Athanasia: A User-Transparent and Fault-Tolerant System for Parallel Applications

Hyunsoo Jung, Hyuck Han, Heon Y. Yeom, and Sooyong Kang

Abstract—This article presents Athanasia, a user-transparent and fault-tolerant system, for parallel applications running on large-scale cluster systems. Cluster systems have been regarded as a de facto standard to achieve multi-tera-flop computing power. These cluster systems, as we know, have an inherent failure factor that can cause computation failure. The reliability issue in parallel computing systems, therefore, has been studied for a relatively long time in the literature, and we have seen many theoretical promises arise from the extensive research. However, despite the rigorous studies, practical and easily deployable fault-tolerant systems have not been successfully adopted commercially. Athanasia is a user-transparent checkpointing system for a fault-tolerant Message Passing Interface (MPI) implementation that is primarily based on the sync-and-stop protocol. Athanasia supports three critical functionalities that are necessary for fault tolerance: a light-weight failure detection mechanism, dynamic process management that includes process migration, and a consistent checkpoint and recovery mechanism. The main features of Athanasia are that it does not require any modifications to the application code and that it preserves many of the high performance characteristics of high-speed networks. Experimental results show that Athanasia can be a good candidate for practically deployable fault-tolerant systems in very-large and high-performance clusters and that its protocol can be applied to a variety of parallel communication libraries easily.

Index Terms—User Transparency, Fault Tolerance, Message Passing Interface, Parallel Systems, Myrinet, InfiniBand, ch_p4

1 Introduction

Parallel programming environments have been developed rapidly with the advent of very high speed networks such as Myrinet and InfiniBand. Unfortunately, distributed systems, such as Myrinet cluster systems, are not reliable enough to guarantee the completion of parallel processes within a specific time frame due to their intrinsic failure factors. Even a single local failure can be fatal to parallel processes, since it could render useless all computations executed to the point of failure. Therefore, it is not difficult to see that fault tolerance is an essential requirement of distributed systems, where the possibility of failure increases as the complexity of the system grows. The traditional methods of building fault-tolerant systems, such as checkpointing and message logging, have been studied and attempts have been made to exploit these for practical use in distributed systems. However, there are few practical fault-tolerant systems except BLCR [1] which has been recently integrated with Open MPI.

It is well recognized that the area of distributed systems is a noticeable area among numerous computer science fields, where we can be aware of a discernible gap between theory and practice. Even a well-known theory in this field is sometimes hard to materialize into practical systems (e.g., Byzantine Fault Tolerance (BFT) [2] and Paxos [3].) The main motivation of our work is on the basis of practicality that can be readily used in real systems. This article presents Athanasia, a practical, multiple fault-tolerant system for MPI applications on three high performance cluster systems: MPICH-GM (Myrinet), MVAPICH (InfiniBand), and MPICH-P4 (ch_p4). We generalized the fault tolerance framework of the earlier version of this article [4]. Athanasia supports three critical functionalities that are necessary for fault tolerance: (a) a light-weight failure detection mechanism, (b) dynamic process management including process migration, and (c) a consistent checkpoint and recovery mechanism. To ease end-user operations, Athanasia is built to be a user-transparent checkpointing system. In other words, application programs do not need to be rewritten to be equipped with fault tolerance. To achieve characteristics (a) and (b), we built our own job submission and process management system to control Athanasia and to deal with multiple failures gracefully. For the feature (c), we implemented a sync-and-stop protocol [5] because it is the simplest version of a blocking coordinated checkpointing protocol [6] to obtain a distributed consistent snapshot in real environment.

From the viewpoint of practicality, implementation issues depicted in later sections reveal measures taken to overcome many obstacles that theoretical algorithms do not take into account. In exploring the question of how to create the protocol in an efficient manner, the sync-and-stop protocol discussed in this article will be limited to the consideration of robust creation of a theoretical protocol.

One of the main implementation issues in building Athanasia is realizing dynamic process management, which is not specified in the original MPI standard (version 1). In order to allow a new MPI instance to communicate with restarted processes, we designed a rejoin mechanism to enable restarted MPI processes to rebuild communication channels and to reorganize all communication sessions with other cooperating processes. This is one of the key protocols required in the consistent recovery mechanism of communicating processes.

The rest of the article is organized as follows. Section 2 gives a detailed explanation of the architecture of Athanasia. Section 3 presents important implementation issues in building Athanasia, and Section 4 describes experimental results. Finally, we reach our conclusions in Section 5.
1.1 Related Work

Our previous work, MPICH-GF [7], [8], enabled MPICH-G2 to be fault-tolerant. MPICH-GF supports both, the coordinated checkpointing protocol and various message logging protocol schemes, although it is not resilient to multiple failures. We implemented fault tolerance at the abstract device interface of MPICH-G2, which is based on standard TCP/IP.

MPICH-V [9] is a fault-tolerant MPICH that supports pessimistic message logging, where all the messages are transferred to the remote Channel Memory (CM) servers that log and replay them. The pessimistic logging protocol has a higher performance overhead than a checkpoint-based protocol, especially when processes of the message passing system transmit numerous messages during failure-free operations [10]. In [11], [12], the authors showed that a coordinated checkpointing protocol outperforms a pessimistic logging protocol during failure-free operations.

The main motivation for MPICH-V2 [13] is to remove the need of CMs to reduce the overhead and cost of fault tolerance. The MPICH-V2 protocol is a pessimistic sender-based one: it keeps a copy of each message payload at the sender side (potentially volatile), and logs some causality information on a reliable medium. MPICH-Vcausal [14] adopts a causal message logging scheme to alleviate the high message latency problem of MPICH-V2. MPICH-Vcausal stores causality information asynchronously into a reliable storage device, which reduces the MPI message latency. However, MPICH-Vcausal still has message logging overhead, which makes a coordinated checkpointing scheme outperform a message logging scheme as Mean Time Between Failures (MTBF) decreases.

Unlike MPICH-V2/Vcausal, MPICH-PCL [15] is a new implementation of a blocking-checkpointing mechanism for fault tolerance inside MPICH2. This for implementation, a new channel (ft-sock) that is based on the TCP sock channel is introduced to support blocking-checkpointing. Recently, Berkeley Lab Checkpoint/Restart (BLCR) [1] is integrated into LAM/MPI, MVAPICH2 and Open MPI. In LAM/MPI with BLCR, out-of-band communication channels in LAM are used to clear MPI communication channel, which is necessary to a blocking-checkpointing mechanism.

W. Huang et al. introduced virtual machine (VM) migration over InfiniBand with MVAPICH for fault tolerance and proposed a new VM-aware MPI library, called MVAPICH2-ivc [16], [17]. MVAPICH2-ivc supports efficient VM-aware communication for shared memory communication in different VMs on the same physical host.

FT-MPI [18] proposed by Fagg and Dongarra, supports MPI-2’s dynamic task management that is based on PVM or HARNESS daemons. Its main drawback is that when a new communication device interface needs to be supported, an entire operating system module for fault tolerance as well as the management system should be modified, while application-level fault-tolerant systems are required to modify the management system only. In Li and Tsay’s LAM-MPI-based implementation, messages are transferred by way of a multicast server on each node [19]. MPI-FT from Cyprus University [20] adopts message logging. An observer processor copies all the messages and reproduces them on recovery, which requires a high storage cost for all the messages. MPI/FT, proposed by Batchu et al. [21], adopts task redundancy for fault tolerance. It has a centralized coordinator that relays messages to all redundant processes. In summary, the method of relaying a message for communication usually suffers a relatively long communication delay (i.e., low performance), which is avoidable.

Legion MPI-FT [22] is the first effort to build a fault-tolerant system on the grid, and is based on the LAM-MPI with a daemon mode. It supports the coordinated checkpointing protocol, where processes interchange complex control messages to ensure the absence of in-transit messages.

Since the increase in message delay is inevitable with the indirect message transfer mode, we have approached the direct message transfer mode in favor of the performance issue. There are some previous works that support the direct message transfer mode. CoCheck [23] is a thin library of programming models that supports the coordinated checkpointing protocol for PVM and tuMPI. While CoCheck for tuMPI supports the direct transfer mode, the PVM version exploits the PVM daemon to transfer messages. Starfish [24] is a heterogeneous checkpointing toolkit based on the Java virtual machine, which makes it possible for the processes to migrate among heterogeneous platforms. The limits of this system are that they have to be written in OCaml and that byte codes run more slowly than native codes. Egidia [25] is an object-oriented toolkit that employs both communication-induced checkpointing and message logging for MPICH. Hector [26] exists as a movable MPI library and several executables. It supports coordinated checkpointing. Before checkpointing, every process closes its channel connection to ensure that there are no in-transit messages left in the network. Processes have to reconstruct the channel after checkpointing. RENEW [27], proposed by Neves et al., has a user-level reliable communication layer built upon the UDP layer, in order to log messages as well as to prevent the loss of in-transit messages. This technique requires additional memory copies at the sender side. RROMP (Rollback Restart for MPI) with C³ (Cornell Checkpoint Compiler) [6] adopts an application-level checkpointing scheme to achieve a consistent coordinated checkpointing.

A compiler-enhanced incremental checkpointing technique is proposed to support OpenMP applications [28]. With the help of compiler analysis, the incremental checkpointing reduced checkpoint size and made asynchronous checkpointing possible. For more general purpose checkpoint/restart, Ansel et al. proposed DMTCP, a user transparent checkpointing for cluster computations and the desktop [29]. DMTCP provides more versatility to a variety of applications on cluster environments, and it aims to support RDMA interfaces (i.e., Infiniband and Myrinet), which Athanasia hereby claims to achieve the goal on the designated target platform (MPICH variants). A hybrid approach combining coordinated checkpointing and message logging has been researched recently using LAM/MPI and MPICH-VCL [30]. They proposed group-based checkpoint/restart in order to enhance scalability by reducing coordination overhead.

2 THE ARCHITECTURE OF ATHANASIA

This section presents the architecture of Athanasia. Athanasia is based on a coordinated checkpointing protocol that involves the synchronization among MPI processes. In Athanasia, the process coordination is controlled by hierarchical process managers. We start this section with an overview of the management system that is designed to support fault tolerance. Then, we describe the implementation of the consistent coordinated checkpointing protocol adopted in Athanasia. Finally, we close the section with our implementation of failure detection and
recovery. We will describe the core design principles using the system built on the Myrinet interface, and customization for InfiniBand and ch_p4 interfaces will follow, if necessary. Note that the fault model used in this work is the fail-stop model of process and node crashes.

2.1 Management System

The management system was developed to manage and monitor multiple MPI processes running on multiple nodes and their execution environment. Its primary role is to coordinate global and consistent checkpoints and to help the system recover from multiple failures gracefully. At the same time, the management system must be lightweight so as not to interfere with the performance of MPI processes.

The main function of the management system is to detect failures and to support recovery from failures. To detect and recover from failures, a process, conventionally called a Local Job Manager (LocalJM), which is separated from the MPI process, must be executed to monitor the MPI processes, one for each MPI process. To monitor LocalJMs, a Leader Job Manager (LeaderJM) is executed. In $M^3$ [4], while LocalJM failures could be recovered from, LeaderJM (i.e., Central Manager) failures could cause the system to crash. To overcome this problem, we incorporate a leader election algorithm into the recovery procedure. All LocalJMs periodically send heartbeats to the LeaderJM, and the LeaderJM responds to each heartbeat. When a LocalJM does not receive a response from the LeaderJM, it suspects that the LeaderJM has failed. If a consensus is reached among LocalJMs that the LeaderJM has failed, the LocalJM with the highest rank is chosen to be the temporary leader using the bully algorithm [31]. The temporary leader restarts the LeaderJM on a remote unused node. The new LeaderJM coordinates LocalJMs in order to restart all the MPI processes based on the last version of checkpoints.

The purpose of the LeaderJM is to manage all system functions and maintain the system state. It receives input from the user and schedules the task with the help of a third-party scheduler. Users can send input to the LeaderJM by way of a simple client program. Users must provide the client program with basic information, such as the address and port number of the LeaderJM and the program to be executed. Once the task has been assigned, the LeaderJM coordinates checkpointing by periodically issuing checkpoint commands. The LeaderJM uses a periodic heartbeat to detect node failures. When failures occur, the LeaderJM is responsible for handling the failures and assisting in recovery. The failure detection and recovery procedures are explained in detail in Section 2.3.

The management system of Athanasia can also support a job suspend/resume mechanism like BLCR. Most cluster systems use third-party schedulers such as PBS, LSF, and LoadLeveler, which require users to submit a job description file with the number of required CPUs, the name of the executable file, and the execution time to the scheduler. Wrong estimation of the execution time can result in loss of computation or computing resources if the scheduler aborts the job before it finishes or allows a faulty process to execute for an inordinate amount of time. To account for these cases, our system allows jobs to be suspended. When the LeaderJM receives a special signal such as `SIGTERM` from the scheduler to suspend all participants, it orders all participants to take a checkpoint and to terminate themselves. Afterwards, the user or system administrator can resume suspended jobs, if necessary, from the last checkpoint.

This mechanism greatly alleviates problems, such as low utilization and high completion time, which is brought about by users’ wrong estimation of the execution time.

2.2 Checkpointing Protocol in Athanasia

This section discusses how to obtain a consistent snapshot of communicating processes. To design a robust and consistent distributed checkpoint protocol of parallel processes running on cluster systems, we can choose one of two widely used protocols: blocking coordinated checkpointing and message logging with checkpointing protocols. Our previous work [7], [8] shows that message logging cannot outperform blocking coordinated checkpointing, especially when parallel processes exchange a great number of messages with each other. In Athanasia, we adopt a `sync-and-stop` protocol [5], the simplest version of a blocking coordinated checkpointing, because Myrinet or InfiniBand cluster systems are likely to execute parallel applications that frequently exchange large messages with each other. The `sync-and-stop` protocol used in Athanasia enforces a distributed snapshot to be taken after all the processes have been synchronized and have stopped their execution. Figure 1 depicts Athanasia’s `sync-and-stop` protocol. The protocol requires processes to orchestrate their checkpoints in order to form a consistent global state. It simplifies a recovery procedure and is not susceptible to the domino effect, since every process always restarts from its most recent checkpoint; this is the main reason for adopting the `sync-and-stop` protocol in Athanasia. Athanasia’s `sync-and-stop` protocol requires each process to maintain only one permanent checkpoint on stable storage, reducing storage overheads and eliminating the need for garbage collection. As a matter of fact, however, the robust implementation of the `sync-and-stop` protocol requires each process to maintain two versions of checkpoint images because a checkpoint period itself is also susceptible to failures. By retaining two versions of checkpoint images, Athanasia can cover the entire lifetime of all processes safely.

2.2.1 State transition diagram of the checkpointing protocol

The primary strategy we adopt to obtain a distributed snapshot is the `sync-and-stop` protocol. The correctness of the `sync-and-stop` protocol is asserted when there is no `z-dependency` between any two communicating processes in different checkpoint intervals. Basically, the `z-dependency` occurs when there is any message that establishes a causal relationship between two different checkpoint intervals. To preserve the correctness and remove such an improper message, we need to define a notion of “message atomicity” and “no pending

![Fig. 1. Blocking Coordinated Checkpointing in Athanasia](image-url)
communication”. We do this because it is non-trivial to eliminate the z-dependency in realistic communications. We hereby define “message atomicity” and “no pending communication” as follows: the message atomicity means there is no in-transit message in any communication channel, while no pending communication means that there is no process waiting for completion of communication. By taking a closer look at the mechanism of MPICH-GM and a concrete implementation of well-categorized communication types, we could design a state machine to capture consistent snapshots for communicating processes. The state machine can be easily applicable to other types of communication libraries, such as MVAPICH and MPICH-P4, on which the implementations of Athanasia on the InfiniBand and ch_p4 interfaces are based, respectively.

Figure 2 shows the state transition diagram of Athanasia’s checkpointing protocol. We divided all the states of processes into seven distinguished states: the three states in the lower half are not checkpoint-related and the other four states are checkpoint-related. In the diagram, the transition edge has two trigger events, which causes the process to transit to another state. The first is categorized as a normal execution flow. Various points of execution paths in the communication operations are depicted in the figure. The second is classified as a checkpoint-related flow. We distinguish both types of events with a slash (“/”).

The critical section state indicates that the process is currently executing communication operations (i.e., sending/receiving a message through the RDMA transfer). In this region, the process cannot be checkpointed due to the necessity of guaranteeing the atomicity of communication operations (i.e., the message can have only one of two distinct states; “sent/received” or “not sent/not received”).

The non-critical section state indicates that the current process can be checkpointed safely without considering channel consistency. A process is in a non-critical section state if it is in a userland computation region or in a checkpoint-safe region of communication operations. If the process receives a checkpoint signal, it can directly enter a checkpoint procedure. The remaining state transition from the non-critical section state is straightforward.

The non-critical, checkpoint-unsafe (NCCU) state is the state that a process is currently involved in when exchanging control messages (non-critical), but it is in the middle of user data transmission (checkpoint-unsafe). This state is created because of MPICH-GM’s Rendezvous operations (Rnv_{(j)}Send/_{(j)}Recv) for the efficient transmission of large data. Figure 3 shows the NCCU state in the Rendezvous communication. When Rendezvous communication is involved, a message is transmitted through the pre-handshaking procedure (i.e., by using an OR_TO_SEND message). Once the pre-handshaking procedure ends, the message is generally fragmented into several chunks of data, and then each chunk can be transmitted in two stages: control message transmission and chunk data transmission. The control message transmission is indeed non-critical, but the chunk data transmission is critical. The Rendezvous operation may have several chunks, which means that the control message transmission is not checkpoint-safe any more due to the atomicity property.

With these seven states, to obey the atomicity property, upon receiving a checkpoint signal while executing Rendezvous communication, the process should perform an appropriate action according to its current state: If the process was in either the NCCU state or the critical section state, the signal handler delays checkpointing until it transits to the non-critical section state. In other words, a decision is made as to whether the process should delay checkpointing until the Rendezvous communication finishes. The message atomicity region is clearly indicated in Figure 3. From the viewpoint of the message atomicity, the safe time to checkpoint is before sending/receiving a message or after sending/receiving a message, however, not in the middle of sending/receiving a message.

Two-phase checkpointing protocol. To ensure safe checkpointing, we devised a simple solution, a two-phase checkpointing protocol. The first phase ensures that there are no in-transit messages in a communication channel. In the first phase, all processes need to flush on-the-fly messages by barrier synchronization. After finishing the barrier message communication, the FIFO message delivery guaranteed by modern communication protocols ensures two properties: there are no orphan messages between any two processes and there are no in-transit messages because barrier messages flush messages that were previously issued to the receiver.

In the second phase, processes can safely perform checkpointing. Each process generates a checkpoint file and informs the LocalJM of the successful checkpoint. This is what we showed in Figure 1. Finally, the LeaderJM checks whether all checkpoint files have been generated. If so, it confirms the completion of the checkpoint by increasing the version of the global checkpoint.

In summary, we derive three properties from the state transition diagram of Athanasia.

Property 1: Communication of Athanasia is based on the FIFO channel.

Property 2: Athanasia’s checkpointing protocol ensures the message atomicity.

Property 3: Athanasia’s checkpointing protocol terminates within a finite time if no fault occurs during checkpointing.

Property 3 needs a detailed explanation; the state transition diagram of Figure 2 indicates that checkpointing terminates after channel commit, and channel commit messages similar to marker messages in [33] are transferred within a finite time under the no network failure condition.

2.2.2 Customization to InfiniBand and ch_p4 devices

MVAPICH has three communication patterns, depending on the size of the messages. Where the InfiniBand implementation differs from the Myrinet implementation is in how it handles the transfer of medium and large messages. Figures 3(b)-(c) show the Rendezvous communication protocol for medium (between 32 KB and 128 KB) and large (≥ 128 KB) messages in MVAPICH. For medium and large messages, control messages,
such as RNDZ\_START, FIN, Ready-To-Send (RTS), Clear-To-Send (CTS), and RNDZ\_SEND, are transferred in the same way small messages are sent, but transmission of the payload is initiated by the receiver using RDMA Read operations. While Myrinet supports only an RDMA Write operation, InfiniBand supports both RDMA Write and Read operations, which means that a receiver must know the virtual memory address of the sender and the remote key for resolving the address for RDMA Read operations.

After the sender registers buffers as DMAable memory, RNDZ\_START for medium messages and RNDZ\_SEND for large messages are transmitted to the receiver with the virtual address and the remote key for resolving the mapping between the virtual and the physical addresses. The virtual address and the remote key become invalid and unrecoverable after recovery from failure. In consequence, critical sections in medium and large messages should contain the transmission of RNDZ\_START, RNDZ\_SEND, and FIN messages as well as the payload. In Myrinet, both Eager communication (for small messages) and Rendezvous communication (for large messages) use RDMA Write operations, which necessitates a new state (Non-Critical Checkpoint-Unsafe) to discriminate between Eager and Rendezvous communications. In InfiniBand, the Rendezvous communication (for medium and large messages) uses the RDMA Read operation, as described earlier, while the Eager communication (for small message) uses the RDMA Write operation. Those operation types can naturally discriminate between two communication protocols, making it unnecessary to use the NCCU state in InfiniBand.

When all MPI processes have confirmed that they are not in their critical sections, they send small barrier messages to other processes. The receipt of barrier messages means that no in-transit messages exist in the communication channel. After receiving barrier messages from all other MPI processes, each MPI process can take a checkpoint locally, and this checkpoint is consistent. For channels built on the InfiniBand interface, InfiniBand can preserve the FIFO property and can guarantee the properties above.

MPICH-GM and MVAPICH fragment a large message into smaller segments and perform an RDMA operation for data transmission, MPICH-P4 does not fragment a large message; rather, it sends (or receives) it in a single transmission. This entails a slight change in the state transition diagram; in MPICH-P4, like in InfiniBand, the NCCU state is unnecessary. This indeed eases our work to adapt the main checkpoint protocol, since we could merge the NCCU state into the Non-Critical Section state. We therefore could adapt the main checkpoint protocol, which is specialized to Eager and Rendezvous operations, to MPICH-P4 without difficulty.

2.2.3 Correctness proof
In Athanasia, a session of N-parallel MPI tasks consists of one LeaderJM, N LocalJMs, and N MPI processes. For the global checkpointing, the LeaderJM initiates the checkpointing, and the checkpoint signal is delivered to all MPI processes through LocalJMs. Only MPI processes follow the distributed checkpointing protocol stated in Section 2.2.1 and perform checkpointing. Let $P_1, P_2, ..., P_N$ denote N MPI processes. The $i$th ($i \geq 0$) checkpoint interval of a process covers all computation between the $i$th and $(i+1)$th checkpoint. To prove the correctness of our protocol, we first show that in Athanasia’s checkpointing protocol there does not exist z-dependency [32] between any two communicating MPI processes.

**Definition 1:** (z-dependency [32]) If a process $P_p$ sends a message to a process $P_q$ during its $i$th checkpoint interval and $P_q$ receives the message during its $j$th checkpoint interval, then $P_q$ z-depends on $P_p$ during $P_p$’s $i$th checkpoint interval and $P_q$’s $j$th checkpoint interval.
Lemma 1: In Athanasia, there does not exist z-dependency between any two processes in different checkpoint intervals.

Proof: By contradiction. Assume that there exists z-dependency between two processes in different checkpoint intervals. The LeaderJM sends a checkpoint signal to all MPI processes in order to initiate a global checkpoint. There are two cases that the assumption satisfies.

Case 1: \( P_p \) sends at least one message to \( P_q \) before \( i \)th checkpoint of \( P_p \), and \( P_q \) receives it after \( i \)th checkpoint of \( P_p \) (missing message in Figure 4). If \( P_p \) sends one application message (\( m \)) to \( P_q \) before \( i \)th checkpoint of \( P_p \), \( P_q \) sends \( m \) to \( P_q \) before receiving the checkpoint signal. Then, \( P_q \) sends a barrier message to \( P_q \) and waits a barrier message from \( P_q \). When \( P_q \) receives the checkpoint signal, \( P_q \) sends a barrier message to \( P_p \) and waits a barrier message from \( P_p \). Owing to the FIFO property of the communication channel (Property 1 and 2), \( P_q \) receives \( m \) from \( P_p \) and stores the message to \( P_q \). Thus, if \( P_p \) sends \( m \) to \( P_q \) before \( i \)th checkpoint of \( P_p \), \( P_q \) also receives \( m \) before \( i \)th checkpoint of \( P_q \).

Case 2: \( P_p \) sends at least one message (\( m \)) to \( P_q \) after \( i \)th checkpoint of \( P_p \), and \( P_q \) receives it before \( i \)th checkpoint of \( P_q \) (orphan message in Figure 4). If \( P_p \) sends \( m \) to \( P_q \) after \( i \)th checkpoint of \( P_p \), \( P_p \) cannot send the message to \( P_q \) until it performs the \( i \)th checkpointing procedure. Thus, due to the barrier message exchange phase of our checkpointing protocol, \( P_q \) cannot receive \( m \) after the completion of the \( i \)th checkpoint.

We show that by the two cases the assumption is a contradiction. Then, we can derive that Athanasia’s checkpointing protocol is a consistent and distributed checkpointing protocol using Property 1, 2 and Lemma 1.

Theorem 1: Athanasia generates a consistent global checkpoint.

Proof: By contradiction. Assume that Athanasia generates an inconsistent global checkpoint. Then, there must be a pair of processes \( P_p \) and \( P_q \) such that \( P_p \) sends at least one message to \( P_q \) before \( i \)th checkpoint of \( P_p \), and \( P_q \) receives it after \( i \)th checkpoint of \( P_p \) (missing messages), or that \( P_q \) sends at least one message to \( P_q \) after \( i \)th checkpoint of \( P_q \), and \( P_q \) receives it before \( i \)th checkpoint of \( P_q \) (orphan messages) [34]. By Lemma 1 and Property 2, this is a contradiction.

Theorem 2: In Athanasia, the distributed checkpointing procedure eventually terminates even if failures occur during the checkpointing procedure.

Proof: If subsequent failures occur during checkpointing, Athanasia recovers them based on the last checkpoint and then tries to perform the checkpointing again later. If such an execution step repeats finite times, the last trial of the checkpointing terminates within finite time by Property 3.

2.3 Multiple Failure Detection and Recovery

2.3.1 Failure detection

Our system can handle three types of failures: node failures, process failures, and Job Manager failures. Node failures are caused by network partitions or node crashes (e.g., power failures, disk or memory errors). The LeaderJM checks for node failures by probing each node. If a node does not respond after one probe, which consists of three consecutive heartbeats, the LeaderJM considers that node as failed. Since our system may have to wait for one complete probe in the worst case, it cannot always immediately detect node failures. However, this waiting period is a necessary evil in eliminating false positives, and the duration of the period is at most five seconds. As mentioned in Section 2.1, heartbeat messages are transferred over the Ethernet network, not over the Myrinet or InfiniBand network where MPI processes communicate with each other, thereby reducing false positives caused by network congestion.

If the Ethernet network is extremely overloaded, heartbeat messages may be dropped, which may cause the LeaderJM to misdiagnose the node as failed. However, the chances of this occurring on high-performance cluster computing systems that are equipped with high bandwidth are very low.

MPI process failures can be detected by LocalJMs using the SIGCHLD signal, and LocalJM failures can be detected by the LeaderJM using TCP FIN packets. Both failures can be detected immediately. Currently, there is no way to keep MPI processes from being accidentally halted by an interrupt or trap in the user’s program. The LeaderJM failure can be detected by using both heartbeat messages (between LocalJMs and the LeaderJM) and the consensus on the LeaderJM failure, as was already discussed in Section 2.1.

2.3.2 Failure recovery

When the LeaderJM detects a failure, it starts to coordinate the recovery procedure. The LeaderJM enters a new epoch and notifies the change to all MPI processes for synchronization. The last checkpoint version is then delivered to all LocalJMs so that they can fetch the appropriate checkpoint image. The image is copied from the remote storage device if it does not exist locally. The MPI processes are then reincarnated by the LocalJMs. Finally, each MPI process sends its communication channel information to the LeaderJM, which aggregates the information and returns the aggregated information to all MPI processes. Through this procedure, the system can be restored back to a consistent state corresponding to the last checkpoint.

Per Algorithm 1, the epoch and event type determine the action taken by the LeaderJM as follows:

If the epoch of the event is not equal to the current epoch of the system (i.e., the event is from a previous epoch), the event is discarded. The reasons for discarding messages from previous epochs are described in section 2.3.4. Note that if the LeaderJM detects a node failure the LeaderJM spawns a new

Algorithm 1 Failure Recovery Procedure for the LeaderJM

```java
// evt is an event from a LocalJM or created by the LeaderJM
if (evt.epoch != global_system_epoch) {
    // discard;
else if (evt.type == Failure then
    global_system_state = FAILED;
    global_system_epoch++;
    Empty queue;
    Event newEvt = new Event;
    newEvt.system_state = global_system_state;
    newEvt.epoch = global_system_epoch;
    newEvt.ckptVersion = GetLastCheckpointVersion();
    BroadcastEvent(newEvt);
else if (evt.type == CommitInfo then
    queue(evt);
    if # of Commit_Info events in queue == # of processes then
        aggregate all Commit_Info events in queue;
        broadcast aggregated information to all members;
        clear all Commit_Info events in queue;
end if
else if (evt.type == SuccessfulRecovery then
    queue(evt);
    if # of SuccessfulRecovery events in queue == # of processes then
        aggregate all SuccessfulRecovery events in queue;
        broadcast aggregated information to all members;
        clear all SuccessfulRecovery events in queue;
    global_system_state = RUNNING;
end if
end if
```
LocalJM and creates a failure event.

If the event type is “Failed,” the global system state is set to “FAILED,” and the epoch is incremented by one. The LeaderJM then notifies LocalJMs of this change and the last checkpoint version. At this time, LocalJMs each process according to the version received from the LeaderJM. New processes then open a port and retrieve any information associated with channel reconstruction, such as the hostname and node ID. MPI processes then send this information to the LeaderJM. This is recognized by the LeaderJM as an event with type “Comm_Info.” If the event type is “Comm_Info,” the LeaderJM waits until all MPI processes have sent their communication information before aggregating the information and broadcasting it to all MPI processes. MPI processes, upon receipt of the aggregated communication information, reinitialize device information. This is explained in detail in the following paragraphs. If MPI processes succeed in reinitialization, they send a related notification with the event type “SuccessfulRecovery” to the LeaderJM. If the event type is “SuccessfulRecovery,” the LeaderJM waits until all MPI processes have sent a “SuccessfulRecovery” event before broadcasting to all MPI processes that the recovery procedure has completed successfully and that the system can use this information to recover. The exchange of communication information in the recovery process is similar to that of the MPI process initialization procedure. When the global system state is “FAILED,” other events except for “Comm_Info” and “SuccessfulRecovery” events are not executed for safe recovery.

Special care must be taken when recovering from failures. During recovery, processes that existed in the same node (for example, in the SMP machine) and communicated with each other using shared memory before the failure may be allocated in different nodes when they are reincarnated, which means that the device information saved in their checkpoint images is no longer valid. Therefore, after reincarnation, each process must reinitialize its device information. Since binding information between DMAable memory and the GM port is invalid after reincarnation, DMAable memory should be rebound to the GM Port.

Per Algorithm 2, the location of other processes and the existence of devices determine the action to take, as follows: if more than one process exists in the same node, we must determine whether an SMP device already exists. If an SMP device does not exist, the process must initialize the device before preparing the shared memory for communication. If a device already exists, we can prepare the shared memory without initialization. This means that the shared memory created using `mmap` in the previous epoch should be released and recreated based on the newly calculated size.

If there are other processes in remote nodes, each process must determine whether a Myrinet device exists. If a Myrinet device does not exist, the process must initialize a Myrinet device and allocate/bound DMAable memory. If a device does exist, DMAable memory must exist from a previous run, so the system can use this memory by rebinding it to the GM port.

When an LeaderJM fails, Athanasia elects the highest ranked LocalJM as a temporal leader, and the temporal leader spawns a new LeaderJM on an available node. The new LeaderJM reads a job specification which contains the last checkpoint version stored in the stable storage. The reincarnated LeaderJM now coordinates LocalJMs to restart all the MPI processes based on the last version of checkpoints. Even though this recovery procedure seems to be inefficient, it is indeed an effective solution because Athanasia does not need to maintain the LeaderJM state.

### 2.3.3 Customization to InfiniBand and ch_p4 devices

To provide fault tolerance for the InfiniBand interface, we used MVAPICH2. We divide the InfiniBand implementations into two versions: one that does not support shared memory-based process communication, and one that does. For the former version, the device reinitialization procedure differs from that for the Myrinet interface. Since MVAPICH2 does not support shared memory-based process communication, all processes (even in the same node in the SMP machine) communicate with each other using the InfiniBand interface. Hence, the device reinitialization procedure becomes very simple: (1) open a queue pair, (2) send communication information to the LeaderJM through the LocalJM, (3) wait for aggregated communication information, and (4) rebind existing DMAable memory to the queue pair. These procedures corresponds to lines 1, 2, 3, and 16 in Algorithm 2. In the latter version of the InfiniBand implementation, the device reinitialization procedure of the former version is used for establishing communication channels with remote processes. To set up communication channels with processes in the same node, the corresponding procedure in Algorithm 2 is reused.

The ch_p4 device, like the Myrinet interface, supports shared memory communication among processes on the same node. All procedures in Algorithm 2 are available for device reinitialization in MPICH-P4 by replacing the Myrinet specifics with those of ch_p4 (e.g., opening a new TCP/IP socket replaces the Myrinet specific work (lines 11-16)).

### 2.3.4 Multiple failures

Our work so far has dealt with recovering from a single failure. We define multiple failures as follows:

- The failure of an SMP machine, which crashes two processes at once.
- A failure that occurs while the system is recovering from a previous failure.

Our system does not have a state for multiple failures, in order to simplify system design. To defend against multiple failures:

1. MVAPICH2-0.60 or the former version does not support the shared memory-based process communication, while MVAPICH2-0.90 or the later version supports it.
while keeping the design simple, we define the following axioms.

**Axiom 1** Each LocalJM does not manage its own state nor does it maintain the global system state.

The global system state, maintained by the LeaderJM, reflects the state of MPI processes. The global system state is “RUNNING” only when all MPI processes are running normally, and “FAILED” if even one node has failed. If the LocalJM were to maintain its own state or the global system state, this information would have to be included in the checkpoint and used when recovering from LocalJM failures. The cost of this procedure far outweighs any possible benefits it may provide.

**Axiom 2** A message sent in one epoch is not valid in another.

The LeaderJM enters a new epoch every time a failure is detected. This epoch value is included in all messages between the LeaderJM and LocalJM and those between the LocalJM and MPI processes. All messages must have the same epoch value. Upon receiving a message, the receiver checks the epoch value to determine whether the message was sent in the current epoch. If the message was sent from another epoch, the message is discarded. These messages, with the exception of failure-notification messages, can safely be discarded because epochs change only when failures occur. For example, when a LocalJM receives a message (e.g., checkpoint command) with a previous epoch, the LocalJM discards the message.

**Axiom 3** A message from a previous epoch received while the system is recovering from a failure signifies multiple failures.

After a failure, the LeaderJM starts the recovery process by regenerating any failed LocalJMs. During the recovery procedure, if the LeaderJM receives a failure-notification message from a previous epoch, it can deduce that another failure has occurred. The LeaderJM detects failure of a LocalJM, and the LeaderJM increases the epoch value (EV) by one. Since other LocalJMs are not notified of the failure, the LocalJM reports the failure of its MPI process to the LeaderJM with the previous epoch value, and the LeaderJM receives the failure-notification message with the previous epoch. Then, the LeaderJM recognizes the occurrence of multiple failures, and starts a new recovery procedure with a new epoch.

**Axiom 4** In the case of multiple failures, the latter failure can be handled only if the previous failure did not result in failed LocalJMs or if the failed LocalJMs have been revived.

Once the LeaderJM deduces that multiple failures have occurred, it must set the foundation for another recovery procedure, i.e., it must verify that all LocalJMs that failed in the first failure have been revived. It is not necessary to complete the recovery procedure because the recovery procedure must be executed again. The recovery procedure includes an atomic operation in which all LocalJMs restart their MPI process based on the last checkpoint. The system is said to have recovered from multiple failures if the recovery procedure has been successfully completed for the last failure.

Our system can recover from multiple failures of different types. Even when multiple MPI processes and LocalJMs have failed, all that the LeaderJM must do is to restart the LocalJMs that have failed (either on the same node or in different nodes), and each LocalJM, in turn, restarts an MPI process.

### 3 IMPLEMENTATION ISSUES

#### 3.1 Atomic Message Transfer

We provide atomicity of message transfer in order to store and restore communication context safely. In other words, checkpointing is not performed while message transfer is in progress. We have made MPI communication operations mutually exclusive by using a checkpoint signal handler. Each mutually exclusive area for send and receive operations has been implemented in different levels. We set the whole send operation area as a critical section for Eager send. However, we differentiated between a non-critical Ckpt-unsafe state and a critical section state for Rendezvous send. Only the critical section state is mutually exclusive.

#### 3.2 Checkpoint/Restart Library

Zandy’s checkpoint/restart library, Ckpt [35], which allows the user program’s state to be saved without modifications to the user’s source code, was imported to the MPICH-GM, MVAPICH, and MPICH-P4 device libraries for Myrinet, InfiniBand, and ch_p4 interfaces, respectively. This allows LocalJMs to send signals to application processes. The reason for using system-level (user-transparent) checkpointing, rather than using an application-level (user-aware) approach, is to prevent the application programmer from rewriting their program to support fault tolerance. This is the trade-off between having the freedom to select checkpointable points and using the checkpoint library without modification. It is worth while developing a new checkpoint/restart library to support other processor architecture because the current version only works on Intel x86 processor.

#### 3.3 Communication Channel Reconstruction

The original MPI specification supports static process group management. That is, once a process group and channels are built, this information cannot be altered during runtime. In other words, no new process can join the group. Once a failure occurs, the communication channels between all MPI processes become invalid. Therefore, before MPI processes can communicate with each other, they must reconstruct the communication channels. This is explained in detail in Section 2.3.2. We implemented a new function for this purpose. To reconstruct the channels, MPI processes need to exchange the following information in the InfiniBand cluster (for the Myrinet cluster, refer to [4]):

- global rank: the logical process ID of the previous run
- hostname: the address of the current node where the process has been restored
- node identifier: local (intra-subnet) and global (inter-subnet) identifier (LID and GID)
- information: the Queue Pair (qp) ID, the start address of the DMAable memory region (mr), the size of the mr, and the associated remote key (rkey)

For ch_p4 devices, the node identifier is the IP address of the node and the information is the port number. Global rank and hostname are identical with those for InfiniBand.

#### 3.4 Checkpoint Image Transfer Scheduling

In today’s high-performance cluster configurations, which exploit shared disk storage systems, a naïve design for saving checkpoint images to shared disks can deteriorate the overall efficiency of the checkpointing procedure. Moreover, our experience demonstrates that concurrent writing of large checkpoint files to the network-attached GPFS file system [36] may stop the entire file system. The checkpoint image transfer scheduling has been studied in some theoretical checkpoint and recovery
Checkpoint image transfer scheduling is primarily motivated by the following: (1) concurrent writing to disk interferes with the execution of MPI processes, (2) network traffic is proportional to the size of concurrently transmitted checkpoint images, and (3) control of congestion caused by excessive traffic slows down the execution of MPI processes.

We devised four methods: "NFS only," "Local Disk only," "Local Disk to NFS," and "Heap to NFS," to ascertain the most practical solution. "NFS only" refers to saving the checkpoint image directly to a remote storage device concurrently and "Local Disk only" refers to saving the checkpoint image on only the local storage device. "Local Disk to NFS" refers to saving the checkpoint image on the local storage device before transferring it to a remote storage device when instructed to do so by the LeaderJM. "Heap to NFS" is similar to "Local Disk to NFS," except that the image is first stored in memory rather than on a storage device.

In the latter two methods, the LeaderJM schedules the image transfer procedure for only one of all checkpoint images to be transferred to the remote storage device at any time. The "Local Disk only" method may be used to guard against process failures, but because the checkpoint image is saved only on the local storage device, it cannot protect the system from node failures. The "Local Disk to NFS" method provides protection from node failures by allowing process migration. The "Heap to NFS" method is faster than writing to disk but may not be practical for long-running scientific codes as these applications tend to be memory-constrained. However, this method may be useful in clusters where computational nodes are diskless but equipped with large memory or in a shared disk environment.

"Local Disk to NFS" and "Heap to NFS" will lead to better performance than "NFS only" because in the former two methods (1) a separate process is assigned to transfer the checkpoint image to the remote storage, which allows the original process to execute without waiting for the completion of the transfer, and (2) the transfer of checkpoint images is serialized.

Here, we model the total overhead of the checkpoint assuming the "Local Disk to NFS" scheduling method. Let there be \( n \) MPI processes from \( p_0 \) to \( p_{n-1} \). Before checkpoint images can be saved, MPI processes must be pre-coordinated to ensure that no in-transit messages exist. We denote this overhead as \( s \). Let the time for process \( p_i \) to save a checkpoint image in its local disk be \( l_i \). Since the MPI process waits until all other processes complete saving their checkpoint image, it takes \( \max[l_i] \) to finish all local checkpointings. Once all checkpoint images have been stored in each local storage device, each LocalJM forks a child process to transfer the checkpoint image to the remote storage device, starting with the process with rank number 0. Let the time it takes for each child process to transfer its checkpoint image be \( t_i \). The MPI process continues computation while the child process transfers the checkpoint image. The concurrent execution of these two processes prevents the MPI process from taking full advantage of the CPU. In other words, transferring a checkpoint image may interfere with the normal computation of the MPI process. Also, the interference continues until all checkpoint images of MPI processes are transferred. We can model this effect using an ‘Interference factor’, \( \lambda \), and image transfer time, \( t_i \), as \( \lambda \left( \sum_{i=0}^{n-1} t_i \right) \). Finally, the total checkpoint overhead, \( O_{ckt} \), can be obtained from the following formula.

\[
O_{ckt} = c \left( s + \max[l_i] + \lambda \left( \sum_{i=0}^{n-1} t_i \right) \right),
\]

where \( c \) is the total number of checkpoints over the entire period.

The only difference with the "Heap to NFS" and the "Local Disk to NFS" method is that not each LocalJM, but instead each MPI process spawns a child process to transfer its checkpoint image. Therefore, we can use equation 1 again to model the total checkpoint overhead of the "Heap to NFS" method. Since the "Local Disk only" method does not include the transfer of the checkpoint image, we can safely delete the last term from equation 1 to model the checkpoint overhead as follows:

\[
O_{ckt} = c \left( s + \max[l_i] \right)
\]

From an MPI process’ viewpoint, it is not possible to distinguish remote storage from local storage when each MPI process saves its checkpoint image. We can model the checkpoint overhead of the "NFS only" method using equation 2. However, the value of the \( \max[l_i] \) term will be very large.

3.5 The Recoverability of a Data File

To guarantee the recoverability of a process that may perform I/O operations to data files, we import the ReFS module [39] into Athanasia. When a process performs checkpointing, ReFS snapshots related open files as well as metadata. After checkpointing open files, Athanasia moves both a process checkpoint and a file checkpoint to a remote storage. Athanasia, therefore, achieves the complete recoverability of all program contexts; it is able to recover a process image, files, and communication channels in a user transparent way.

4 Evaluation

4.1 Experimental Environment

We implemented Athanasia for the Myrinet, InfiniBand, and ch_p4 devices. The experiments for Myrinet and ch_p4 implementations were performed on the Hamel cluster, serviced by the Korea Institute of Science and Technology Information (KISTI) Supercomputing Center. The Hamel cluster consists of 256 nodes with dual Intel Xeon 2.8 GHz CPUs, 3 GB RAM, and a 36 GB SCSI disk drive running Linux 2.4.20. All machines are connected through a switched 1 Gbps Ethernet LAN and Myrinet 2000 and share a 10 TB storage device. The storage device consists of 8 file server nodes (IBM eServer x345) with dual Xeon 2.8 GHz CPUs and 4 GB RAM and 2 storage server nodes (FASiT900) with 3 disk boxes. The file server nodes and storage server nodes are connected through dual TotalStorage SAN FC F16 switches. The file system that runs on the file server nodes is GPFS [36]. The experiment for InfiniBand implementation was performed on our 8-node cluster in which each node is equipped with dual Intel Xeon 3.0 GHz CPUs and 4 GB RAM running Red Hat Linux Enterprise 3 with 2.4.21 kernel. Nodes share a 1TB storage node with a Xeon 3.0 GHz CPUs and 4 GB RAM and are connected through a switched 1 Gbps Ethernet LAN and InfiniBand. Version PBSPro_5.4.0.40152 of the Portable Batch System, PBS, by Altair Grid Technologies was used as the third-party job scheduler.

We assessed the performance of Athanasia both by use of a benchmark test (strong scaling) and by running real-world researches [37], [38], and we adapted the previous results to the extent of leveraging both shared and local disk storage.
applications (weak scaling). Benchmark testing was performed by running LU and BT of the Numerical Aerodynamics Simulation (NAS) Parallel Benchmark 2 suite, a set of programs designed to measure the performance of parallel computers. LU solves a finite difference discretization of the 3-D compressible Navier-Stokes equations through a block-lower-triangular and block-upper-triangular approximate factorization of the original difference scheme. LU is the most communication-intensive application among NAS Parallel Benchmarks. BT is similar to LU, but the core of the BT solution phase involves the solution of (5x5)-systems of equations and multiplication of (5x5) matrices, both of which are part of the application of GE to the block-tridiagonal systems. LU sends messages less than 50 bytes each, while BT is communication-bound and sends medium-sized messages ranging from a few hundred to a few thousand bytes each. LU and BT are categorized into four classes based on the problem size. Classes A and B are small-sized while classes C and D are relatively large-sized. Hence, we used classes A and B for experiments in our own cluster and C and D for experiments in the Hamel cluster.

For a real-world application test, we used four applications from various fields of science. The applications are either written by scientists for private use or open source programs. Each application carries out parallel scientific computation. The lifetimes of the applications are dependent on the input provided by a user. These applications are four of the most commonly executed applications on the Hamel cluster. 

`mm_par` [40] is a molecular dynamics (MD) simulation application. Molecular dynamics simulations are used to study molecular systems at nanometer and nanosecond scales. `mm_par` is parallelized with MPI to enhance performance. Each processor must update its information before calculation, leading it to communicate with all other processors. As a result, the communication cost becomes significant. `mpin` [41] is a parallel Navier-Stokes solver used in aerospace engineering to study control methods of unsteady, separated flow fields. `droplet` [42] is a tool used in mechanical engineering to simulate liquid droplets of argon surrounded by vapor for various temperatures and various sizes using parallel molecular dynamics. The application calculates Leonard-Jones argon droplets and calculates local density profiles of the argon systems. `heat2d` applies alternative directional implicit (ADI), approximate factorization (AF), and Crank-Nicolson schemes to simulate heat conduction in a two-dimensional space.

Every experiment was repeated five times. We chose each measured value from the case of which running time is median.

### 4.2 Results: Athanasia for Myrinet Device

#### 4.2.1 Checkpoint Overhead and Running Time

LU and BT were executed with a checkpoint frequency of 60 and 100 seconds, respectively.

Table 1 shows the size of one checkpoint image created by each process, and the average time required to pre-coordinate (PC OH) and write to disk (Disk OH) in each checkpoint and Table 2 shows the number of checkpoints. The total overhead (Total OH) is the sum of the pre-coordination overhead and disk overhead. We used class C of LU and BT while varying the number of nodes. Logically, the runtime of the process on Athanasia with no checkpoint is shorter than that on Athanasia with checkpoints due to the checkpoint overhead. Since the checkpoint frequency is constant, longer runtimes lead to more checkpoints. In the case of LU class C with eight nodes, the size of the checkpoint image of each process is about 143 MB and the average checkpoint overhead, the sum of pre-coordination (0.49) and disk overhead (2.08), is around 2.6 seconds. Since LU class C with 8 nodes has 15 checkpoints, the total checkpoint overhead is approximately 38 seconds. The other experiments can be analyzed similarly.

The pre-coordination time is composed of channel flush time and barrier exchange time. The barrier exchange time is proportional to the number of processes, while the channel flush time depends on application characteristics (i.e., applications exchange many MPI messages) or the time of receiving a checkpoint signal (i.e., applications are exchanging rendezvous messages on receiving a checkpoint signal). Thus, in the case of LU experiments, since the channel flush time decreases substantially, the pre-coordination time decreases even if the number of processes increases.

As the size of a process image grows, the disk overhead tends to increase. Since the disk overhead could be affected by activities of the underlying OS such as file system cache, disk overhead values could show slight inconsistency. For example, the disk overhead of LU.C.8 is twice as large as that of LU.C.62, while the disk overhead of LU.C.16 is slightly larger (1.2 times) than that of LU.C.128. This overhead, however, is not large enough to significantly affect the running time. Since pre-coordination is basically a synchronization mechanism, the time required for pre-coordination is dependent on the number of nodes. Pre-coordination can also be lengthened if there are in-transit messages. This overhead, however, is negligibly small, as shown in Table 1. Since the checkpoint overhead remains small even with more participating nodes, we can conclude that Athanasia is efficient and scalable.

Figure 5 shows the running times of LU and BT. Since the problem size is constant for both LU and BT, as the number of nodes increases, the running time decreases. The amount of memory used by each node also decreases, which leads to a smaller checkpoint size.

Figure 6 shows the execution time of real world applications, and Table 3 shows the checkpoint overhead of each application. All real world applications are used for weak scaling and the checkpoint image size of each application is the same regardless of the number of processes. In this experiment,
we set the input values for each application so that they can finish their work in about two hours. Since the focus of the experiments was to measure the checkpoint overhead, only a few checkpoints were required. A checkpoint interval for this experiment is set to 50 minutes, which means that each application will save two checkpoint images. Even if mm_par has the largest checkpoint overhead due to its large size, the overhead for mm_par is less than 1% of the total running time. From this experiment, we can see that the checkpoint overhead is negligible for applications whose checkpoint image size is less than 300MB and checkpoint interval is around 50 minutes.

The experimental results with mm_par characterize the behavior of the interference factor $\lambda$ in Equation 1. In the case of mm_par shown in Figure 6(a), the checkpoint image was around 308 MB regardless of the number of processes. Even if mm_par has the largest checkpoint overhead due to its large size, the overhead for mm_par is less than 1% of the total running time. From this experiment, we can see that the checkpoint overhead is negligible for applications whose checkpoint image size is less than 300MB and checkpoint interval is around 50 minutes.

The decrease in the interference factor indicates that more computation takes place while the checkpoint image is transferred by a communication process. It is noted that with satisfactory conditions (both network and storage) it shows better performance to transfer multiple checkpoint images simultaneously.

### 4.2.2 Startup and Recovery Cost

Figure 7 shows the startup and recovery cost (after process failure) of LU and the startup cost of real-world applications. Startup cost is determined by the time it takes for MPI processes to exchange communication information and initialize devices. The higher the number of MPI processes executing in parallel, the higher the startup cost. The startup procedure includes launching LocalJM via `ssh`. We observed that launching a LocalJM requires more time than we expected.

#### Table 3
Checkpoint Overhead (“Local Disk only”)

<table>
<thead>
<tr>
<th>Application</th>
<th>Ckpt size (MB)</th>
<th>PC OH (seconds)</th>
<th>Disk OH (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mm_par (16)</td>
<td>308</td>
<td>0.2207</td>
<td>1.039</td>
</tr>
<tr>
<td>mm_par (32)</td>
<td>314</td>
<td>0.3147</td>
<td>1.007</td>
</tr>
<tr>
<td>mm_par (64)</td>
<td>482</td>
<td>0.4819</td>
<td>1.1139</td>
</tr>
<tr>
<td>mpin (16)</td>
<td>88</td>
<td>0.1740</td>
<td>0.9637</td>
</tr>
<tr>
<td>mpin (32)</td>
<td>325</td>
<td>0.3251</td>
<td>0.9042</td>
</tr>
<tr>
<td>mpin (64)</td>
<td>64</td>
<td>0.6418</td>
<td>0.8917</td>
</tr>
<tr>
<td>droplet (16)</td>
<td>50</td>
<td>0.3181</td>
<td>0.6794</td>
</tr>
<tr>
<td>droplet (32)</td>
<td>399</td>
<td>0.3998</td>
<td>0.6174</td>
</tr>
<tr>
<td>droplet (64)</td>
<td>64</td>
<td>0.4129</td>
<td>0.8705</td>
</tr>
<tr>
<td>heat2d (16)</td>
<td>46</td>
<td>0.1750</td>
<td>0.6491</td>
</tr>
<tr>
<td>heat2d (32)</td>
<td>213</td>
<td>0.2138</td>
<td>0.6231</td>
</tr>
<tr>
<td>heat2d (64)</td>
<td>257</td>
<td>0.2572</td>
<td>0.6072</td>
</tr>
</tbody>
</table>

#### Table 4
Estimating Interference Factor ($\lambda$) in mm_par

<table>
<thead>
<tr>
<th># of processes</th>
<th>Transfer time (A)</th>
<th>Difference (B)</th>
<th>$\lambda := \frac{B}{A}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>60</td>
<td>30</td>
<td>0.5</td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>25</td>
<td>0.25</td>
</tr>
<tr>
<td>16</td>
<td>200</td>
<td>40</td>
<td>0.2</td>
</tr>
<tr>
<td>32</td>
<td>340</td>
<td>40</td>
<td>0.12</td>
</tr>
<tr>
<td>64</td>
<td>500</td>
<td>40</td>
<td>0.08</td>
</tr>
</tbody>
</table>
due to the unexpected high response time of $ssh$. This leads to the unexpected startup costs of LU (LU.C.16 and LU.C.64). Unexpected recovery costs of LU (LU.C.8, LU.C.16 and LU.C.64) result from the unexpected longer time for fetching checkpoint images from the local storage device.

Figure 7(b) shows the average startup times for each real-world application. Even with 64 processes, startup takes only a few seconds. Considering (1) that startup only occurs once, and (2) that non-fault-tolerant execution using $mpiexec$ also requires an initialization period, we can say that the initialization cost of Athanasia is acceptable.

Our system supports both process and node failures. The recovery cost is the sum of the time it takes (1) to fetch the checkpoint image that the LeaderJM designates, (2) to reincarnate the MPI processes by calling $fork$ and $exec$, and (3) to exchange communication information between MPI processes. Therefore, the smaller the size of checkpoint images and the lower the number of MPI processes, the smaller the recovery cost. With satisfactory network conditions (large bandwidth and short latency), disks, and file systems, the range of the recovery cost will remain small.

We simulated process failures by killing the process using the $SIGKILL$ signal. We then measured the time it took for the process to restart using its checkpoint image. Since the node was not affected, the reincarnated process read the checkpoint image from its local storage device. In the case of node failures, the recovery time would reflect the time it takes to detect the failure and the time it takes for the failed process to read the checkpoint image from a remote storage device.

![Fig. 8. Recovery Overhead](image)

**Fig. 8. Recovery Overhead**

Figure 8 shows the recovery cost for real-world applications. After the processes are restarted, they must exchange their new communication information. Therefore, recovery from a process failure should take little more than what initialization required. A quick comparison between Figures 7(b) and 8(a) shows that this is true. The recovery time largely depends on the size of the checkpoint image and the number of MPI processes. $mm_{par}$, which creates a larger image than the others, requires more time than the others to recover from a node failure.

The recovery from a node failure includes launching a LocalJM via $ssh$ on a new node. We observed that launching a LocalJM requires more time than we expected due to an undeterministic high response time of $ssh$. This leads to the unexpected results of $heat2d$ (4, 8 and 16 processes) and droplet (4 processes).

Table 5 shows the recovery time from the LeaderJM failure. If a failure occurs on the node on which the LeaderJM resides, a new LeaderJM must be chosen. First, LocalJMs must reach a consensus regarding the LeaderJM failure. After the consensus procedure, the LocalJM with the smallest rank number among LocalJMs that participate the consensus spawns a new LeaderJM on a new node, that is similar to the leader election procedure. The LeaderJM takes job information such as the application name, the current epoch number and the checkpoint interval from the LocalJM that spawns the LeaderJM. The recovery cost from the LeaderJM failure is proportional to the number of processes. Table 5 shows that even with 64 nodes, this recovery procedure is finished in a matter of seconds. Since the recovery procedure is performed by LocalJMs, the procedure does not affect the MPI computation, that allows the MPI computation time to be independent of the time for the recovery from the LeaderJM failure.

### 4.2.3 Checkpoint Image Transfer Cost

![Fig. 9. Running Time in Each Scheduling Scheme](image)

**Fig. 9. Running Time in Each Scheduling Scheme**

<table>
<thead>
<tr>
<th># of Processes</th>
<th>Consensus</th>
<th>Election</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0.0002771</td>
<td>0.000088</td>
<td>0.000285</td>
</tr>
<tr>
<td>8</td>
<td>0.03252</td>
<td>0.00151</td>
<td>0.03403</td>
</tr>
<tr>
<td>16</td>
<td>0.209682</td>
<td>0.003844</td>
<td>0.21356</td>
</tr>
<tr>
<td>32</td>
<td>1.423554</td>
<td>0.025834</td>
<td>1.44939</td>
</tr>
<tr>
<td>64</td>
<td>2.507117</td>
<td>0.106725</td>
<td>2.61384</td>
</tr>
</tbody>
</table>

**TABLE 5**

**Recovery From LeaderJM Failure (seconds)**

**TABLE 6**

**Checkpoint Overhead**

<table>
<thead>
<tr>
<th>Ckpt size (MB)</th>
<th>PC OH (seconds)</th>
<th>Disk OH (seconds)</th>
<th>Transfer OH (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LU.D.32</td>
<td>430</td>
<td>0.0625</td>
<td>2.9452</td>
</tr>
<tr>
<td>LU.D.64</td>
<td>230</td>
<td>0.1669</td>
<td>2.4642</td>
</tr>
<tr>
<td>LU.D.128</td>
<td>157</td>
<td>0.1854</td>
<td>2.1427</td>
</tr>
<tr>
<td></td>
<td>Local to NFS</td>
<td></td>
<td>5.095</td>
</tr>
<tr>
<td></td>
<td>Heap to NFS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 9 presents the evaluation results of checkpoint image transfer scheduling schemes using LU with D class. This experiment was executed with a checkpoint frequency of 1000
seconds, and there were three, two and one checkpoints in LUD.32, LUD.64 and LUD.128 cases, respectively. Table 6 shows per-checkpoint overhead of each scheduling policy. PC OH, Disk OH and Transfer OH means the average time required to pre-coordinate, write to a local disk and transfer each image to the remote storage device in each checkpoint, respectively. Since “Heap to NFS” uses heap space as temporal checkpoint storage instead of local disk, the average time required to copy an MPI process’ image to its heap space (Memory OH) is needed to be measured. In the case of “NFS only”, all MPI processes try to save their images to a remote storage device, and NFS OH means the total time for all processes to complete their checkpoint transfer.

As expected, “Local Disk Only” incurs the least overhead. “Heap to NFS” shows slightly better performance than “Local Disk to NFS.” Although copying to memory is faster than writing to the local file system, the difference is small because modern file systems cooperate with buffer or cache mechanisms. “NFS only” has the largest overhead among the four methods due to the high costs of concurrent write operations on the remote storage and transfer of checkpoint images. “Heap to NFS” and “Local Disk to NFS” have a significantly lower overhead, around 300 - 500 seconds, than “NFS only” because allocating a separate process to handle the transfer allows execution to continue while checkpointing and transferring checkpoint images serially exploits network capacity and the remote storage device.

In “Heap to NFS” and “Local Disk to NFS,” the transfer of checkpoint images overlaps with computation for about 50 seconds. It is possible to further reduce the time spent on each checkpoint if more than one checkpoint image can be transferred concurrently with larger network bandwidth.

4.3 Results: Athanasia for InfiniBand and ch_p4 Devices

The experimental results of Athanasia for InfiniBand and ch_p4 devices are quite similar to those for the Myrinet device. Hence, in this section, we omit much of the experimental results and show only meaningful ones.

<table>
<thead>
<tr>
<th>Ckpt size</th>
<th>No Ckpt</th>
<th>Local PC OH</th>
<th>Local Disk OH</th>
<th>NFS OH</th>
</tr>
</thead>
<tbody>
<tr>
<td>154 MB</td>
<td>61</td>
<td>+17</td>
<td>+28</td>
<td>+44</td>
</tr>
<tr>
<td>123 MB</td>
<td>83</td>
<td>+24</td>
<td>+37</td>
<td>+73</td>
</tr>
<tr>
<td>339 MB</td>
<td>323</td>
<td>+42</td>
<td>+58</td>
<td>+267</td>
</tr>
<tr>
<td>191 MB</td>
<td>203</td>
<td>+16</td>
<td>+28</td>
<td>+152</td>
</tr>
<tr>
<td>327 MB</td>
<td>933</td>
<td>+55</td>
<td>+74</td>
<td>+261</td>
</tr>
<tr>
<td>mm_par</td>
<td>832 MB</td>
<td>+39</td>
<td>+66</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 7 shows the running time and checkpointing overhead of each experiment in the InfiniBand implementation. “No Ckpt” refers to the running time of applications using MVAPICH2-0.6.0. In each case, we configured checkpoint period to perform only one checkpointing during the run time. As seen in the table, except for “NFS Only,” the performance of Athanasia is comparable to that of non-fault-tolerant MVAPICH2.

Table 8 shows the breakdown of the overhead of Athanasia on the InfiniBand device. We can see that the checkpointing overhead lies within a reasonable range. The startup cost of Athanasia for InfiniBand is relatively larger than that of Athanasia for Myrinet since InfiniBand takes a longer time to establish a communication channel than Myrinet and Athanasia for InfiniBand exchanges n * (n – 1) channel information for n MPI processes.

<table>
<thead>
<tr>
<th>Ckpt size</th>
<th>No Ckpt</th>
<th>Local PC OH</th>
<th>Local Disk OH</th>
<th>NFS OH</th>
</tr>
</thead>
<tbody>
<tr>
<td>154 MB</td>
<td>61</td>
<td>+17</td>
<td>+28</td>
<td>+44</td>
</tr>
<tr>
<td>123 MB</td>
<td>83</td>
<td>+24</td>
<td>+37</td>
<td>+73</td>
</tr>
<tr>
<td>339 MB</td>
<td>323</td>
<td>+42</td>
<td>+58</td>
<td>+267</td>
</tr>
<tr>
<td>191 MB</td>
<td>203</td>
<td>+16</td>
<td>+28</td>
<td>+152</td>
</tr>
<tr>
<td>327 MB</td>
<td>933</td>
<td>+55</td>
<td>+74</td>
<td>+261</td>
</tr>
<tr>
<td>mm_par</td>
<td>832 MB</td>
<td>+39</td>
<td>+66</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 8 shows the overhead of recovery from failures. In “Heap to NFS” and “Local Disk to NFS,” the transfer of checkpoint images overlaps with computation for about 50 seconds. It is possible to further reduce the time spent on each checkpoint if more than one checkpoint image can be transferred concurrently with larger network bandwidth.

4.3 Results: Athanasia for InfiniBand and ch_p4 Devices

The experimental results of Athanasia for InfiniBand and ch_p4 devices are quite similar to those for the Myrinet device. Hence, in this section, we omit much of the experimental results and show only meaningful ones.

<table>
<thead>
<tr>
<th>Ckpt size</th>
<th>No Ckpt</th>
<th>Local PC OH</th>
<th>Local Disk OH</th>
<th>NFS OH</th>
</tr>
</thead>
<tbody>
<tr>
<td>154 MB</td>
<td>61</td>
<td>+17</td>
<td>+28</td>
<td>+44</td>
</tr>
<tr>
<td>123 MB</td>
<td>83</td>
<td>+24</td>
<td>+37</td>
<td>+73</td>
</tr>
<tr>
<td>339 MB</td>
<td>323</td>
<td>+42</td>
<td>+58</td>
<td>+267</td>
</tr>
<tr>
<td>191 MB</td>
<td>203</td>
<td>+16</td>
<td>+28</td>
<td>+152</td>
</tr>
<tr>
<td>327 MB</td>
<td>933</td>
<td>+55</td>
<td>+74</td>
<td>+261</td>
</tr>
<tr>
<td>mm_par</td>
<td>832 MB</td>
<td>+39</td>
<td>+66</td>
<td>-</td>
</tr>
</tbody>
</table>
5 Conclusion

This article presents Athanasia, a user-transparent and multiple fault-tolerant system. The primary purpose of Athanasia is enabling MPI processes to be resilient to various failures without violating user transparency. Athanasia demonstrates that it is possible for parallel MPI processes to realize an extremely high level of reliability with minor performance overhead, which is largely dependent on how the checkpoint image transfer scheduler is implemented.

Athanasia is implemented for Myrinet, InfiniBand, and ch_p4 interfaces. Our experience with Athanasia shows that: (1) the implementation of a multiple fault-tolerant system is achievable with only modest engineering efforts, (2) the checkpoint-write overhead is not a trivial problem in commercial fields, and (3) the performance degradation caused by a naive implementation of checkpoint-write can be easily overcome with memory-based checkpoint image transfer scheduling.

References