

# Interconnect Agnostic Checkpoint/Restart in Open MPI

Joshua Hursey  
Indiana University  
Open Systems Laboratory  
Bloomington, IN USA  
jjhursey@osl.iu.edu

Timothy I. Mattox  
Indiana University  
Open Systems Laboratory  
Bloomington, IN USA  
timattox@osl.iu.edu

Andrew Lumsdaine  
Indiana University  
Open Systems Laboratory  
Bloomington, IN USA  
lums@osl.iu.edu

## ABSTRACT

Long running High Performance Computing (HPC) applications at scale must be able to tolerate inevitable faults if they are to harness current and future HPC systems. Message Passing Interface (MPI) level transparent checkpoint/restart fault tolerance is an appealing option to HPC application developers that do not wish to restructure their code. Historically, MPI implementations that provided this option have struggled to provide a full range of interconnect support, especially shared memory support. This paper presents a new approach for implementing checkpoint/restart coordination algorithms that allows the MPI implementation of checkpoint/restart to be interconnect agnostic. This approach allows an application to be checkpointed on one set of interconnects (e.g., InfiniBand and shared memory) and be restarted with a different set of interconnects (e.g., Myrinet and shared memory or Ethernet). By separating the network interconnect details from the checkpoint/restart coordination algorithm we allow the HPC application to respond to changes in the cluster environment such as interconnect unavailability due to switch failure, re-load balance on an existing machine, or migrate to a different machine with a different set of interconnects. We present results characterizing the performance impact of this approach on HPC applications.

## 1. INTRODUCTION

High Performance Computing (HPC) applications are running longer to solve more complex problems and scaling farther to take advantage of leadership class HPC systems. Unfortunately, as the HPC systems grow to meet the demand of the applications, their mean-time-between-failure falls dramatically. This means that the HPC applications are increasingly likely to encounter hardware failure. The resultant process failure may cause the loss of hours or days of computation. This problem is becoming so prevalent that some HPC system manufactures are advising application developers to prepare for inevitable failure by incorporating fault tolerance techniques [30].

The checkpoint/restart technique is a commonly used technique for implementing fault tolerance in HPC applications. Transparent solutions are appealing since they do not require the applica-

tion developers to alter their algorithms to take advantage of the technique. Transparent checkpoint/restart techniques are typically implemented around an MPI implementation. The Message Passing Interface (MPI) [22] is the *de-facto* standard for message passing in HPC systems. Since MPI controls the interactions between processes, checkpoint/restart techniques can take advantage of the MPI implementation's knowledge of the distributed system's state to ensure correctness in the checkpoint algorithm.

In this paper we discuss a new approach to implementing checkpoint/restart fault tolerance in MPI that enables us to support a variety of interconnects including Ethernet, InfiniBand, Myrinet, and shared memory. Additionally, this technique allows us to adapt the interprocess communication mechanism of the restarted application to take advantage of the new process placement and/or interconnect availability. This opens the door to improved performance in load balancing systems and flexibility in the types of machines an application can migrate between.

Current transparent checkpoint/restart solutions for MPI struggle with interconnect support. Most implementations support Ethernet communication and sometimes a high speed interconnect such as Myrinet or InfiniBand. Rarely do implementations support shared memory, thus limiting performance on the multi-core processing environments prevalent in modern HPC systems. Most of these implementations require that the restarted MPI application use the same set of interconnects for interprocess communication on restart as was configured at the time of the checkpoint. This has significant performance implications for applications that wish to restart in a different process layout or on a different machine.

In our approach to implementing checkpoint/restart fault tolerance in MPI, we lift the checkpoint/restart coordination algorithm out of the interconnect driver and place it above the MPI internal point-to-point stack. This placement allows the checkpoint/restart coordination algorithm to be interconnect agnostic; therefore, specific interconnect behaviors like RDMA operations do not muddle the coordination algorithm. By not lifting the coordination algorithm all the way out of the MPI library our implementation sits below much of the MPI interface (including collectives), leading to a more focused coordination algorithm. For example, our approach allows an MPI application to run with shared memory and InfiniBand, then restart in a different process layout using shared memory and Myrinet or Ethernet. This capability allows an application to respond to changes in the cluster environment such as interconnect unavailability due to switch failure, re-load balance on an existing machine, or migrate to a different machine with a potentially different set of interconnects.

The checkpoint/restart implementation described in this paper opens the door to adaptive and improved performance for restarted applications. Migration between machines is constrained by the process level checkpoint/restart system that is chosen. For this paper we chose the Berkeley Laboratory Checkpoint/Restart (BLCR) system [8, 10]. BLCR does not permit the checkpointing on one type of machine architecture and restarting on a different architecture. Given this limitation, this paper will instead focus on responding to cluster configuration changes (loss of switches) and rebalancing load within a single machine.

## 2. RELATED WORK

Distributed checkpoint/restart requires coordination algorithms to ensure consistent recoverable states in the distributed system. Checkpoint/restart coordination protocols have a relatively long history in computer science [2, 6, 19, 26]. Elnozahy, et.al., in [9], present a survey of many rollback recovery techniques, as well as a discussion of both checkpoint/restart and message logging techniques.

Over the years there have been many attempts at integrating fault tolerance techniques, including checkpoint/restart, into message passing software and applications. Most depend upon a system-level checkpoint/restart service to capture and restore the state of a single process. Checkpoint/restart services, such as libckpt [25] and BLCR (Berkeley Lab's Checkpoint/Restart) [8, 10] differ in their APIs, amount of state preserved, quality of implementation, and process coverage.

These attempts at implementing fault tolerance techniques into message passing libraries also differ in the checkpoint/restart coordination protocol that they employ. Starfish [1] provides support for coordinated and uncoordinated checkpoint/restart protocols. CoCheck [32] uses the Condor checkpoint/restart system and a Ready Message checkpoint/restart coordination protocol. That protocol allowed a subset of the processes in the application to quiesce their network channels before taking a checkpoint. Since both of these implementations only support Ethernet communication, we distinguish ourselves by adding support for a wider variety of interconnects.

LAM/MPI [5] incorporated transparent checkpoint/restart functionality with support for Ethernet [29] and later Myrinet GM using the BLCR library. They used an all-to-all bookmark exchange coordination algorithm implemented as part of the TCP/IP and Myrinet GM interconnect drivers. Though we use a similar coordination algorithm, we distinguish ourselves from this work by lifting the checkpoint state out of the individual drivers, allowing for greater flexibility in network configuration on restart. LAM/MPI was unique in its support for a modular interface to the checkpoint/restart services on a given machine [28]. It supported BLCR and a user-level callback system called *SELF*.

Transparent checkpoint/restart in MPICH-GM using the Myrinet GM driver has also been demonstrated [20]. They use a 2-phase coordination procedure based, in-part, on CoCheck's Ready Message protocol [32]. They implement their coordination algorithm as part of the GM driver relying on the FIFO message ordering provided therein. The GM Driver supports shared memory communication for peers on the same node. Even though their coordination algorithm is influenced by the RDMA semantics of the GM driver, they do support reconfiguration between shared memory and Myrinet GM communication, though they require Myrinet to be available upon restart to all processes in the application. We

distinguish ourselves from this study by adding support for additional interconnects and by lifting the coordination protocol out of the device driver so that it does not need to be further complicated by the RDMA semantics of high speed interconnects like Myrinet.

The MPICH-V project focuses its effort towards message logging techniques with the MPICH-V1, MPICH-V2 and MPICH-Vcausal implementations [3]. They also have a MPICH-Vcl implementation that supports coordinated checkpoint/restart using a Chandy/Lamport coordination algorithm [6]. Comparing their coordinated checkpointing and message logging approaches they showed that it is often advantageous to use coordinated checkpointing algorithm until the fault frequency reached a crossover point after which it became advantageous to use message logging [7]. They demonstrate their implementation running on clusters with 100 Mbit/s Ethernet, Myrinet, and SCI networks [7, 21]. To run on the Myrinet and SCI networks they take advantage of the Ethernet emulation provided by these interconnects and do not interact directly with the hardware drivers. Though this saves in the overall complexity of the solution, it comes at a significant performance loss. We distinguish ourselves from this project by using the native drivers for communication instead of relying on the Ethernet emulation.

MVAPICH2 [12] demonstrated transparent MPI checkpoint/restart functionality over InfiniBand [18] interconnects. This work highlights the complexity of dealing with the OS bypass technique used by InfiniBand. They show that the only way to properly handle such an interconnect driver is to completely shutdown all network connections before a checkpoint and re-establish them directly after a checkpoint. They implement a Chandy/Lamport [6] style coordination algorithm at the InfiniBand driver level operating on network-level messages instead of MPI-level messages. This coordination algorithm relies on FIFO message ordering provided by the interconnect driver. In a later study they use a group-based coordination algorithm [11]. This algorithm is similar to the staggered algorithm presented by Vaidya [34] which is used to minimize the stress on the file system during checkpointing. We distinguish ourselves from these studies by supporting InfiniBand interconnects alongside other interconnects such as shared memory, Myrinet, and Ethernet. Further, we do not incorporate our coordination mechanism inside the InfiniBand driver, but instead operate above the point-to-point stack on MPI level messages.

Adaptive MPI [15, 16] implements migratable virtual MPI processes as user level threads. Adaptive MPI provides a checkpoint/restart solution closer to application level checkpointing than system level checkpointing since it is not transparent to the user level application. The MPI application must place checkpoint function calls into its code at points when it can guarantee that no messages are being transferred between processes [14, 36]. Adaptive MPI then saves the state of the thread to disk as a checkpoint. Adaptive MPI is implemented on top of Charm++ which, in turn, is implemented on top of a native MPI implementation. Being this far removed from the interconnects significantly degrades performance. Additionally, since MPI processes are implemented as user-level threads and share the global address space, Adaptive MPI places additional constraints on the MPI application such as prohibiting the use of global variables. We distinguish ourselves from this work by providing a transparent checkpoint/restart solution that is closer to the interconnects allowing for improved performance, algorithmic clarity and flexibility on restart.

The  $C^3$  implementation [4] also presents a checkpoint/restart solution for MPI applications through the use of a specialized compiler and wrappers around the MPI interface. This implementation must virtualize the entire MPI interface instead of just the point-to-point stack, making it much more complex than our solution, especially when dealing with collective operations. In order to properly restart the application, the authors had to modify the MPI standard semantics to allow a process to go through `MPI_INIT` and `MPI_FINALIZE` more than once. By operating inside the MPI library our checkpoint/restart implementation can be more focused, and requires no alterations of the MPI standard.

DejaVu [27] is a user-level transparent checkpoint/restart system, and is unique in that, with its socket virtualization, it is able to checkpoint/restart multiprocess applications, including MPI applications, that use only sockets for communication. It uses a hybrid coordinated checkpoint/restart and online logging protocol. In this work the authors also demonstrated the ability to checkpoint and restart MPI applications using InfiniBand connections by incorporating their checkpoint/restart coordination protocol into the MVA-PICH InfiniBand driver. We distinguish ourselves from this work by supporting additional network drivers, and lifting the coordination algorithm out of the InfiniBand driver.

The work presented in this paper combines many of the best practices from these previous works into the Open MPI implementation. We extend the previous work in two fundamental ways. First, we separate the coordination protocol from the interconnect behavior by lifting the coordination protocol above the internal point-to-point implementation in Open MPI making the coordination protocol interconnect agnostic. Secondly, we support a wider variety of interconnects than any previous transparent checkpoint/restart MPI implementation. This allows applications greater flexibility in the network configurations they can use both while checkpointing and restarting. For example, applications are able to checkpoint in an InfiniBand and shared memory environment and restart in a Myrinet and shared memory environment with a different process layout on each compute node.

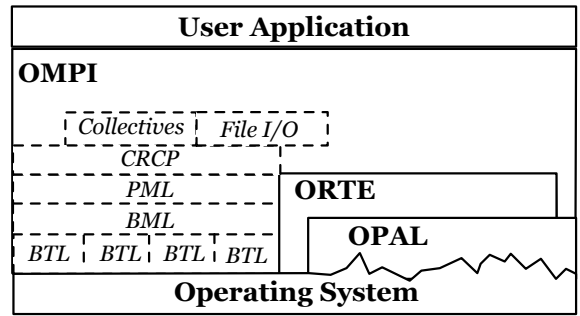
### 3. IMPLEMENTATION

We implement our solution as part of the Open MPI [13] implementation of the MPI standard [22]. We take advantage of the entire checkpoint/restart infrastructure [17] incorporated within Open MPI. In this paper we focus on the Checkpoint/Restart Coordination Protocol (CRCP) framework and the Point-to-Point Management Layer (PML) framework that it abstracts. By taking advantage of this abstraction we can adjust the peer-to-peer connections on restart to be as efficient as possible, without muddling the coordination protocol with the behavior of individual interconnect drivers.

#### 3.1 Open MPI Architecture

Open MPI is an open source, high performance, MPI-2 compliant implementation of the MPI standard. Open MPI is designed around the Modular Component Architecture (MCA) [31]. The MCA provides a set of component frameworks to which a variety of point-to-point, collective, and other MPI related algorithms can be implemented. The MCA allows for runtime selection of the best set of components to properly support an MPI application in execution.

MCA frameworks in Open MPI are divided into three distinct layers: Open Portable Access Layer (OPAL), Open Run Time En-



**Figure 1: Open MPI MCA layers including OMPI layer frameworks.**

vironment (ORTE), and Open MPI (OMPI). OPAL is composed of frameworks that are concerned with portability across operating systems. ORTE is composed of frameworks that are concerned with the launching, monitoring, and cleaning up of processes in the HPC environment. The ORTE layer sits above the OPAL. The OMPI layer sits above both the OPAL and ORTE layers. The OMPI layer is composed of frameworks to support the MPI interfaces exposed to the application layer. This layering is illustrated in Figure 1.

In the OMPI layer, most frameworks (e.g., Collectives, File I/O) sit above the Point-to-Point Management Layer (PML). The PML controls all point-to-point communication in Open MPI and exposes MPI point-to-point semantics. The PML framework is a stack of three frameworks all working together to provide flexibility and performance to the application. The PML framework breaks messages with potentially complex data types into byte streams that can be consumed by the lower layers. The Byte Transfer Layer (BTL) framework encapsulates each of the supported interconnects in Open MPI (e.g., shared memory, Ethernet, InfiniBand, Myrinet MX, etc.). Between the PML and BTL is the BTL Management Layer (BML) that provides the ability to stripe a message across multiple interconnects. This may include using multiple paths between peers, and using multiple interfaces of a single driver.

The PML exchanges connectivity information, at startup, during a module exchange or `modex` procedure. This connectivity information is used to determine the best possible routes between all peers in the system. This provides the necessary information to all processes in the MPI application so that they can establish communication with any other peer in the MPI application.

#### 3.2 Checkpoint/Restart Infrastructure

The checkpoint/restart infrastructure integrated into Open MPI is defined by process and job levels of control. These two levels of control work in concert to create stable job-level checkpoints (called *global snapshots*) from each individual process-level checkpoint (called *local snapshots*).

Job-level control is composed of two frameworks: Snapshot Coordinator (SnapC) and File Management (FileM). The SnapC framework controls the checkpoint life-cycle, from distributing the checkpoint request to all processes, to monitoring their progress, to collecting and verifying the final checkpoint. The File Management (FileM) framework focuses on the movements of individual local checkpoints to and from storage devices. In this paper, we will assume that FileM is not needed because all processes save their

local snapshots directly to a central stable storage medium (e.g., SAN, parallel file system). Additionally we assume a centralized SnapC coordinator. Since job-level control is not the focus of this paper, exploring alternative configurations of these frameworks is left for future work.

This paper instead focuses on the frameworks that support process level control in the checkpoint/restart infrastructure. The Checkpoint/Restart Service (CRS) framework provides an API to the single process checkpoint and restart service on a particular machine. Open MPI currently supports system-level checkpoint/restart with BLCR, and user-level checkpoint/restart with *SELF*, which is a callback based mechanism. For this paper we will be using the BLCR system. The Interlayer Notification Coordinator (INC) provides internal coordination among the many levels inside the Open MPI library to prepare for a checkpoint, continue after a checkpoint or restart a process from a checkpoint. This coordination may include flushing caches and activation of checkpoint/restart specific code paths.

The Checkpoint/Restart Coordination Protocol (CRCP) framework implements coordination protocols that control for in-flight messages [6]. The CRCP framework is positioned above the PML layer and tracks all messages moving in and out of the point-to-point stack. For this paper we are using an all-to-all bookmark exchange algorithm similar to the one used by LAM/MPI [29], except that instead of operating on bytes we are operating on entire MPI messages. In this algorithm we exchange message totals between peers on checkpoint, then wait for the totals to equalize. This equilibrium indicates a quiet channel and guarantees no in-flight messages. Care is taken in the coordination algorithm to avoid deadlock during the draining process, and to preserve MPI semantics.

Positioning the coordination algorithm above the point-to-point stack is different than most transparent MPI checkpoint/restart implementations. Most alternative implementations position their coordination protocols in the individual interconnect drivers. By implementing the algorithm in the interconnect driver, the algorithm needs to track bytes being moved through the interface, monitor special behaviors of the device such as RDMA operations, and, for the most part, does not need to worry as much about MPI level semantics. However, forcing the state to be saved as part of the interconnect driver within the point-to-point stack requires the application to be restarted in a similar process layout. This hinders performance by restricting the ability to choose the fastest routes upon restart. Additionally, the coordination algorithm often becomes muddled with device driver restrictions and requires the driver to provide strict FIFO ordering of messages.

By lifting the algorithm out of the interconnect driver and placing it above the point-to-point stack we are able to save the point-to-point state at a high enough level to allow for the reconfiguration of the interconnects upon restart. This reconfiguration enables us to adapt to the new process layout to achieve better performance. This is at the cost of a slightly more complex implementation of the coordination algorithm since it must operate on entire messages with MPI semantic restrictions.

### 3.3 Interconnect Driver Support

There are three distinct phases of a checkpoint operation: pre-checkpoint, continue, and restart. In the pre-checkpoint phase the application is provided an opportunity to prepare for a requested checkpoint. This involves bringing external resources (e.g., files,

network connections) to a stable, checkpointable state. The continue phase occurs just after a checkpoint has been taken and allows the application to recover any external resources that it may have stabilized or suspended during the pre-checkpoint phase. The restart phase occurs when the process is restarted from stable storage and provides the application with an opportunity to flush caches, and reconstruct any necessary information needed to continue normal operation.

The Checkpoint/Restart Coordination Protocol (CRCP) is positioned above the PML and ensures that the lower levels of the point-to-point stack have been drained of all messages coming from and going to this peer. We can take advantage of this assurance when moving the point-to-point stack through the three phases of checkpointing. It should be noted that we do not require that all processes drain their messages before checkpointing any individual process, only that the individual process taking the checkpoint, at that moment, must be drained of messages. In future work we plan to explore other coordination algorithms that allow for even looser synchrony between processes [11, 33, 34].

After the CRCP has completed its pre-checkpoint quiescence operation the INC may choose to shutdown all active interconnect drivers in order to avoid problematic interactions between checkpoint/restart services (e.g., BLCR) and interconnect drivers. However, in the INC, we only want to do the absolute minimum amount of work needed to bring the process into a checkpointable state in order to minimize the failure-free overhead [9]. As a performance optimization our implementation allows interconnect drivers to indicate if they are *checkpoint friendly* meaning that they can be safely used with the active checkpoint/restart service on the system. This optimization is a result of recognizing that shutting down and re-initializing all of the interconnect drivers during a checkpoint operation is often expensive and not always necessary. Since this optimization occurs in the INC and outside of the CRCP, the CRCP remains interconnect agnostic even when this optimization is enabled. The performance benefits of this optimization are analyzed in Section 4.3.

In the restart phase we need to first clear out the previous set of interconnects and connectivity (i.e., `modex`) information since the machine set and corresponding network address will have typically changed. Then we re-exchange the `modex` to get the new connectivity information and reconnect the processes selecting a new set of interconnect drivers (BTLs).

#### 3.3.1 TCP/Ethernet Driver

The Ethernet driver in Open MPI (TCP BTL) does not need to do anything during the pre-checkpoint phase since open sockets are preserved across a checkpoint operation, thus classifies as checkpoint friendly. Since nothing was closed during the pre-checkpoint phase, nothing needs to occur during the continue phase. The restart phase must make sure to close the old socket file descriptors before the `modex` can occur so as not to waste resources.

#### 3.3.2 Shared Memory Driver

The shared memory driver in Open MPI (SM BTL) is also checkpoint friendly so also does not need to do anything during the pre-checkpoint phase, because the checkpoint/restart service will not checkpoint the contents of open files or shared memory segments. BLCR is able to checkpoint the shared memory segment shared by processes of the same group (family of processes) as long as they are restarted together. Since our goal is flexibility in the process

layout on restart, we do not take advantage of this feature. Instead, BLCR will keep a reference to the memory-mapped shared memory file, but not its contents. Upon restart we must make sure to close the stale file descriptor before the `modex` operation. The `modex` reestablishes the shared memory segment with the set of peers on the node at restart time which may be different than when originally checkpointed.

### 3.3.3 InfiniBand Driver

The InfiniBand driver in Open MPI (OpenIB BTL) uses the OpenFabrics [24] interface to interact with a wide range of InfiniBand hardware. In the pre-checkpoint phase we must close the driver which entails closing the ports and releasing all resources allocated on the InfiniBand card, as confirmed by [12]. Experimentally, if we use BLCR with the InfiniBand driver still active, then the state of the kernel becomes unstable and panics which results in node loss. Since we close the connections during pre-checkpoint, on continue we must reopen the InfiniBand driver and reestablish connections with our peers. So, because we have to re-exchange the `modex` and reconnect peers, the continue and restart phases look similar at the interconnect level. However, the continue operation can take advantage of the fact that the processes have not changed position, and are likely still reachable via the previously established set of interconnects. So there is no need, for example, to reconnect shared memory segments since processes on the same node have not moved. On restart peer processes may have moved to different machines, so we need to consider the new topology information when establishing routes.

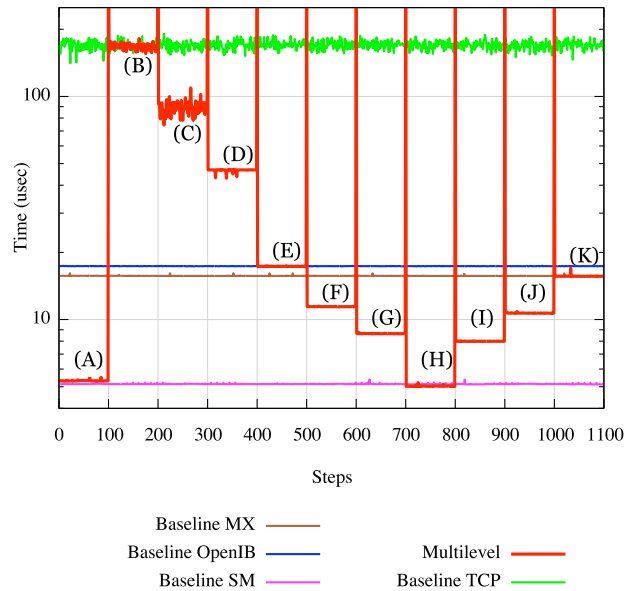
### 3.3.4 Myrinet Driver

The Myrinet driver in Open MPI (MX BTL) uses the Myrinet MX interface to interact with Myrinet interconnect hardware. In the pre-checkpoint phase we must close the driver which entails closing the ports and releasing all resources allocated on the Myrinet card; this is similar to the procedure with the InfiniBand driver. However, in this case, the limitation is not kernel instability, but reconnecting to the device driver upon restart. On restart, BLCR will attempt to reallocate the open endpoint file descriptors with the network card, and receive a permission denied error and fail to restart the application. To get around this limitation, we shut down the MX BTL before a checkpoint. On continue we must reopen the connection to the Myrinet card and reestablish connections with our peers. This requires a `modex` operation to re-exchange the new contact information. As future work, we plan on looking into ways around this limitation.

## 4. RESULTS

This section explores the performance of our implementation and demonstrates the migration of processes between different network topologies. In these tests we are using the Sif cluster at Indiana University. Sif is an 8 node Dual Intel 1.86 GHz Quad-Core Xeon machine with 16 GB of memory per compute node. It is connected with gigabit Ethernet, InfiniBand, and Myrinet. It is running Red-Hat Linux 2.6.18-53, BLCR 0.7.3, and Open MPI v1.3.0.

In this section we will be using the NAS Parallel Benchmark Suite (version 2.4) [23] to represent a class of typical HPC application kernels. We used 4 of the 8 kernels, specifically the LU class C, EP class D, BT class C, and SP class C kernels. This decision was based on two criteria. First the kernel must run for long enough to perform the test at 32 or 36 processes, which meant running for longer than one minute, which excluded the IS kernel. Secondly,



**Figure 2: Continuous latency test with 8 processes exchanging an 8KB message and migrating between different machine configurations. Spikes indicate time spent on disk between the checkpoint and the subsequent restart.**

due to storage space restrictions on the SAN, the kernels snapshot image must be less than 15 GB in total size, which excluded the MG class C, FT class D, and CG class D kernels.

We assess the impact of checkpointing a real application called GROMACS [35]. GROMACS is a molecular dynamics application. For this test we used the DPPC benchmark from version 3.0 of their benchmark suite. In our analysis we ran these benchmarks using 4 of the 8 cores on each of the compute nodes, as to alleviate some of the memory contention on the nodes.

### 4.1 Network Topology Migration

To demonstrate the migration of an MPI application between different network topologies we took advantage of the large SMP base of Sif and occasionally forced Open MPI to choose certain network drivers simulating different network configurations and availabilities. We used a continuous latency test that measures the time taken for an 8KB message to travel around a ring of 8 processes. This test allows us to account for the shared memory and interconnect latency as if they were a single value. As you can see in Figure 2 the Ethernet network is fairly noisy on Sif due to administrative traffic on this particular machine.

In our demonstration we checkpoint and terminate the MPI application every 100 steps. Then we restart it in a different network topology and let it run for another 100 steps before checkpointing and terminating it again. The network topology progression in Figure 2 is as follows:

- (A) 8 processes on 1 node using shared memory
- (B) 8 processes 1 on each of 8 nodes using Ethernet
- (C) 8 processes 2 on each of 4 nodes using Ethernet and shared memory
- (D) 8 processes 4 on each of 2 nodes using Ethernet and shared

(a) NetPIPE 1 byte latency overhead

Interconnect	No C/R	With C/R	Overhead %
Ethernet (TCP)	49.92 usec	50.01 usec	0.09 usec (0.2%)
InfiniBand	8.25 usec	8.78 usec	0.53 usec (6.4%)
Myrinet MX	4.23 usec	4.81 usec	0.58 usec (13.7%)
Shared Memory	1.84 usec	2.15 usec	0.31 usec (16.8%)

(b) NetPIPE bandwidth overhead

Interconnect	No C/R	With C/R	Overhead %
Ethernet (TCP)	738 Mbps	738 Mbps	0 Mbps (0%)
InfiniBand	4703 Mbps	4703 Mbps	0 Mbps (0%)
Myrinet MX	8000 Mbps	7985 Mbps	15 Mbps (0.2%)
Shared Memory	5266 Mbps	5258 Mbps	8 Mbps (0.2%)

**Table 1: NetPIPE 1 byte latency and bandwidth CRCP framework overheads.**

memory

- (E) 8 processes 1 on each of 8 nodes using InfiniBand
- (F) 8 processes 2 on each of 4 nodes using InfiniBand and shared memory
- (G) 8 processes 4 on each of 2 nodes using InfiniBand and shared memory
- (H) 8 processes on 1 node using shared memory
- (I) 8 processes 4 on each of 2 nodes using Myrinet and shared memory
- (J) 8 processes 2 on each of 4 nodes using Myrinet and shared memory
- (K) 8 processes 1 on each of 8 nodes using Myrinet

## 4.2 Failure Free Overhead

Failure-free overhead is the overhead seen by the application during normal operations when a failure does not occur. This overhead includes the time taken by the CRCP protocol monitoring the message traffic, and the additional time to completion for the application when a checkpoint is taken. One of our goals is to minimize the failure-free overhead seen by the application.

One assessment of the failure free overhead is the overhead seen in the latency and bandwidth parameters between two communicating peers without taking a checkpoint. This measurement accounts for the overhead of the CRCP framework monitoring the message traffic though the system. Using NetPIPE we can assess the latency and bandwidth effects of wrapping the PML layer in Table 1. The NetPIPE results show that the differences in bandwidth is negligible, and the 1 byte latency is varies between 0.09 and 0.58 microseconds depending on the interconnect.

Another assessment of the failure free overhead is the performance impact on completion time for an application when various numbers of checkpoints are taken. For this assessment we look at checkpointing to both a globally accessible NFS disk and to the local disk on each machine. The local disk checkpoint time is provided as a basis for comparison in order to highlight the impact of the file system on the performance of the checkpoint operation.

In Figure 3(a) we look at the effect of checkpointing the NAS Parallel Benchmark LU Class C with 32 processes. The size of the checkpoint is 1 GB or about 32 MB per process. For a single checkpoint, Figure 3(a) indicates that there is an overhead of 17% when checkpointing to NFS and 3% when checkpointing to local disk.

For 4 checkpoints there is an overhead of 84% and 6% respectively highlighting the importance of the file system. The checkpoint frequency, or time between checkpoints, for an HPC application is typically measured in hours, in these experiments we are forced to checkpoint more frequently due to the limited runtime of these applications.

In Figure 3(b) we look at the effect of checkpointing the NAS Parallel Benchmark EP Class D with 32 processes. This benchmark creates a checkpoint of 102 MB, or about 3.2 MB per process. Figure 3(b) shows us that the checkpoint overhead is almost negligible. This is due to the infrequent communication pattern and small memory footprint of the benchmark.

In Figure 3(c) we look at the effect of checkpointing the NAS Parallel Benchmark BT Class C with 36 processes. The size of the checkpoint is 4.2 GB or about 120 MB per process. For a single checkpoint, Figure 3(c) indicates an 18% overhead on NFS and 3% overhead on local disk. For 4 checkpoints there is a 74% and 8% overhead respectively.

In Figure 3(d) we look at the effect of checkpointing the NAS Parallel Benchmark SP Class C with 36 processes. The size of the checkpoint is 1.9 GB or about 54 MB per process. For a single checkpoint, Figure 3(d) indicates a 17% overhead on NFS and 3% overhead on local disk. For 4 checkpoints there is a 69% and 6% overhead respectively.

Next, we assess the performance impact of checkpointing GRO-MACS with the DPPC benchmark running with 8 and 16 processes. This benchmark creates a checkpoint of 267MB for 8 processes, or about 33MB per process. For 16 processes the checkpoint is 473MB or 30MB per process. The overhead of adding the CRCP layer is negligible, adding at most 1 second to the application runtime for both 8 and 16 processes. Figure 4 shows that the performance impact of checkpointing with between 1 and 4 checkpoints spaced evenly throughout the execution. The performance impact grows with each checkpoint, but is relatively small for any single checkpoint.

## 4.3 Checkpoint Overhead Analysis

Analyzing the impact of checkpointing on HPC application is important. It is equally important that we attempt to dissect the checkpoint overhead to determine where the checkpoint is spending the most time. In this analysis we split the pre-checkpoint phase into two phases: CRCP Protocol and Suspend BTLs. The former is the checkpoint/restart coordination protocol and the latter is the time spent suspending interconnect drivers. We expect the time to handle Ethernet and shared memory networks to be near 0 since we do not need to take action during the pre-checkpoint phase. Since InfiniBand and Myrinet each have to tear down their network connections, they each are required to spend some time during this phase. The CRCP Protocol time can vary slightly depending on when processes enter into the coordination algorithm and if the process is forced to wait on messages to drain from the network. To control for process skew, in this particular analysis, we introduced a barrier between each of the individual operations highlighted in this analysis.

The checkpoint operation is the time it takes BLCR to save the process image to stable storage. During the continue phase we may have to rebuild the PML by re-exchanging the `modex`, which is an all-to-all collective operation involving all processes in the applica-

<b>lu.C.32</b>	<b>Ethernet &amp; Shmem</b>		<b>InfiniBand &amp; Shmem</b>		<b>Myrinet &amp; Shmem</b>	
CRCP Protocol	0.01 sec	(0.0%)	0.02 sec	(0.1%)	0.02 sec	(0.1%)
Suspend BTLs	0.02 sec	(0.1%)	0.04 sec	(0.2%)	0.02 sec	(0.1%)
Checkpoint	17.00 sec	(99.9%)	18.43 sec	(99.3%)	16.81 sec	(99.2%)
Rebuild PML	0.00 sec	(0%)	0.08 sec	(0.4%)	0.10 sec	(0.6%)
Total	17.03 sec		18.57 sec		16.95 sec	

**Table 2: Checkpoint overhead analysis for NAS Parallel Benchmark LU Class C with 32 processes. Global snapshot is 1GB or 32MB per process.**

<b>ep.D.32</b>	<b>Ethernet &amp; Shmem</b>		<b>InfiniBand &amp; Shmem</b>		<b>Myrinet &amp; Shmem</b>	
CRCP Protocol	0.10 sec	(8.8%)	0.09 sec	(3.4%)	0.09 sec	(5.4%)
Suspend BTLs	0.02 sec	(1.8%)	0.03 sec	(1.1%)	0.03 sec	(1.8%)
Checkpoint	1.02 sec	(89.5%)	2.04 sec	(76.1%)	1.03 sec	(61.3%)
Rebuild PML	0.00 sec	(0%)	0.52 sec	(19.4%)	0.53 sec	(31.5%)
Total	1.14 sec		2.68 sec		1.68 sec	

**Table 3: Checkpoint overhead analysis for NAS Parallel Benchmark EP Class D with 32 processes. Global snapshot is 102MB or 3.2MB per process.**

<b>bt.C.36</b>	<b>Ethernet &amp; Shmem</b>		<b>InfiniBand &amp; Shmem</b>		<b>Myrinet &amp; Shmem</b>	
CRCP Protocol	0.04 sec	(0.1%)	0.07 sec	(0.1%)	0.06 sec	(0.1%)
Suspend BTLs	0.02 sec	(0.0%)	0.08 sec	(0.1%)	0.03 sec	(0.0%)
Checkpoint	67.71 sec	(99.9%)	68.39 sec	(99.63%)	69.28 sec	(99.7%)
Rebuild PML	0.00 sec	(0%)	0.11 sec	(0.2%)	0.12 sec	(0.2%)
Total	67.76 sec		68.65 sec		69.49 sec	

**Table 4: Checkpoint overhead analysis for NAS Parallel Benchmark BT Class C with 36 processes. Global snapshot is 4.2GB or 120MB per process.**

<b>sp.C.36</b>	<b>Ethernet &amp; Shmem</b>		<b>InfiniBand &amp; Shmem</b>		<b>Myrinet &amp; Shmem</b>	
CRCP Protocol	0.02 sec	(0.1%)	0.03 sec	(0.1%)	0.12 sec	(0.4%)
Suspend BTLs	0.02 sec	(0.1%)	0.08 sec	(0.3%)	0.03 sec	(0.1%)
Checkpoint	29.76 sec	(98.9%)	33.02 sec	(99.4%)	32.19 sec	(99.2%)
Rebuild PML	0.00 sec	(0%)	0.09 sec	(0.3%)	0.12 sec	(0.4%)
Total	29.80 sec		33.21 sec		32.45 sec	

**Table 5: Checkpoint overhead analysis for NAS Parallel Benchmark SP Class C with 36 processes. Global snapshot is 1.9GB or 54MB per process.**

<b>GROMACS DPPC</b>	<b>Ethernet &amp; Shmem</b>		<b>InfiniBand &amp; Shmem</b>		<b>Myrinet &amp; Shmem</b>	
CRCP Protocol	0.01 sec	(0.2%)	0.02 sec	(0.4%)	0.04 sec	(0.8%)
Suspend BTLs	0.01 sec	(0.2%)	0.01 sec	(0.2%)	0.01 sec	(0.2%)
Checkpoint	4.76 sec	(99.6%)	5.24 sec	(98.4%)	4.95 sec	(97.8%)
Rebuild PML	0.00 sec	(0%)	0.05 sec	(1.0%)	0.06 sec	(1.2%)
Total	4.78 sec		5.32 sec		5.06 sec	

**Table 6: Checkpoint overhead analysis for GROMACS DPPC running with 8 processes. Global snapshot is 267MB or 33MB per process.**

<b>GROMACS DPPC</b>	<b>Ethernet &amp; Shmem</b>		<b>InfiniBand &amp; Shmem</b>		<b>Myrinet &amp; Shmem</b>	
CRCP Protocol	0.01 sec	(0.1%)	0.02 sec	(0.2%)	0.02 sec	(0.3%)
Suspend BTLs	0.03 sec	(0.4%)	0.04 sec	(0.5%)	0.02 sec	(0.3%)
Checkpoint	8.07 sec	(99.5%)	7.88 sec	(98.4%)	7.48 sec	(97.9%)
Rebuild PML	0.00 sec	(0%)	0.07 sec	(0.9%)	0.12 sec	(1.6%)
Total	8.11 sec		8.01 sec		7.64 sec	

**Table 7: Checkpoint overhead analysis for GROMACS DPPC running with 16 processes. Global snapshot is 473MB or 30MB per process.**

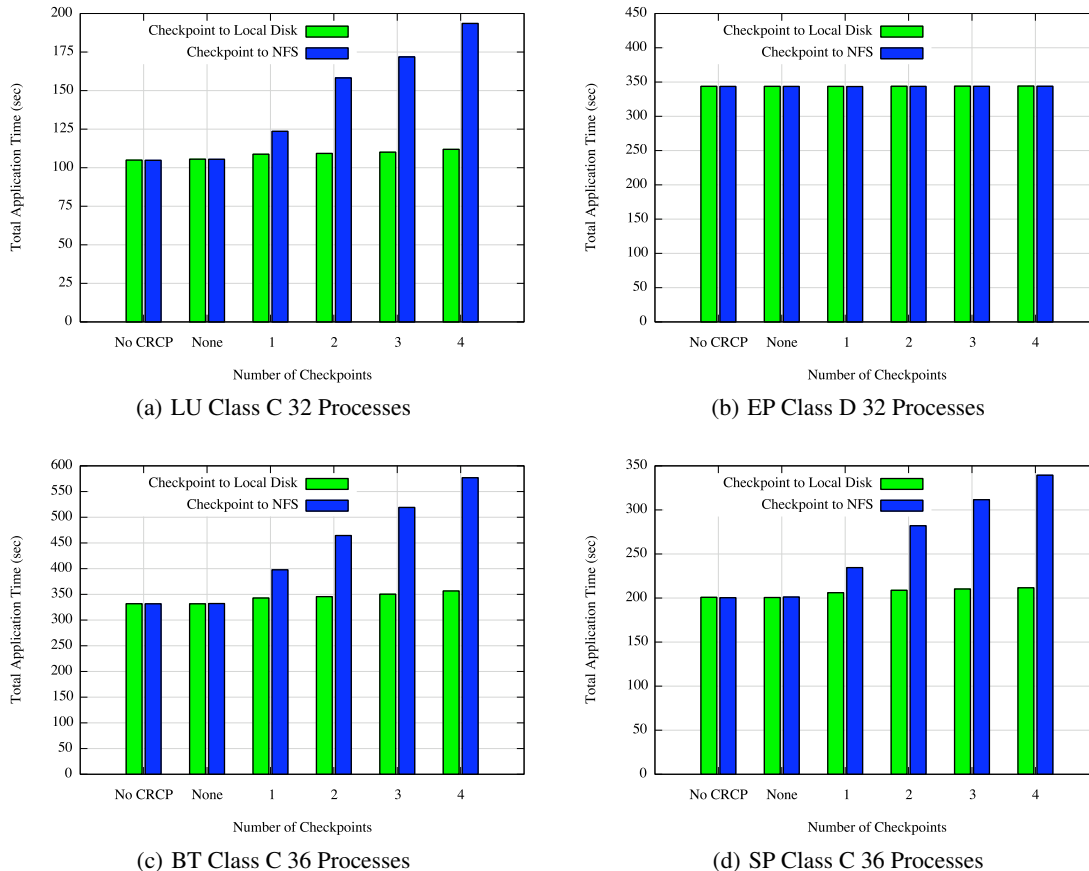


Figure 3: Performance impact of checkpointing NAS Parallel Benchmarks.

tion.

In Table 2 we look at the overhead involved when checkpointing the LU Class C NAS parallel benchmark. In Table 3 we look at the overhead involved for the EP Class D NAS parallel benchmark. In Table 4 and Table 5 we look at the overhead involved when checkpointing the BT and SP Class C NAS parallel benchmarks respectively. For all of these benchmarks we can see that the time to create the checkpoint and save it to the central stable storage device dominates the checkpoint time. This is closely followed by the time needed to re-exchange the `modex` during the continue phase. The overhead of the checkpoint/restart coordination algorithm is a factor, but not quite as severe as the time spent in the checkpoint and `modex` operations.

In Tables 6 and 7 we analyze the overhead involved when checkpointing the GROMACS DPPC benchmark with 8 and 16 processes respectively. This data confirms what was seen with the NAS benchmarks, most notably that the time taken by the checkpoint/restart coordination protocol is overshadowed by the time needed to take the single process checkpoint. In future work, we plan on exploring techniques for addressing each of these identified bottlenecks.

## 5. CONCLUSIONS

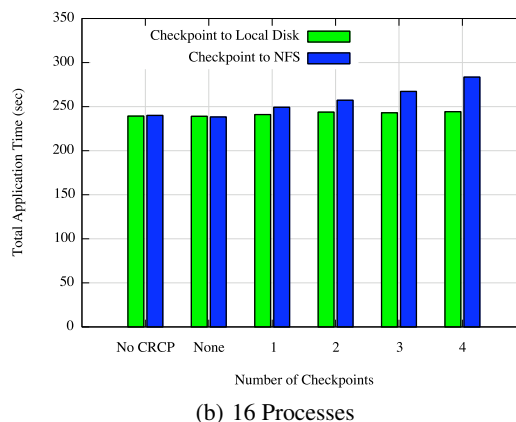
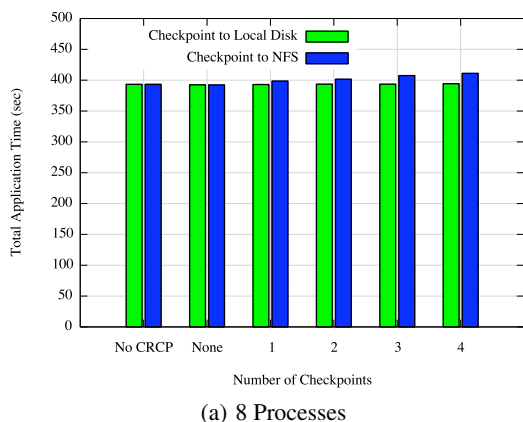
In addition to performance at scale, HPC applications are being required to consider fault tolerance when designing their applica-

tion to run on modern HPC systems. Among the fault tolerance techniques available, transparent checkpoint/restart techniques are a popular option to application developers. This option does not require any code modifications in the application in order to take advantage of the fault tolerance capability. Unfortunately many of the transparent checkpoint/restart implementations restrict the types of interconnects that could be used to support MPI communication between processes, including shared memory communication.

In this paper we presented an approach to implementing transparent checkpoint/restart in an MPI implementation that does not restrict the application's network choices. Our approach allows for a process to take full advantage of the best communication medium between any two processes in their MPI application. We discussed support for Ethernet, shared memory, Myrinet, and InfiniBand interconnects. Additionally, we demonstrated an application moving between various network topologies and taking full advantage of the interconnects available upon restart. Our approach enables an HPC application to adapt to changes in the process layout and network availability in such a way that they can achieve high performance in a wider variety of restart scenarios.

## 6. FUTURE WORK

High speed interconnects, such as InfiniBand and Myrinet, require MPI implementations to tear down the network connections during pre-checkpoint and rebuild them during the continue phase which is an expensive operation. Moving forward with this work we plan on



**Figure 4: Performance impact of checkpointing 8 and 16 processes with GROMACS DPPC.**

investigating interconnect driver and BLCR improvements directed at alleviating the need to tear down these connections.

Additionally, we plan to investigate alternative system-level checkpoint/restart services, and coordination protocols. Adding support for other system-level checkpoint/restart supports allows us to work on a wider variety of HPC installations. We will also explore alternative checkpoint/restart coordination algorithms that may allow for looser synchrony between processes. Open MPI's MCA architecture provides us with a unique opportunity to compare these protocols in a rich and controlled manner at runtime.

The stress of checkpointing on the central stable storage device is considerable. We are investigating alternative global coordination and file movement techniques to alleviate some of this stress.

## Acknowledgments

This work was supported by a grant from the Lilly Endowment and National Science Foundation grants NSF ANI-0330620, CDA-0116050, and EIA-0202048.

## 7. REFERENCES

- [1] A. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. In *HPDC '99: Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing*, page 31, Washington, DC, USA, 1999. IEEE Computer Society.
- [2] A. Avizenis. Fault-tolerance and fault-intolerance: Complementary approaches to reliable computing. In *Proceedings of the international conference on Reliable software*, pages 458–464, New York, NY, USA, 1975. ACM Press.
- [3] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello. MPICH-V project: A multiprotocol automatic fault-tolerant MPI. In *International Journal of High performance Computing Applications*, volume 20, pages 319–333. Sage Publications, Inc., 2006.
- [4] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated application-level checkpointing of MPI programs. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 84–94, New York, NY, USA, 2003. ACM Press.
- [5] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.
- [6] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.
- [7] C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 127, New York, NY, USA, 2006. ACM.
- [8] J. Duell, P. Hargrove, and E. Roman. The design and implementation of Berkeley Lab's linux checkpoint/restart. Technical Report LBNL-54941, Lawrence Berkeley National Lab, 2003.
- [9] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.
- [10] Future Technologies Group. Berkeley Lab Checkpoint/Restart (BLCR). <http://ftg.lbl.gov/checkpoint/>.
- [11] Q. Gao, W. Huang, M. Koop, and D. Panda. Group-based coordinated checkpointing for MPI: A case study on InfiniBand. In *ICPP'06: Proceedings of the 35th International Conference on Parallel Processing*, Jan 2007.
- [12] Q. Gao, W. Yu, W. Huang, and D. K. Panda. Application-transparent checkpoint/restart for MPI programs over InfiniBand. *Parallel Processing*, Jan 2006.
- [13] E. Garbriel, G. Fagg, G. Bosilica, T. Angskun, J. J. Dongarra, J. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. Castain, D. Daniel, R. Graham, and T. Woodall. Open MPI: goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, 2004.
- [14] C. Huang. System support for checkpoint and restart of Charm++ and AMPI applications. Master's thesis, Dept. of Computer Science, University of Illinois, 2004.
- [15] C. Huang, G. Zheng, and L. V. Kalé. Supporting adaptivity in MPI for dynamic parallel applications. Technical Report 07-08, Parallel Programming Laboratory, Department of

Computer Science, University of Illinois at Urbana-Champaign, 2007.

- [16] C. Huang, G. Zheng, S. Kumar, and L. V. Kalé. Performance evaluation of Adaptive MPI. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, March 2006.
- [17] 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 *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, March 2007.
- [18] InfiniBand Trade Association. InfiniBand. <http://www.infinibandta.org>.
- [19] D. P. Jasper. A discussion of checkpoint/restart. *Software Age*, pages 9–14, October 1969.
- [20] 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 ( $m^3$ ). *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, Nov 2005.
- [21] P. Lemarinier, A. Bouteiller, T. Herault, G. Krawezik, and F. Cappello. Improved message logging versus improved coordinated checkpointing for fault tolerant MPI. In *CLUSTER '04: Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 115–124, Washington, DC, USA, 2004. IEEE Computer Society.
- [22] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proc. of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, November 1993.
- [23] National Aeronautics and Space Administration. NAS parallel benchmarks. <http://www.nas.nasa.gov/Resources/Software/npb.html>.
- [24] OpenFabrics Alliance. OpenFabrics. <http://www.openfabrics.org/>.
- [25] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. Technical report, Knoxville, TN, USA, 1994.
- [26] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, New York, NY, USA, 1975. ACM Press.
- [27] J. Ruscio, M. Heffner, and S. Varadarajan. DejaVu: Transparent user-level checkpointing, migration and recovery for distributed systems. *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, Nov 2006.
- [28] S. Sankaran, J. M. Squyres, B. Barrett, and A. Lumsdaine. Checkpoint-restart support system services interface (SSI) modules for LAM/MPI. Technical Report TR578, Indiana University, Computer Science Department, 2003.
- [29] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. *International Journal of High Performance Computing Applications*, 19(4):479–493, Winter 2005.
- [30] C. Sosa. IBM system Blue Gene solution: Blue Gene/P application development. Technical report, IBM, September 2008.
- [31] J. M. Squyres and A. Lumsdaine. The component architecture of Open MPI: Enabling third-party collective algorithms. In *Proceedings of 18th ACM International Conference on Supercomputing, Workshop on Component Models and Systems for Grid Applications*, pages 167–185, St. Malo, France, July 2004. Springer.
- [32] G. Stellner. CoCheck: Checkpointing and process migration for MPI. In *10th International Parallel Processing Symposium (IPPS '96)*, page 526, 1996.
- [33] N. H. Vaidya. A case for two-level distributed recovery schemes. In *SIGMETRICS '95/PERFORMANCE '95: Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pages 64–73, 1995.
- [34] N. H. Vaidya. Staggered consistent checkpointing. *IEEE Trans. Parallel Distrib. Syst.*, 10(7):694–702, 1999.
- [35] D. Van Der Spoel, E. Lindahl, B. Hess, G. Groenhof, A. E. Mark, and H. J. Berendsen. GROMACS: Fast, flexible, and free. *Journal of Computational Chemistry*, 26(16):1701–1718, 2005.
- [36] G. Zheng, C. Huang, and L. V. Kalé. Performance evaluation of automatic checkpoint-based fault tolerance for AMPI and Charm++. *ACM SIGOPS Operating Systems Review: Operating and Runtime Systems for High-end Computing Systems*, 40(2), April 2006.