluster. With 5 or 6 replicas (not shown) our system can tolerate failures of 9.1% and 12.6% of the cluster while still maintaining a 99.9% probability of restart.

5.5 Scalability Studies

To demonstrate scalability we also 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), gigabit ethernet, Myrinet 2G, a 30 TB EMC Clariion CX700-based SAN as well as a high performance commercial Ibrix parallel file system. 21 active Ibrix segment servers are in use and connect to the existing EMC SAN. The ethernet network has a 44 gigabit theoretical bisection bandwidth. The data mover associated with the SAN's scratch file system used in our benchmarks has a maximum sustained bandwidth of 260 MB/s. Despite the cluster's Myrinet network we chose to implement our results using gigabit ethernet, due to its prevalence in the Top500 list [33]. Similar strategies could be employed for both Myrinet and InfiniBand, and we would expect performance improvements in those cases.

Because the checkpointing engine, *Ckpt* [30], 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, the remaining portions of our checkpointing system remain intact (coordination, continue, file writing, and replication). Tests on all shared storage devices (EMC SAN and Ibrix parallel file system) were performed multiple times to ensure that spikes in resource usage could be taken into account while also being performed at non-peak times to minimize the impact of cluster users on the shared storage. We focus our remaining tests on the HPL benchmark as its runtime is easily adjusted to maintain a reasonable amount of computation. The NPB benchmarks simply execute too quickly on clusters of 128 and 256 nodes and do not demonstrate realistic overheads.

In Fig. 10 we demonstrate the impact of our replication scheme. Each set of nodes (64, 128, and 256) operates on a unique data set to maintain a run time of approximately 1000 seconds. The individual figures in Fig. 10 all represent the percent overhead of the HPL benchmark at each cluster size for checkpointing intervals of 8 and 16 minutes (2 checkpoints and 1 checkpoint, respectively). From Fig. 10(a) we can see that the cost of a single replica is quite low, exhibiting overheads of less than 5% at 16 minute checkpointing intervals. By halving the checkpointing interval to 8 minutes, we incur approximately twice the overhead at 9.4% for 256 nodes.

Similar single checkpoint results can be seen at 2, 3, and 4 replicas with only a minimal increase in overhead for each

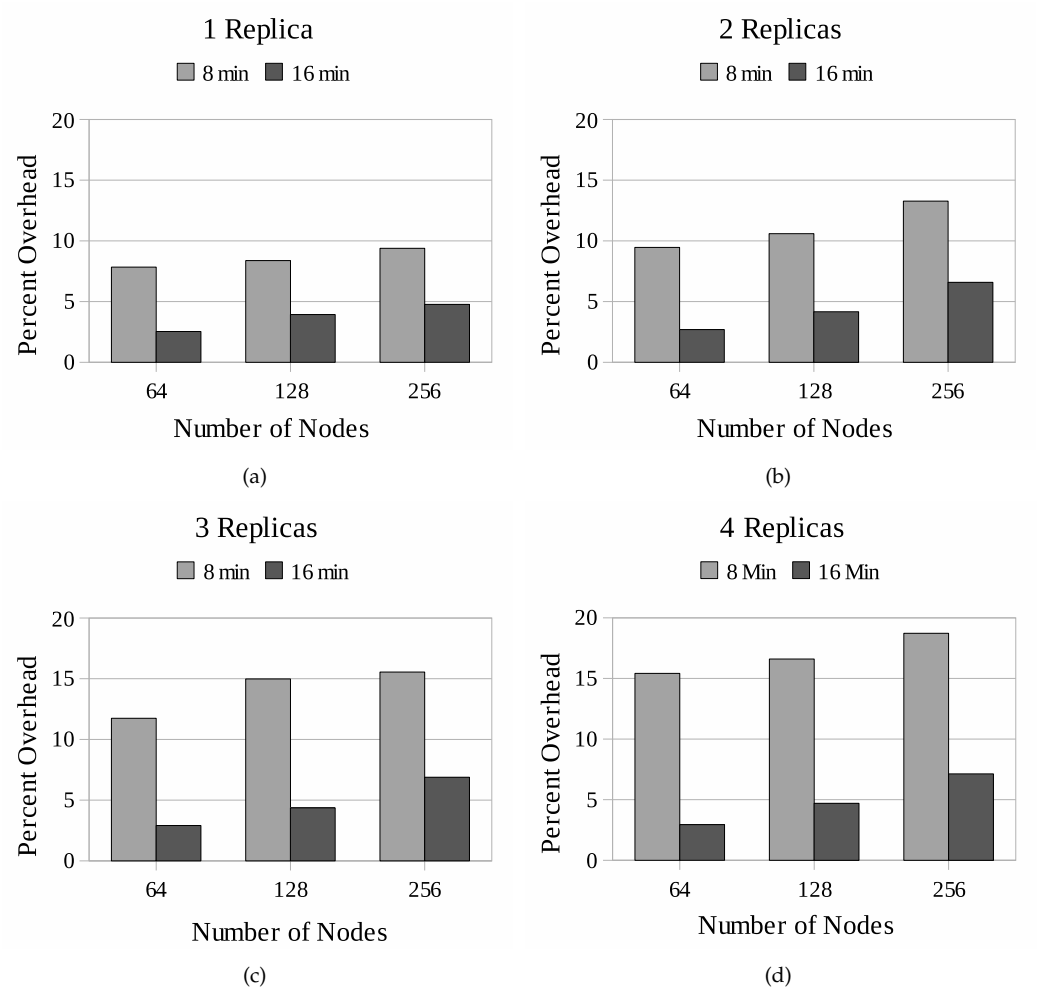

Fig. 10. Scalability tests using the HPL benchmark with 1-4 replicas. Each node size (64, 128, 256) is compared against a base runtime of approximately 1000 seconds.

replica. However, by halving the checkpointing interval to 8 minutes, we incur a slight increase in overhead as replicas are added, with 256 nodes resulting in an overhead of approximately 2.5 times the cost of a single checkpoint. This is due largely to the networking overhead added by replication, resulting in decreased bandwidth for the running application.

For comparison, we also present the overhead of checkpointing to the EMC SAN and Ibrix parallel file system in Fig. 11. Large scale evaluation of the centralized server technique was not possible due to the limited disk capacity of the compute nodes. 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. At 256 nodes, for example, a single checkpoint to the EMC SAN results in an overhead of nearly 200%, while our checkpoint/replication strategy with 4 replicas results in an overhead of only 7.1%. This suggests that for today's large-scale clusters SAN-based checkpointing is not currently a viable solution.

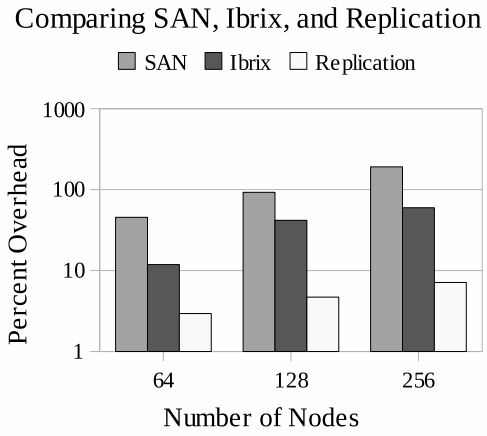

Fig. 11. Comparing overheads of replication to SAN and the Ibrix PFS. A single checkpoint is performed for the SAN and Ibrix cases. Results are compared against a single checkpoint using 4 replicas.

The Ibrix file system, while scaling much better than the EMC SAN, is quickly overwhelmed as the ratio of compute nodes to IO/segment servers increases. At 64 nodes, the

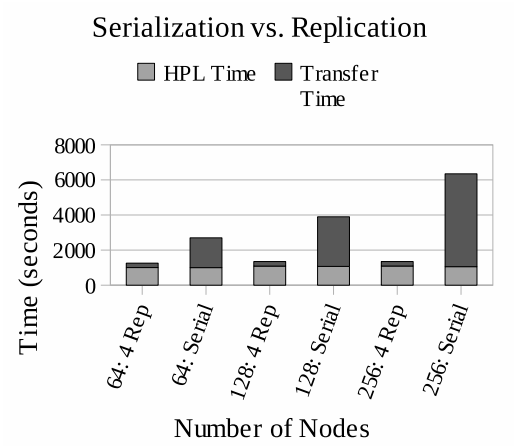

Fig. 12. Comparing replication to serialized SAN writes. The labels "x: 4 Rep" indicates timings for x nodes with 4 replicas. The labels "x: Serial" indcate serialized writes to the SAN. All timings include a single checkpoint.

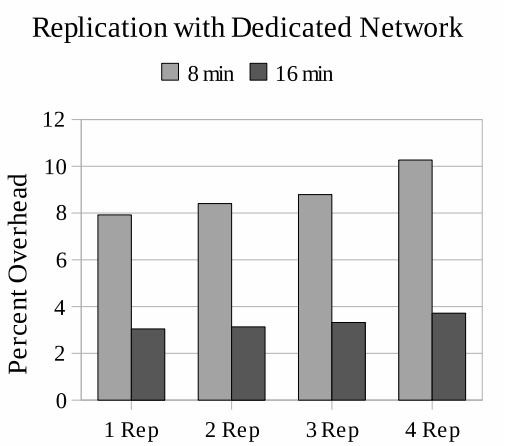

Fig. 13. Replication overhead with dedicated network for checkpointing using 256 nodes.

21 segment servers are able to easily accommodate the checkpoint data with minimal overhead (11.9% for Ibrix vs. 2.3% with 4 replicas at 16 minute intervals). However, as the ratio of compute nodes to segment servers increases to 6:1 and 12:1 for 128 and 256 nodes, the overhead increases dramatically. Indeed, the overhead of saving checkpoints to the Ibrix parallel file systems for cluster sizes of 128 and 256 nodes is 41.8% and 59.8%, respectively.

Jung et al. suggest a serialized approach to reducing the overhead of storing checkpoints to shared devices such as a SAN [17]. Their strategy is to first checkpoint to local storage, then copy the checkpoints to a SAN. We reproduced this experiment, comparing a single serialized checkpoint against a single 4 replica checkpoint. The results, shown in Fig. 12, show that serializing the transfer of checkpoints to shared storage closely matches our HPL runtimes. However, one must also consider the time needed to complete a checkpoint, shown in the stacked bar of Fig. 12. Here we see that the consequence of the serialization is drastically increased checkpointing times, ranging from 1700 seconds at 64 nodes to nearly 5300 seconds at 256 nodes. This represents a checkpoint time of 5 times the benchmark time. Our replication strategy, however, exhibits nearly constant checkpoint times of approximately 250 seconds for 4 replicas, which is well within the benchmark times.

The result of the increased checkpointing times is that applications are effectively limited to a checkpointing interval proportional to the number of nodes participating in the checkpoint. From Fig. 12 we can see that a 256 node computation, each writing 1 GB checkpoints, may checkpoint at a frequency of approximately 90 minutes. Increasing the size of checkpoints or the number of nodes will increase the checkpointing time commensurately. Of course, the checkpointing time may be reduced somewhat by increasing the copy rate to network storage; however, the consequence is increased overhead on the application.

In order to further reduce the replication overhead while maintaining low checkpoint writing/replication times, we include the ability for an MPI application to utilize additional networks to carry checkpoint data. While not mandatory, separating the replica communication from the application's communication may substantially reduce the impact of replication on the application. Clusters that already utilize multiple networks may simply exchange checkpoint replicas using a secondary network to further reduce overheads for no additional cost. We chose to use our Myrinet network, with IP over Myrinet (IPoM), for replication. Checkpoints could instead be carried by the ethernet network with the main application operating over Myrinet; however, we chose to replicate over the Myrinet network to facilitate comparison with our existing single network ethernet tests. A similar strategy could be used to enable checkpoints over an InfiniBand network.

In Fig. 13 we show the results of 1-4 replicas at 256 nodes. As the figure shows, the use of additional networks to offload checkpoint replication can lead to substantial reductions in overhead. Our results show an average improvement of 40% at 8 minute checkpointing intervals and 50% at 16 minute checkpointing intervals when comparing the 256 node overhead of Fig. 10. We note that while utilizing a secondary network improves the performance of our replication scheme, it cannot fully eliminate all networking overhead. Ultimately, each node must individually transmit and receive all checkpoint data (as well as write the replicas to disk). Processing such data leads to a small amount of overhead for increasing amounts of data.

5.6 Comparing the Costs of SAN, PFS, and Local Storage

Some have argued that contention for SAN resources could be mitigated through the use of a checkpoint backplane with a dedicated SAN/parallel file system [38]. However a SAN solution capable of handling the massive amounts of data generated by tomorrow's supercomputers would simply cost too much. We consider two commercial SAN-based systems, one is a mid-range solution that includes

the cost of a commercial parallel file system. The second is considered a high-end system that, due to incompatibilities, does not utilize a parallel file system. Both are designed to support a cluster of approximately 1000 nodes. We do not list vendor names as the vendors wished to remain anonymous. Current pricing for a 54 TB (usable) mid-range SAN, including a parallel file system, is approximately $1.1 million plus $16,570 annually in projected energy costs. A high-end SAN, with 54 TB of usable storage, would cost nearly $1.4 million plus $34,632 annually in projected energy costs. Neither system includes the cost of IO servers, host bus adapters (HBAs), or a backup system, and both include heavy academic discounts for the hardware.

To equip a similar 1000 node computational cluster with up to 1 TB of usable local storage for checkpointing could be accomplished at a cost of only $644 per node. This includes the cost of two 1 TB (SATA II 7200 RPM) hard disks[1] ($255 each) and a hardware RAID 1 controller[2] ($134). The cost drops to only $334 per node if 500 GB hard disks[3] are used instead ($100 each).

Comparing the usable storage of the local storage vs. SAN backplane solutions, we are left with 500TB of local storage if 500GB hard disks are used and 1 *petabyte* of usable storage if 1TB hard disks are used instead. Thus, local storage/replication solution can be purchased for only 25-58% of the SAN backplane cost, with lower checkpointing overhead.

Of course it should be noted that the addition of hard disks may increase the failure rate of an individual node. This may be due to an increase in heat produced by nodes, which may result in a strain on the cooling capabilities of the data center. However, whether the disks used for checkpointing are stored locally at each node, or centrally in a SAN, they will output similar amounts of heat. Ultimately, the data center's cooling system must remove the heat regardless of where it's generated. Further, recent studies suggest that disks themselves are less susceptible to heat than originally thought [39]. This, combined with the more recent emphasis on data-intensive computing [31], suggests that next-generation supercomputers may require local storage despite the possibility of increased failures.

6 Conclusions and Future Work

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 storage. We comprehensively addressed this issue with a comparison against all major storage techniques, including SAN-based strategies and commercial parallel file systems. 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 reliance on network storage, and is able to take advantage of additional network availability (such as Myrinet or additional ethernet networks) to further reduce the overhead of checkpoint replication. We hope to extend this work to provide support for additional network interconnects, such as InfiniBand and Myrinet, using their native interfaces rather than IP. We believe that this will result in further increases in performance. Finally, we showed that our local storage/replication solution is far less expensive than a dedicated checkpointing backplane, costing as little as 25% of a SAN-based solution.

We are currently investigating the use of coding strategies for further reduction in checkpointing overhead. This has the potential to reduce the amount of data sent over the network as full replicas need not be sent. However, the encoding overhead is a concern that must be balanced against the reduction in replication overhead.

1. Western Digital WD10EACS
2. 3ware 8006-2LP
3. Western Digital WD5000AAJS

Acknowledgments

We would like to acknowledge the input of the anonymous reviewers whose suggestions have greatly improved the quality of this article. We also acknowledge Sam Guercio Jr. of the Center for Computational Research at the University at Buffalo, as well as Aragorn Steiger and Michael Thompson from Wayne State University for their invaluable help in setting up the test environments. 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.

References

[1] The MPI Forum, "MPI: A Message Passing Interface," in *SC '93: Proceedings of the annual Supercomputing Conference.* ACM Press, 1993, pp. 878–883.

[2] 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 35^{th} annual International Conference on Parallel Processing*. IEEE Computer Society, 2006, pp. 471–478.

[3] G. Burns, R. Daoud, and J. Vaigl, "LAM: An Open Cluster Environment for MPI," in *Proceedings of the Supercomputing Symposium*. IEEE Computer Society, 1994, pp. 379–386.

[4] 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*, vol. 19, no. 4, pp. 479–493, 2005.

[5] J. P. Walters and V. Chaudhary, "A Scalable Asynchronous Replication-Based Strategy for Fault Tolerant MPI Applications," in *HiPC '07: the International Conference on High Performance Computing, LNCS 4873*. Springer-Verlag, 2007, pp. 257–268.

[6] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks," *International Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[7] J. M. Squyres and A. Lumsdaine, "A Component Architecture for LAM/MPI," in *Proceedings of the 10^{th} European PVM/MPI Users' Group Meeting, LNCS 2840*. Springer-Verlag, 2003, pp. 379–387.

[8] InfiniBand Trade Association, "InfiniBand," 2007, http://www.infinibandta.org/home.

[9] Myricom, "Myrinet," 2007, http://www.myricom.com/.

[10] J. Hursey, J. M. Squyres, T. I. Mattox, and A. Lumsdaine, "The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI," in *IPDPS '07: Proceedings of the* 21^{st} *annual International Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2007.

[11] E. N. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," *ACM Computing Surveys*, vol. 34, no. 3, pp. 375–408, 2002.

[12] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent Checkpointing under Unix," in *In Proceedings of the USENIX Winter Technical Conference*. USENIX Association, 1995, pp. 213–223.

[13] J. Duell, "The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart," Lawrence Berkeley National Lab, Tech. Rep. LBNL-54941, 2002.

[14] R. Gioiosa, J. C. Sancho, S. Jiang, and F. Petrini, "Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers," in *SC '05: Proceedings of the annual Supercomputing Conference*. IEEE Computer Society, 2005, pp. 9–23.

[15] Y. Zhang, D. Wong, and W. Zheng, "User-Level Checkpoint and Recovery for LAM/MPI," *SIGOPS Operating Systems Review*, vol. 39, no. 3, pp. 72–81, 2005.

[16] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "A Job Pause Service under LAM/MPI+BLCR for Transparent Fault Tolerance," in *IPDPS '07: Proceedings of the* 21^{st} *International Parallel and Distributed Processing Symposium*. IEEE Computer, 2007, pp. 116–125.

[17] 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 annual Supercomputing Conference*. IEEE Computer Society, 2005, pp. 32–46.

[18] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *OOPSLA '93: Proceedings of the conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 1993, pp. 91–108.

[19] C. Huang, G. Zheng, S. Kumar, and L. V. Kalé, "Performance Evaluation of Adaptive MPI," in *PPoPP '06: Proceedings of the* 11^{th} *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, 2006, pp. 306–322.

[20] S. Chakravorty and C. Mendes and L. V. Kalé, "Proactive Fault Tolerance in MPI Applications via Task Migration," in *HiPC '06: Proceedings of the* 13^{th} *International Conference on High Performance Computing, LNCS 4297*, 2006, pp. 485–496.

[21] S. Chakravorty and L. V. Kalé, "A Fault Tolerance Protocol with Fast Fault Recovery," in *IPDPS '07: Proceedings of the* 21^{st} *annual International Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2007, pp. 117–126.

[22] G. Zheng, L. Shi, and L. V. Kalé, "FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI," in *Cluster '04: Proceedings of the International Conference on Cluster Computing*. IEEE Computer Society, 2004, pp. 93–103.

[23] J. S. Plank, "Improving the Performance of Coordinated Checkpointers on Networks of Workstations using RAID Techniques," in *SRDS '96: Proceedings of the* 15^{th} *Symposium on Reliable Distributed Systems*. IEEE Computer Society, 1996, pp. 76–85.

[24] J. S. Plank and L. Kai, "Faster Checkpointing With N+1 Parity," in *SFTC '94: Proceedings of the* 24^{th} *annual International Symposium on Fault-Tolerant Computing*, 1994, pp. 288–297.

[25] R. Y. de Camargo, R. Cerqueira, and F. Kon, "Strategies for Storage of Checkpointing Data Using Non-Dedicated Repositories on Grid Systems," in *Proceedings of the* 3^{rd} *International Workshop on Middleware for Grid Computing*. ACM Press, 2005.

[26] X. Ren, R. Eigenmann, and S. Bagchi, "Failure-Aware Checkpointing in Fine-Grained Cycle Sharing Systems," in *HPDC '07: Proceedings of the 16th International Symposium on High Performance Distributed Computing*. ACM Press, 2007, pp. 33–42.

[27] Z. Chen, G. E. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra, "Fault Tolerant High Performance Computing by a Coding Approach," in *PPoPP '05: Proceedings of the* 10^{th} *annual ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, 2005, pp. 213–223.

[28] P. Nath, B. Urgaonkar, and A. Sivasubramaniam, "Evaluating the Usefulness of Content Addressable Storage for High-Performance Data Intensive Applications," in *HPDC '08: Proceedings of the* 17^{th} *International Symposium on High Performance Distributed Computing*. ACM Press, 2008, pp. 35–44.

[29] J. Cao, Y. Li, and M. Guo, "Process Migration for MPI Applications based on Coordinated Checkpoint," in *ICPADS '05: Proceedings of the* 11^{th} *annual International Conference on Parallel and Distributed Systems*. IEEE Computer Society, 2005, pp. 306–312.

[30] V. Zandy, "Ckpt: User-Level Checkpointing," 2005, http://www.cs.wisc.edu/~zandy/ckpt/.

[31] R. E. Bryant, "Data-Intensive Supercomputing: The Case for DISC," Carnegie Mellon University, School of Computer Science, Tech. Rep. CMU-CS-07-128, 2007.

[32] S. Gurumurthi, "Should Disks be Speed Demons or Brainiacs?" *SIGOPS Operating Systems Review*, vol. 41, no. 1, pp. 33–36, 2007.

[33] "Top500 list http://www.top500.org," 2008.

[34] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK Benchmark: Past, Present, and Future," *Concurrency and Computation: Practice and Experience*, vol. 15, pp. 1–18, 2003.

[35] 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* 18^{th} *annual Supercomputing Conference*. ACM Press, 2006, pp. 127–140.

[36] 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* 16^{th} *International Conference on Supercomputing*. ACM Press, 2002, pp. 84–95.

[37] S. Hoory, N. Linial, and A. Wigderson, "Expander Graphs and their Applications," *Bulletin of the American Mathematical Society*, vol. 43, no. 4, pp. 439–561, 2006.

[38] "CiFTS: Coordinated infrastructure for Fault Tolerant Systems. http://www.mcs.anl.gov/research/cifts/," 2008.

[39] E. Pinheiro, W.-D. Weber, and L. A. Barroso, "Failure trends in a Large Disk Drive Population," in *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies*. USENIX Association, 2007, pp. 17–28.

John Paul Walters is currently a postdoctoral researcher at the University at Buffalo. He received his PhD from Wayne State University in Detroit, Michigan in 2007 and his bachelor's degree from Albion College in Albion, Michigan in 2002. His research spans many areas of high performance computing, including fault-tolerance, cluster scheduling, and accelerating life sciences applications.

Vipin Chaudhary is an Associate Professor of Computer Science and Engineering and the New York State Center of Excellence in Bioinformatics and Life Sciences at University at Buffalo, SUNY. Earlier he was the Senior Director of Advanced Development at Cradle Technologies, Inc. Prior to that he was the Chief Architect with Corio, Inc. In addition, he is on the advisory boards of several startup companies. He received the B.Tech. (Hons.) degree in Computer Science and Engineering from the Indian Institute of Technology, Kharagpur, in 1986, the MS degree in Computer Science, and the Ph.D. degree in Electrical and Computer Engineering from The University of Texas at Austin, in 1989 and 1992, respectively. His current research interests are in the area of Computer Assisted Diagnosis and Interventions; Medical Image Processing; Grid and High Performance Computing and its applications in Computational Biology and Medicine; Embedded Systems Architecture and Applications; and Sensor Networks and Pervasive Computing. He was awarded the prestigious President of India Gold Medal in 1986 for securing the first rank amongst graduating students in IIT.