Cluster Computing 7, 303–315, 2004
© 2004 Kluwer Academic Publishers. Manufactured in The Netherlands.

MPI/FT: A Model-Based Approach to Low-Overhead Fault Tolerant Message-Passing Middleware

RAJANIKANTH BATCHU and YOGINDER S. DANDASS *
Mississippi State University, Department of Computer Science, Box 9637, Mississippi State, MS 39762, USA

ANTHONY SKJELLUM
Verari Systems Software, Inc., 110 12th Street North, Suite D103, Birmingham, AL 35203, USA

MURALI BEDDHU
The University of Southern Mississippi, Department of Computer Science and Statistics, 118 College Drive, Box 5106, Hattiesburg, MS 39406, USA

Abstract. Fault tolerance in parallel systems has traditionally been achieved through a combination of redundancy and checkpointing methods. This notion has also been extended to message-passing systems with user-transparent process checkpointing and message logging. Furthermore, studies of multiple types of rollback and recovery have been reported in literature, ranging from communication-induced checkpointing to pessimistic and synchronous solutions. However, many of these solutions incorporate high overhead because of their inability to utilize application level information.

This paper describes the design and implementation of MPI/FT, a high-performance MPI-1.2 implementation enhanced with low-overhead functionality to detect and recover from process failures. The strategy behind MPI/FT is that fault tolerance in message-passing middleware can be optimized based on an application's execution model derived from its communication topology and parallel programming semantics. MPI/FT exploits the specific characteristics of two parallel application execution models in order to optimize performance. MPI/FT also introduces the self-checking thread that monitors the functioning of the middleware itself. User aware checkpointing and user-assisted recovery are compatible with MPI/FT and complement the techniques used here.

This paper offers a classification of MPI applications for fault tolerant MPI purposes and MPI/FT implementation discussed here provides different middleware versions specifically tailored to each of the two models studied in detail. The interplay of various parameters affecting the cost of fault tolerance is investigated. Experimental results demonstrate that the approach used to design and implement MPI/FT results in a low-overhead MPI-based fault tolerant communication middleware implementation.

Keywords: model-based fault tolerance, MPI, cluster computing, fault detection, group communication

1. Introduction

Clusters of commodity-off-the-shelf (COTS) components are rapidly replacing traditional supercomputers. Recent advances in processors and interconnect technologies have made clusters more reliable, scalable, and affordable [7]. They have become the hardware of choice for solving problems in several domains that require large computation power. The communication middleware and the parallel programming environment determine the efficacy of clusters for parallel computing. Typically, the middleware connects the various components in the cluster and the parallel programming environment provides the abstractions and programming interfaces to users. MPI is currently the *de facto* standard that defines the API for a message-passing parallel programming model [20]. MPI is designed for achieving portable high-performance communication in parallel applications. Numerous implementations of MPI have been realized in industry and academia, and MPI has successfully been used in many domains (e.g., scientific computing, visualization, and bioinformatics) [2,5,16,23].

With the increasing popularity of clusters and MPI applications, the issue of reliability at the middleware and application layers has recently gained prominence. Clusters inherently increase the availability at hardware level with redundant and hot-swappable components. However, this availability and reliability are not automatically transferred to higher layers. In long-running and large-scale applications and in harsh environments (e.g., in space-based applications [16]), introduction of *external faults* (i.e., faults unrelated to software defects) is unavoidable. For example, MPI processes may fail because of processor, memory, power supply, or network failures. Because MPI provides a static process model that requires all processes to finish successfully, faults that affect only a single process typically result in application-wide failure (in fact, the standard mandates this behavior).

A failed application may terminate gracefully, hang indefinitely, or crash, depending on its implementation, point of failure, and failure mode. Typical fault mitigation strategies in MPI environments involve saving application progress at predetermined checkpoints and to recover failed applica-

* Corresponding author.
E-mail: yogi@cse.msstate.edu

tions from checkpoint data. The MPI-1.2 standard, however, provides limited fault handling facilities (MPI "error return" mode) but lacks application checkpoint, recovery mechanisms, and asynchronous notification of failure events.

Several research efforts have realized reliable MPI at the cost of high overheads that contradict the high-performance goal of MPI. Most of these efforts and resulting middleware are based on "black-box" approaches that provide static abstractions of fault tolerance features and services to all types of parallel applications [1,12,26,28]. These rigid approaches, however, can result in unnecessarily large overheads in support of unused features and limited the flexibility to applications that can tradeoff reliability in favor of performance.

This paper presents MPI/FT [3], an MPI-based middleware that provides additional services for detection and recovery of failed MPI processes. These services for fault tolerance are tailored to minimize overhead for specific application execution models, resulting in separate middleware implementation for each model. Application execution models are abstractions derived from certain combinations of communication topologies and parallel programming models that are amenable to providing low-overhead fault tolerance services. The presented API supports MPI-1.2 with the understanding that the model implicitly uses more restrictive syntax and semantics.

MPI/FT is derived from MPI/Pro 1.51 [9,10,22], a fully progressive implementation of the MPI-1.2 standard. MPI/Pro utilizes progress threads in order to maximize the opportunity for transferring asynchronous sends and receives independent of the number of times a user calls the library. Polling progress is also optionally supported for certain applications that prefer to poll. MPI/Pro optimizes many aspects of the MPI-1.2 specification, including high-speed implementation of persistent communication, efficient gather/scatter datatype handling, fast collective algorithms, and both asynchronous notification and asynchronous transfer of data.

MPI/Pro supports high-speed networks, TCP/IP, and shared-memory communications, as well as multiple operating systems. MPI/Pro's design emphasizes that the triple {overhead, latency, bandwidth} must be optimized to yield highest parallel application performance, and demonstrates that for most applications, minimum latency is not the optimal; low latency, together with low overhead and high bandwidth are apparently optimal in practice, because these support high utilization of the CPU by the application. Both polling and non-polling MPI's achieve comparable asymptotic bandwidths.

In the context of MPI/FT, MPI/Pro's progress threads are modified and monitored in support of fault detection. These progress threads are central to the efficient operation of MPI/Pro under non-fault conditions, and double in this work as access points to asynchronous communication without broad modification of the MPI implementation.

In MPI/FT, the MPI API is also extended to provide services fault detection including, notification, recovery, process redundancy, and user-aware checkpointing. The selection and implementation of fault tolerant services depend on the targeted application execution model in order to optimize performance and eliminate avoidable overhead. Furthermore, the fault tolerant services in MPI/FT allow flexibility to users in trading off reliability for performance as may be required by a particular application. (The extended API is not emphasized in this paper.)

The MPI/FT fault model assumes that the applications are free of *internal faults* (i.e., flawed software). This is because if flawed software causes an MPI process failure, then after recovery is completed and execution is resumed from a checkpoint, the software flaw can cause yet another process failure. This progression can repeat indefinitely with the application making little or no progress. Additionally, MPI/FT only covers external faults that lead to a *complete process failure* that is manifested in the form of an unresponsive process. Conversely, transient faults that lead to the production of incorrect results, but allow the MPI application to continue are not addressed here. Detection of coverage of such transient errors must be provided by the application through the explicit use of traditional fault detection and recovery techniques (e.g., algorithm- or application-based fault detection and recovery [14,29]).

The remainder of the paper is organized as follows. Section 2 presents the deficiencies in the MPI standard as related to fault detection and recovery. This section also addresses results from previous work in implementing fault tolerant MPI. Section 3 identifies features of MPI applications that are used to identify application execution models. Section 4 describes in detail the design and implementation of MPI/FT with regard to two of the application execution models. Section 5 presents and analyzes experimental results to evaluate the effectiveness of the implementation and to measure the tradeoffs between performance and reliability in MPI/FT. Section 6 discusses potential future research, and section 7 offers conclusions.

2. Related work

2.1. The MPI standard

MPI-1 [20] has become the ubiquitous API for parallel message-passing scientific applications. The main goals of this *de facto* standard are to provide portable mechanisms for communications for parallel applications in a variety of high-performance environments. MPI-1 provides point-to-point and collective communication operations within the context of *communicators* (i.e., group and context-oriented communication) in a static world of processes that share a common session.

Because the addition of reliability features in communication evidently increases processing and system resource overheads, MPI offers limited fault-handling facilities. While consistent with the goal of achieving high-performance, these limited fault-handling features confine the usefulness of MPI to environments where the occurrence of external faults must be accounted for within the applications in order to ensure the completion of applications, or where hardware and lower-level services mask all faults.

Following is a summary of MPI deficiencies with respect to tolerance of external faults:

Fault Model: The MPI standard assumes that the underlying host and network infrastructure are 100% reliable. The standard and implementations do not provide mechanisms to applications for handling node failures or lost messages. For example, there is no timeout mechanism in the MPI user API, and therefore, a blocking receive call can wait indefinitely if the sending MPI process fails or if an unreliable communication layer loses the incoming message. In the presence of an unreliable communication layer, the MPI implementation must assume the responsibility of providing reliable message passing to the application.

Fault Detection and Notification: Detection of external faults is not defined in MPI. MPI only provides for detection of limited instances of local internal errors such as detecting and reporting incorrect function arguments and resource errors (e.g., exceeding buffer space). Notification of errors takes the form of invoking an application specified error handler function, and if the error handler does not halt the process, the MPI call returns an error code. The type of all errors detected and the error codes returned are not specified in the standard.

Fault Recovery: In MPI applications, recovery from external faults is inhibited by the restricted information contained in the non-standardized error codes returned by MPI calls. Recovery options are also hampered because MPI mandates a static communicator. Also, the MPI standard "mandates" killing jobs when a process or node fails.

Therefore, an application cannot start a new MPI process to take the place of a failed MPI process. Neither can the application continue in a degraded mode by ignoring the failed MPI process.

These limitations in the MPI standard and implementations make more difficult automatic, application-based, detection and recovery from faults. In particular, if any one MPI process in a parallel application *hangs* (i.e., becomes unresponsive because of a deadlock or an infinite loop), or *crashes* (i.e., terminates abnormally and prematurely), the entire application is required to terminate, either gracefully or abnormally, or in practice, may become unresponsive. In this situation users must restart the application. In order to make progress in the face of faults, users typically periodically save partial results in checkpoints, and recover prematurely terminated applications using the data from the last checkpoint. However, requiring human intervention in a simplistic terminate-and-restart regime is a source of significant inefficiency in fault tolerant parallel computing.

The MPI-2 [21] standard extends MPI by providing dynamic process management, one-sided communication operations, extended collective communication operations, and parallel I/O. The addition of dynamic process management is significant in that it allows applications to create and terminate process "worlds" under application control. Applications can use this feature to compensate for failed processes. However, these extensions do not address MPI's assumption that the underlying infrastructure is reliable. Therefore, detection of and options for recovery from external faults continue to be restricted in MPI-2.

Because of the popularity of MPI, several research efforts are striving to enhance MPI with reliability features. Solutions proposed in literature have ranged from transparent checkpointing, to emphasizing the health of communicators, to utilizing dynamic process management in MPI-2; these are discussed next.

2.2. CoCheck

CoCheck [28] is one of the earliest efforts to make MPI more reliable. CoCheck extends the single process checkpoint mechanism in Condor [18] to a distributed message-passing application. Unlike most checkpointing middleware, CoCheck is visible to the user, and is available at a layer above the message-passing middleware. Common problems with checkpointing and recovery such as global inconsistent states and domino effects [11] are eliminated through the use of a protocol to "flush" all in-transit messages before a checkpoint is created.

CoCheck is primarily intended to facilitate process migration, load balancing, and stalling of long running applications for resumption at a later time. CoCheck incurs a large overhead because it checkpoints the entire process state. Recovery of a failed process is performed by a user-level function. However, CoCheck only provides coarse-grained reliability for MPI because CoCheck does not have access to the message-passing middleware's internal data structures.

2.3. Egida

Egida [26] is an object-oriented toolkit for transparent logging, and rollback recovery. Egida is extensible and allows users to define their own logging and rollback recovery protocols. Implementations for the described protocol are synthesized by gluing pre-existing objects. Egida's log-based fault tolerance facilities emphasize the reduction of overhead during recovery. However, unless the checkpoint protocols are carefully designed, the use of Egida can lead to large overheads.

2.4. FT-MPI

The communicator is an important MPI data structure that defines a communication context and the set of processes for the context. Premature termination of a process places communicators involving the dead process in an invalid state. FT-MPI [12] allows the application to continue using a communicator with the failed rank while explicitly excluding communication with the failed rank, or to shrink the communicator by excluding the failed rank, or to spawn a new process to take the place of a failed process.

This ability of the communication middleware to repair an invalid communicator results in application-based recovery

choices other than restarting the application from a previous checkpoint. This flexibility enables the construction of applications with fine-grained fault recovery mechanisms. However, the FT-MPI literature does not elaborate on strategies used for failure detection. FT-MPI also does not provide an API for fault notification and for checkpointing.

2.5. Starfish

Starfish [1] provides a parallel execution environment that adapts to changes in the cluster caused by node failure and recovery. The Starfish environment for execution of dynamic MPI programs is based on the Ensemble group communication system [13]. Starfish uses an event model wherein application processes register to listen for events reflecting changes in cluster configuration and process failures.

Starfish also provides application- and system-driven checkpointing facilities. When a process failure is detected, Starfish can automatically recover the application from a previous checkpoint. However, consistency of communicators is not addressed in Starfish; in order to recover a single failed process, the entire MPI application must be restarted. Essentially, many of the powerful dynamic process management features of Starfish cannot be used directly by MPI applications.

3. Model-based fault tolerance

Many of the existing efforts to enhance the reliability of MPI use a "black-box" approach of providing a fixed set of services (e.g., transparent checkpointing, message logging, and roll-back recovery) to all applications. However, this approach can incur unnecessarily large overheads for certain classes of applications. For example, consider a closely coupled iterative parallel application that loads a large volume of data from mass storage on startup and performs an *all-reduce* operation at the end of every iteration. Further assume that the application only requires a small amount of memory for maintaining the progress state information (i.e., for variables containing intermediate results and data and iteration indices). However, under an application-transparent checkpointing and rollback regime the middleware must save a snapshot of the entire application state, including the entire memory allocated in all processes, and must log all in-transit messages, thereby incurring large overhead in terms of processing, storage, and network bandwidth.

The strategy behind the design of MPI/FT is that overhead can be minimized by providing fault tolerant message-passing middleware services specifically tailored to meet the requirements of the application at hand. In order to balance the cost of implementing and deploying customized middleware for many parallel applications versus the performance costs of middleware induced overheads, MPI/FT provides distinct middleware implementations for applications grouped into several high-level application execution models, summarized in table 1. MPI/FT also provides mechanisms for failure notification and process restart but leaves the task of checkpointing and resuming execution from the last valid checkpoint to the application.

Table 1
Application execution models.

Programming style	Communication topology	Middleware redundancy and checkpointing	Designation	Currently implemented
Master–slave	Star	None	*Model-Ia*	Yes
		Master only – passive	*Model-Ib*	No
		Master only – active	*Model-Ic*	No
Regular-SPMD	All-to-all	None	*Model-IIa*	Yes
		Coordinator at rank 0 only – passive	*Model-IIb*	No
		Coordinator at rank 0 only – active	*Model-IIc*	No

MPI/FT uses the following parameters to distinguish application execution models:

Virtual Communication Topology: MPI applications have a virtual communication topology defined by a directed graph wherein nodes represent processes and edges represent messages. Commonly used topologies in parallel applications include hypercube, rectangular grid, all-to-all fully connected, and star. This graph describes the virtual communication topology more precisely than the set of instantiated MPI communicators. For example, each process may communicate only to a subset of neighbors (i.e., in a stencil) while using the default all-inclusive MPI_COMM_WORLD communicator. The virtual communication topology of a parallel application directly influences the design and implementation of low-overhead collective communication, application-based detection, and middleware level recovery services.

Program Style: MPI-1.2 applications typically follow one of two common styles – single program multiple data (SPMD) or master–slave (or client–server). In these styles, program flow consists of iterations of computation interspersed with communication. The program style governs the appropriate location of calls to the MPI/FT fault tolerance services. Analysis of the program style also suggests potential optimizations (e.g., only a small subset of processes may need to be rolled-back in order to recover from a failure).

Middleware Redundancy: Replication of both processes and data is utilized for achieving reliability. Careful management of redundancy-induced overhead is required in order to achieve high performance. In order to reduce overhead, unnecessary redundancy must be eliminated wherever possible. For example in order to conserve resources, *passive* redundancy can be utilized instead of *active* redundancy whenever allowed by the application's reliability requirements [24]. In passive redundancy, spare processes are either idle or used for other tasks, and these spares must be initialized to replace the failed process. In active redundancy, the spare processes duplicate the tasks being performed by the processes that are to be replaced,

and therefore, do not require initialization when replacing failed processes.

3.1. The Master–slave application execution models

In these related models (see *Model-Ia*, *Model-Ib*, and *Model-Ic* in table 1), based on a virtual star communication topology, the master node communicates with the slave nodes and the slaves communicate with the master; slaves do not directly communicate with other slaves. Because there is no explicit interaction (including synchronization) between slaves, no collective calls are implemented or permitted for this model. The master node sends jobs to and receives results from the slaves.

Failure of a slave causes the middleware to return an error code to the master process from only those MPI calls involving the failed node; communication between the master and other nodes is normally unaffected. The master process uses MPI/FT services to restart the failed slave process and to resend the failed job. MPI/FT repairs the MPI_COMM_WORLD communicator at the master process to reflect the replaced process.

Failure of the master node requires that the entire application be restarted from a checkpoint. This results in a single single-point-of failure as opposed to multiple single points of failure in traditional MPI applications. *Model-Ib* and *Model-Ic* increase the resiliency of the application by enabling recovery of master node failure. However, this capability increases overhead because of the communication required to monitor the master node's health and the processing and I/O required to save the master's state in a checkpoint. (Currently, the design options for *Model-Ib* and *Model-Ic* are under investigation and their implementations are not yet available in MPI/FT.)

Note that for *Model-Ia*, MPI/FT does not provide a checkpoint mechanism because only the state of the master process needs to be saved in a checkpoint. This simple task can be accomplished within the application by streaming the state information to a data file, or otherwise.

3.2. The Regular-SPMD application execution models

In these related clique models (see *Model-IIa*, *Model-IIb*, and *Model-IIc* in table 1), all processes may communicate with each other (i.e., these models exhibit an all-to-all communication topology). Furthermore, the processes consist of synchronous iterative loops and the communication occurs at the start (or end) of each iteration.

In these models, the failure of an MPI process results in the synchronous propagation of the failure notification to all remaining ranks. As a result of this notification, all remaining processes rollback to the previously saved checkpoint. The middleware provides services to restart and recover the failed process from the saved checkpoint. The middleware also repairs the MPI_COMM_WORLD communicator at all the processes to reflect the replaced process.

In the current implementation of MPI/FT for the Regular-SPMD models, the process with rank 0 is responsible for coordinating the recovery of the failed process. Therefore, this *coordinator* process represents a single single-point-of-failure similar to that in the Master–slave models. In *Model-IIa*, the failure of rank 0 requires the manual restart of the entire application from the previous checkpoint. *Model-IIb* and *Model-IIc* enable the recovery of rank 0 and the underlying coordinator, resulting in increased application resiliency with a concomitant increase in overhead.

For these models, MPI/FT also provides a synchronous collective operation for producing globally consistent checkpoints. Consistency of checkpoints is established by requiring the participation of all processes in the collective operation and the checkpoint operation essentially behaves like a barrier operation. A complementary collective operation for reading data from a previously saved checkpoint is also provided. (As with *Model-Ib* and *Model-Ic*, the design options for *Model-IIb* and *Model-IIc* are currently under investigation and their implementations are not yet available in MPI/FT.)

4. Design and implementation of *Model-Ia* and *Model-IIa*

In addition to the existing send and receive progress threads in MPI/Pro, MPI/FT implements a self-checking thread (SCT) that monitors the status of the middleware in order to provide fault detection, notification and recovery services. An MPI/FT process consists of at least four threads: (1) the send progress thread, (2) the receive progress thread, (3) the fault monitoring SCT, and (4) one or more user thread. The send and receive progress threads derive from MPI/Pro's design, while the fault monitoring SCT is a new feature. The SCT at the master process in the Master–Slave models and at the process with rank 0 in the Regular-SPMD models coordinate the fault detection and recovery function for the entire MPI/FT application, and therefore, are known as *coordinator* threads to distinguish them from other SCTs.

4.1. Fault detection and notification

Detecting process failures is a key feature of MPI/FT. A good fault detection strategy must balance accuracy (i.e., minimize false positives), speed (i.e., minimize the time interval between the occurrence and detection of a fault), and cost (i.e., impose low overheads when faults are not occurring). The fault detection mechanism in MPI/FT is apparently the largest contributor to the total fault free overhead. To limit this overhead, the current implementation of MPI/FT is restricted to detecting unresponsive MPI processes; user-selected runtime support for other kinds of fault detection and correction mechanisms (e.g., application and message replication) may be integrated into MPI/FT in the future.

Unresponsive processes are detected through the use of *internal* and *external* heartbeat messages generated and monitored by the various MPI/FT progress and self-checking threads. The SCTs use the internal heartbeats to monitor the progress of the send and receive threads in the MPI/FT

processes. The coordinator uses external heartbeats to monitor and gather progress information collected by remote SCTs.

4.1.1. Internal heartbeats
The SCT and progress threads communicate using shared memory to perform internal heartbeat monitoring. The SCT periodically "posts messages" to the progress threads requesting status updates. The mechanisms for delivering these requests take a variety of forms depending on the underlying communication layer used. Under Ethernet/TCP, for example, a receive thread blocked on a *select* call is awoken by a message sent by the SCT to the receive thread over a special intra-process socket. A send thread blocked on a semaphore waiting for new MPI messages to transmit is awoken when then SCT signals the event object. After posting the request, the SCT blocks on a semaphore, waiting for the progress thread to update the status information in the shared memory area. The progress thread signals the SCT's semaphore after writing status information in the shared memory area.

If the SCT does not receive the event within an interval (specified by the user in the form of an *mpirun* parameter), the SCT assumes that the progress thread has failed. It is possible that while processing large communication requests, the progress threads will not be able to respond to SCT status requests in time, thereby causing false failure indications. To prevent this situation from occurring, a progress thread that is about to process long-running communication operations writes the start time and expected duration of the period during which the progress thread will be unable to respond to SCT requests in the shared memory area. The SCT uses this information to extend the deadline for reception of the heartbeat from the progress thread in question.

4.1.2. External heartbeats
In *Model-Ia*, the coordinator passively listens for periodic heartbeat messages from the slave processes. If the heartbeat is not received from a slave process within an interval (specified in the form of an *mpirun* parameter), the coordinator assumes that process has failed and initiates recovery. In the current implementation of *Model-IIa*, the coordinator actively requests the SCTs at the other processes to provide status information, and expects to receive responses within user-specified intervals. Receipt of the request at the SCT indicates that the coordinator was unaware of any failure at any other process at the time the request was sent. As soon as the coordinator detects a failure, it broadcasts a failure notification to all surviving processes. Receipt of failure notification enables the SCTs to initiate recovery procedures and to update local copies of the global communicators.

Model-Ia requires only one-way SCT heartbeat communication because slaves are not notified of failures at the Master or other slave processes. Failure of the Master requires manual application restart. Because a slave's communicator only has itself and the master process, slave communicators need not be repaired to reflect failure of a slave.

Investigation of other techniques for heartbeat monitoring and failure notification (e.g., hierarchical organization of coordinators and "gossiping" [8,27]) to mitigate the bottleneck cause by the currently implemented centralized monitoring is ongoing for the remaining application execution models.

4.2. Recovery

In all models, the coordinator and SCTs collectively recover the internal state of the MPI implementation at the behest of the application once it is notified of the failure. Failed processes are replaced with processes taken from a pool of processes held in reserve.[1] Once the pool of reserve processes is exhausted, MPI/FT can no longer recover a failed process and the application fails. In the Master–slave models, only the Master process is notified of the failure of slaves. In *Model-Ib* and *Model-Ic*, the slaves' SCTs are responsible for collectively establishing a new Master process when the Master node fails. In the Regular-SPMD models, the coordinator is responsible for disbursing failure notification to all the processes in the MPI/FT application.

In *Model-IIb* and *Model-IIc*, the SCTs collectively establish a new coordinator and rank 0 if the process with rank 0 fails. The following sections describe the implementation of the recovery processes for *Model-Ia* and *Model-IIa* (efficient recovery processes for other models are currently under investigation). Replacing processes in the reserve pool (once exhausted) or having no reserve until needed will be offered in conjunction with MPI-2 in the future.

4.2.1. Recovery in Model-Ia
A failed slave process is recovered in *Model-Ia* by replacing the failed process with a process from the reserve pool. This is accomplished by replacing the descriptor for the failed process with the replacement process in the Master's communicators. Figure 1 depicts the following notification and recovery steps taken in *Model-Ia*:

1. The Coordinator detects failure of a slave and updates internal failure status data structures. All outstanding MPI calls using the failed process return failure error codes and message queues associated with the failed process are cleared. Furthermore, subsequent MPI calls using the failed process also fail until recovery is initiated. The application thread can also call a new API function `MPIFT_GET_DEAD_RANKS()`, in order to poll for failure notification.
2. The application thread calls a new API function `MPIFT_RECOVER_RANK()` in order to initiate recovery.
3. The Coordinator releases the replacement process. The replacement process is given the information required to reconstruct an MPI communicator.
4. The application and coordinator threads wait for the replacement process to complete the MPI progress and SCT connections. When the connection is completed, the replacement processes' descriptor replaces the failed

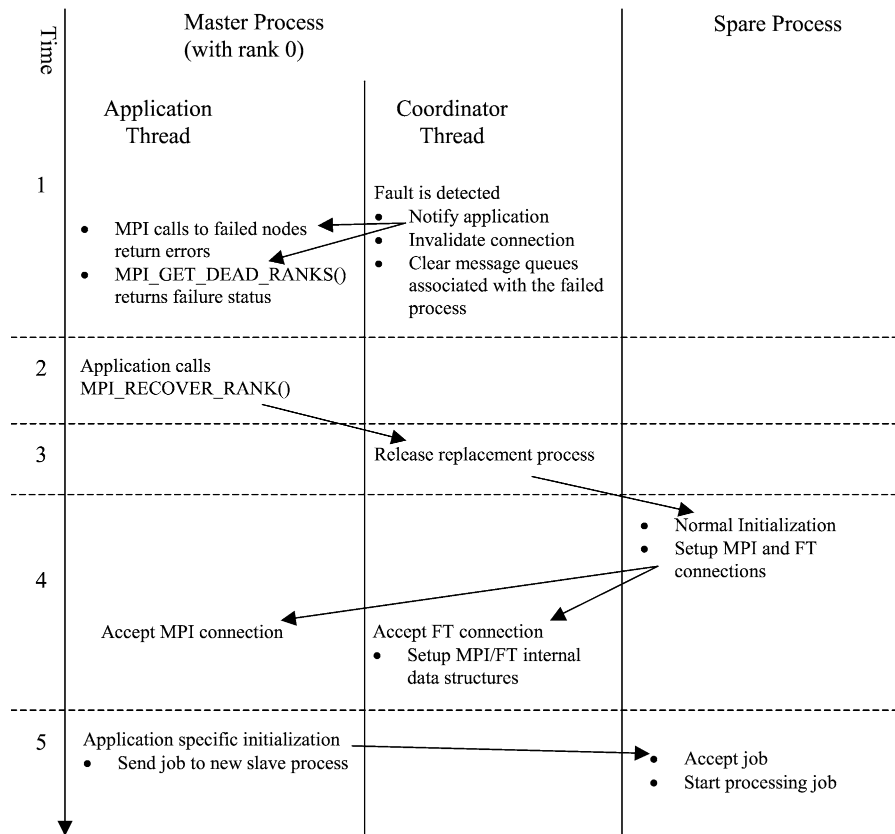

Figure 1. Implementation of notification and recovery in *Model-Ia*.

processes' descriptor in the Master's communicator. This marks the end of middleware recovery.

5. The application threads proceed with application-level recovery and process state initialization. This typically requires the Master to resend the job request being processed by the failed process to the replacement process.

Note that the other slaves are not affected and can continue processing while the Master process recovers the failed process.

4.2.2. Recovery in Model-IIa

In *Model-IIa*, the application cannot make progress when even a single process fails. Therefore, recovery is a collective operation involving all remaining processes. Figure 2 presents a depiction of the following notification and recovery steps taken in *Model-IIa*:

1. The coordinator detects failure of a slave and updates internal failure status data structures. The coordinator informs all other SCTs of the failed process. Upon reception of the failure notification, all outstanding MPI calls to all processes return failure error codes. Essentially, all communication is halted in order to ready the application for global recovery. Furthermore, all subsequent MPI calls also fail until recovery is initiated. The application can also call `MPIFT_GET_DEAD_RANKS()` to poll for failure notification.
2. The application threads at all processes call `MPIFT_RECOVER_RANK()` to initiate recovery.
3. Message queues at all processes are cleared.
4. The Coordinator releases the replacement process. The replacement process is given the information required to reconstruct an MPI communicator.
5. The application and SCT threads at all processes wait for the replacement process to complete the MPI progress and SCT connections. When the connection is completed, the replacement processes' descriptor replaces the failed processes' descriptor in the communicator at each process. This marks the end of middleware recovery.
6. Start the application-level recovery by loading and initializing application state from a previous checkpoint.

4.2.3. Checkpointing in Model-IIa

MPI/FT provides two new collective operations for application-level checkpointing and recovery. `MPIFT_CHKPT_DO()` is called by all ranks to save marshaled application-specific checkpoint data currently implemented as separate per-process files. This call successfully completes when all processes have saved the checkpoint data. `MPIFT_CHKPT_RECOVER()` is called by all processes in order to load the data from the last successful collective checkpoint. After loading the checkpoint data, the individual MPI processes unmarshal the data and restore the process's state before pro-

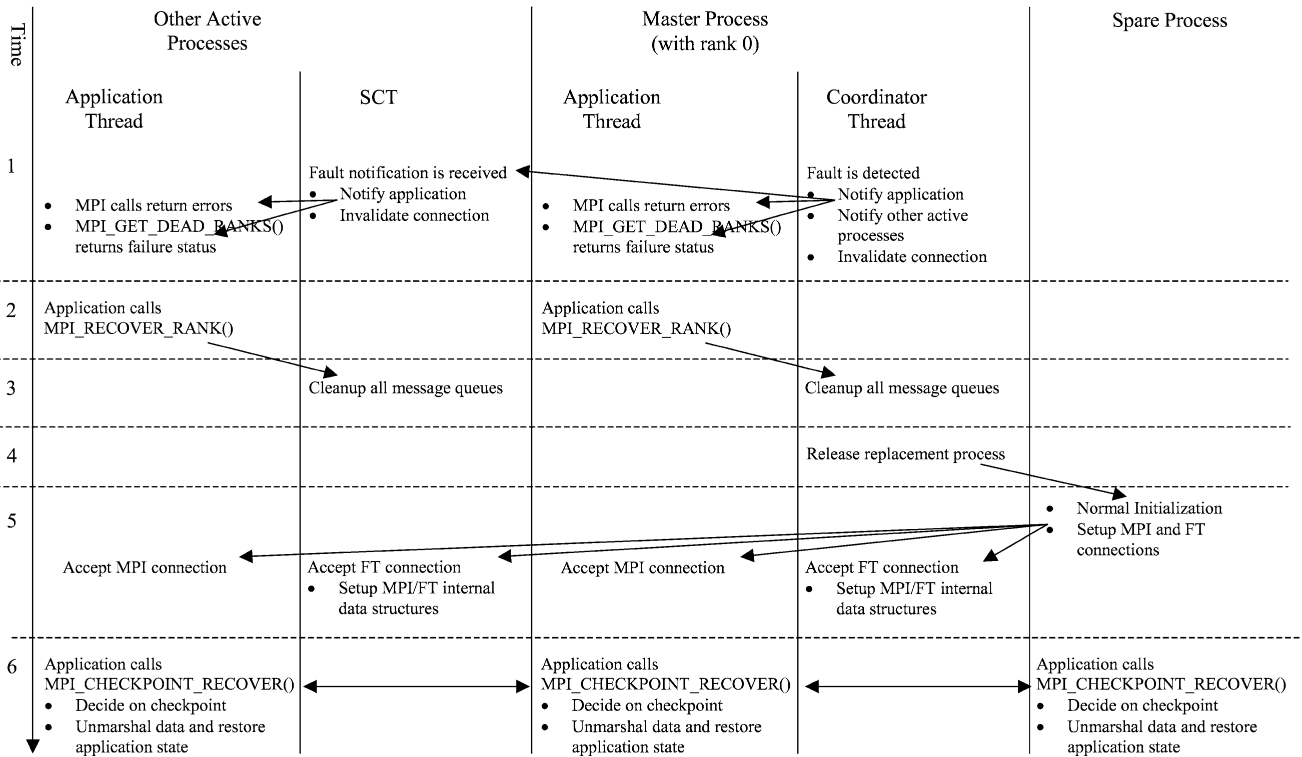

Figure 2. Implementation of notification and recovery in *Model-IIa*.

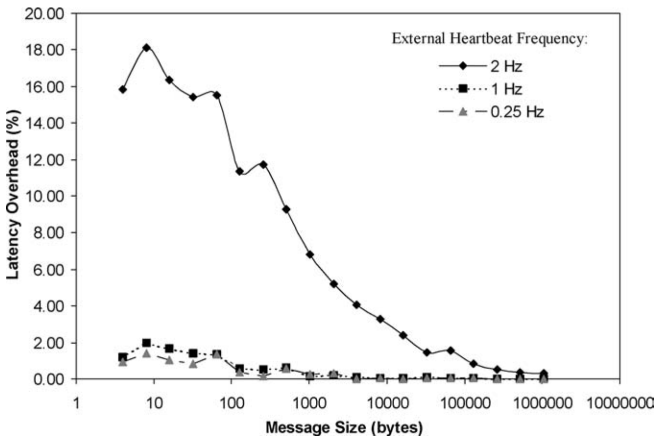

Figure 3. Increase in latency with external heartbeats only (internal heartbeats disabled).

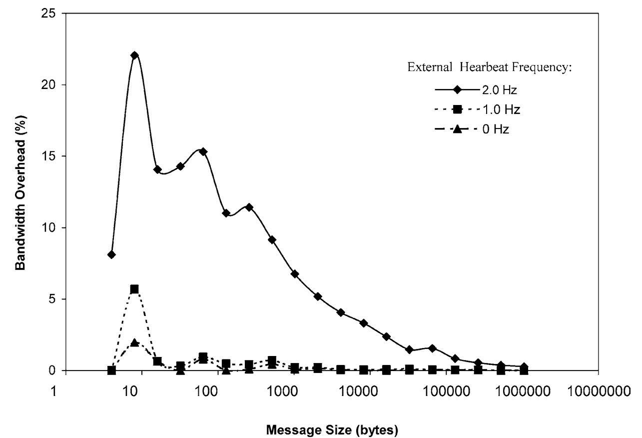

Figure 4. Decrease in bandwidth with external heartbeats only (internal heartbeats disabled).

ceeding. It is possible for processes to fail during the checkpoint recovery phase. In this case, the application-level recovery will fail and the application must repeat the process and application-recovery steps.

5. Experimental results

Latency and bandwidth are typically used for evaluating the performance of a message-passing middleware. The introduction of fault tolerance mechanisms increases the overhead inherent in any MPI implementation. This increase in overhead in MPI/FT results in decreased effective bandwidth and increased latency compared to MPI/Pro. In order to determine the additional overhead contributed by the fault tolerance mechanisms of MPI/FT accurately, a number of latency and bandwidth measurements were taken from a "ping-pong" test program under varying heartbeat parameters in a fault free environment. These measurements were compared to the latency and bandwidth measurements taken from the same ping-pong program using MPI/Pro. Most of the following results were achieved on a cluster of 750 MHz Intel Pentium III processors running Linux 2.4 and connected by 100 Mbps switched Ethernet. Results plotted in figures 7, 8, 9, and 10 were derived from a cluster of 900 MHz Pentium III processors.

Figures 3 and 4 plot the percent increase in latency and percent decrease in bandwidth, respectively, with disabled inter-

nal heartbeats and external heartbeats of varying frequencies over a range of message sizes. Figures 5 and 6 plot the percent increase in latency and percent decrease in bandwidth, respectively, with varying internal heartbeat frequency and one external heartbeat every 4 seconds over a range of message sizes. As expected, these graphs show that overhead increases with increasing frequency of heartbeats and that the impact of the additional overhead is reduced with increasing MPI message sizes. Therefore, the fault free overhead of point-to-point communication in MPI/FT is negligible in long-running applications using heartbeat rates in the order of tens of seconds.

In addition to latency and bandwidth, total completion time of applications is also an important criterion for comparing the performance of MPI/Pro and MPI/FT. Figures 7–10 compare the total completion time for *pmandel* (a *Model-Ia* implementation of the Mandelbrot Set [19] visualization program) running with three slaves (four processes in all) under the following situations:

(a) fault free MPI/Pro,

(b) fault free MPI/FT,

(c) single slave failure after 10% of the pixels are computed and recovery is not initiated (i.e., after the failure of the slave, the application continues with the remaining two slaves), and

(d) single slave failure after 10% of the pixels are computed and recovery is performed.

Clearly, the benefit of performing recovery depends on when in the application's lifecycle the failure occurs. An application may decide not to perform recovery (i.e., to proceed without the failed node) if the application is nearly complete and the recovery process will either provide no benefit or extend total completion time. In the *Model-Ia pmandel* implementation investigated in this research, recovering from a checkpoint is only beneficial if the failure occurs before 50% of the pixels have completed. The overhead of recovering

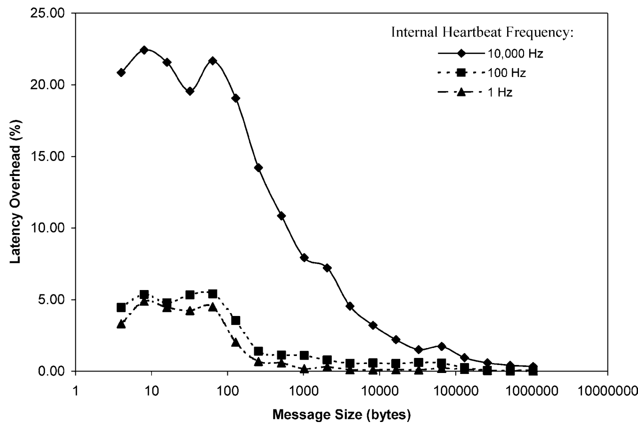

Figure 5. Increase in latency with varying internal heartbeat frequency and one external heartbeat every 4 seconds.

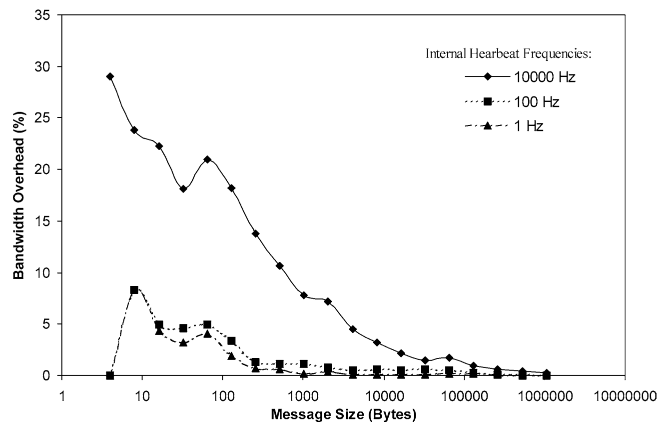

Figure 6. Decrease in bandwidth with varying internal heartbeat frequency and one external heartbeat every 4 seconds.

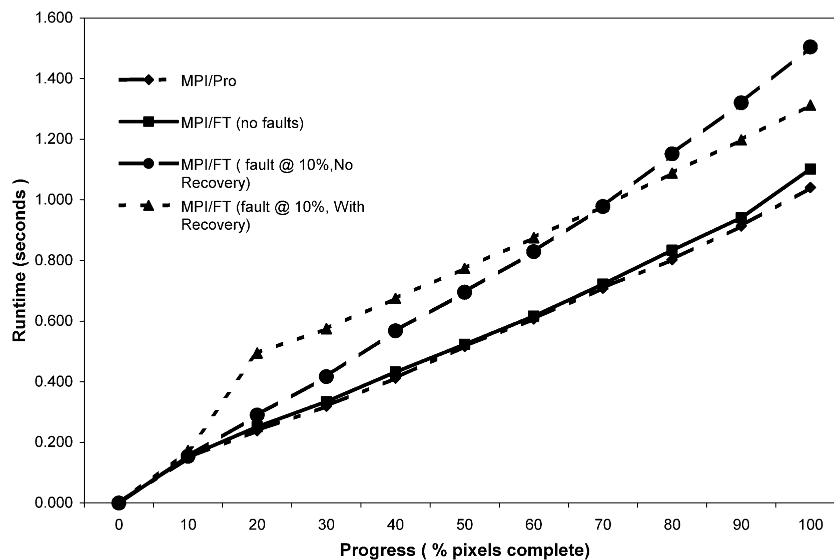

Figure 7. Comparison of runtimes for the *Model-Ia pmandel* application with and without recovery when the error occurs after 10% of the pixels have been completed.

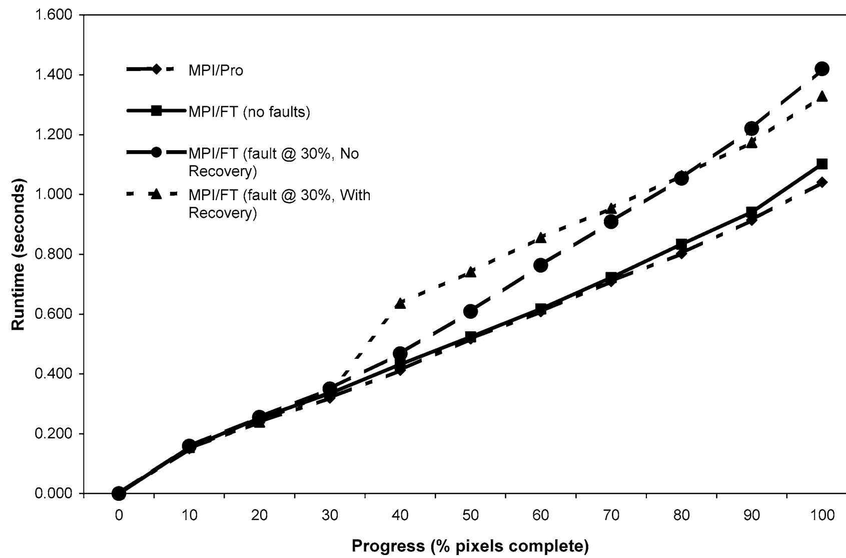

Figure 8. Comparison of runtimes for the *Model-Ia pmandel* application with and without recovery when the error occurs after 30% of the pixels have been completed.

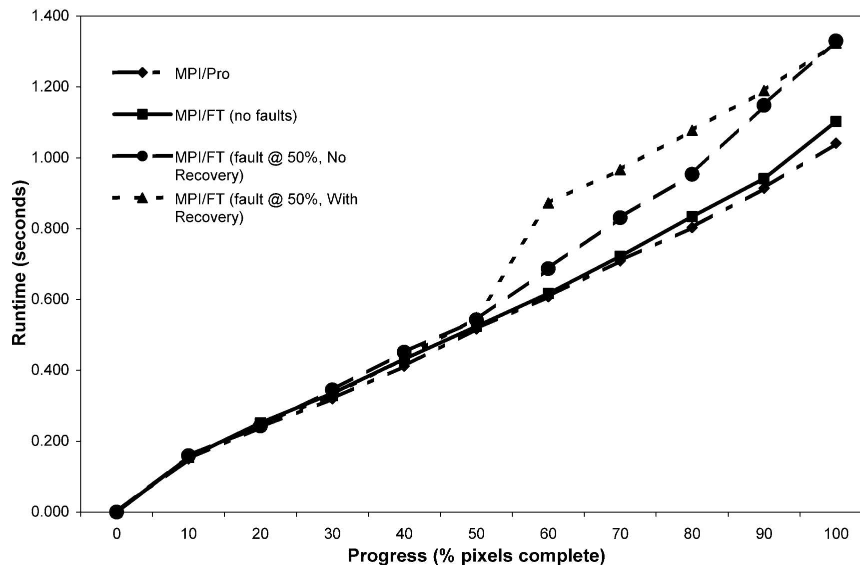

Figure 9. Comparison of runtimes for the *Model-Ia pmandel* application with and without recovery when the error occurs after 50% of the pixels have been completed.

Table 2
Runtime overhead for *Model-IIa* (Game of Life) without checkpointing over 10,000 iterations.

Grid size	MPI/Pro	MPI/FT	Overhead
4×4	8.84 s	9.26 s	4.7%
16×16	8.97 s	9.43 s	5.2%
100×100	10.63 s	11.17 s	5.0%
250×250	20.75 s	21.25 s	2.4%

from checkpoints after 50% of the pixels have completed extends the total completion time.

Table 2 compares the fault free runtime for MPI/Pro and MPI/FT for the *Model-IIa* sample application using four processors. The sample application is the parallel implementation of the cellular automata simulation “Game of Life” [4] and is executed over a 2D grid of four processors arranged in a 2×2 square topology. The timing results were obtained by executing the simulation for 10,000 iterations. As expected, checkpointing overhead increases with frequency of checkpoints taken. The cost of recovery from failure is similar to that in *Model-Ia*, and therefore, timing results from *Model-IIa* recovery experiments are omitted for brevity.

Checkpointing adds considerable overhead to the total runtime of an application and the MPI/FT user must determine the frequency of checkpointing appropriate for the particular application and its environment. Invoking checkpoint routines at a greater frequency than required leads to unnecessary fault free overhead. A lower than required frequency, in the presence of faults, can result in an increase in the amount of work that is repeated. Both these conditions retard or prevent

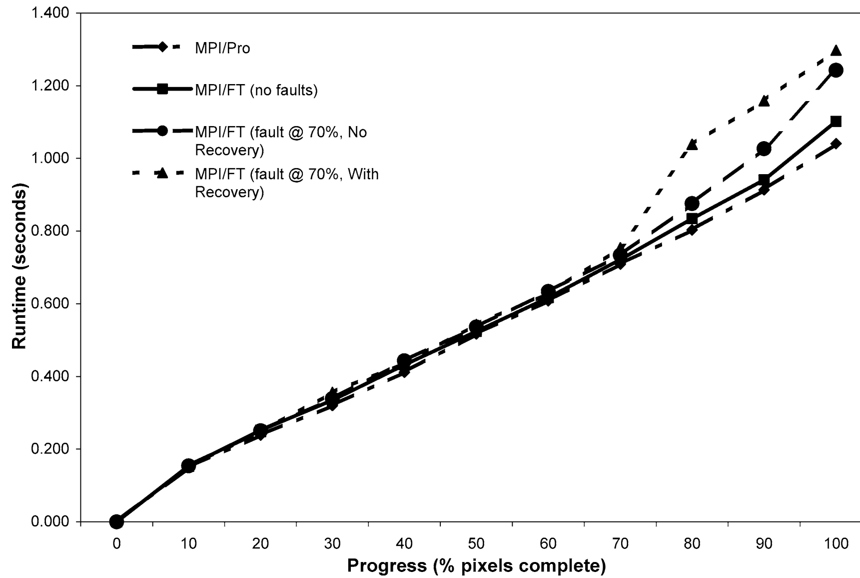

Figure 10. Comparison of runtimes for the *Model-Ia pmandel* application with and without recovery when the error occurs after 70% of the pixels have been completed.

application progress. Figure 11 illustrates the overhead resulting from taking checkpoints at various frequencies using the Game-of-Life *Model-IIa* example application.

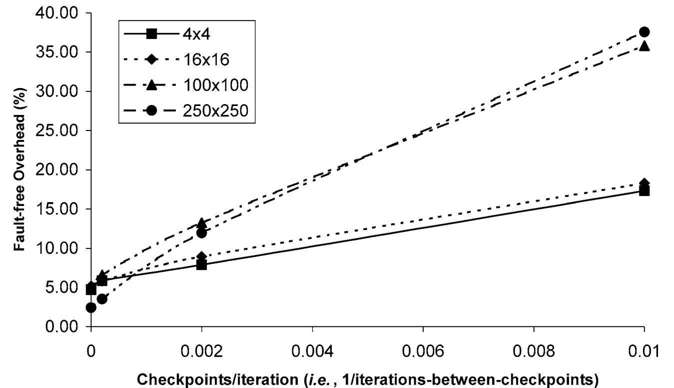

Figure 11. Checkpoint induced fault-free overhead in the *Model-IIb* "Game of Life" simulation of varying sizes.

6. Future work

This version and prototype of MPI/FT is practical, but just scratches the surface of potential scalable, fault tolerant, message passing keyed to practical production computing. Future work includes providing support for all models described in table 1 (additional models are also being investigated); connection of MPI/FT's checkpoint to MPI-2 parallel I/O; combination of MPI/FT with n-modular redundancy and redundant messages as initially reported in [3]; automation of decision to recover or not based on application input for *Model-I*; network failure detection and recovery for Ethernet/TCP, Myrinet [6], and Infiniband [15]; and combination with "gossip" [8,27] for scalable failure detection and notification.

7. Conclusions

This paper introduces model-based fault tolerance for parallel applications based on MPI 1.2. Model-based restrictions of syntax and semantics are effective means to focus on fault tolerance practically, while separating concerns that occur for different kinds of MPI-based applications. Specific models offer to lower fault-free overhead, enhance scalability, and promote practical solutions. A specific model taxonomy is given in table 1 and further extensions to this taxonomy are under investigation.

A fault tolerant MPI for two models – Master–slave and Regular-SPMD – is offered. This prototype, MPI/FT, is based on the MPI/Pro 1.51 commercial software package. MPI/FT successfully implements *Model-Ia* and *Model-IIa*, and offers specific measures of fault free overhead. MPI/FT is beneficial for the example applications shown. An initial API for recovery and checkpointing is also provided, but the formulation of the same is a subject for future research and development.

The initial success with model-based fault tolerance and internal monitoring of MPI for process faults provides a useful, pragmatic, but as-yet incomplete fault tolerant environment for production computing. Future work, noted above, is keyed to further advances in robustness and functionality.

Acknowledgements

The work of R.B., Y.S.D. and A.S. was supported in part by contract 1221287 from NASA Jet Propulsion Laboratory, California Institute of Technology. The work of A.S. and M.B. was supported in part by contract 1219475 from NASA Jet Propulsion Laboratory, California Institute of Technology.

Note

1. Because idle spares use few cycles and small virtual memory, this approach seems workable for small–medium clusters.

References

[1] A. Agbaria and R. Friedman, Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations, in: *Proc. 8th IEEE Int. Symposium on High Performance Distributed Computing* (IEEE CS Press, Los Alamitos, CA, 1999) pp. 167–176.

[2] S. Balay et al., PETSc users manual, Tech. report No. ANL-95/11, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL (1995).

[3] R. Batchu et al., MPI/FT(tm): Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing, in: *Proc. 1st IEEE Int. Symposium of Cluster Computing and the Grid* (IEEE CS Press, Los Alamitos, CA, 2001) pp. 26–33.

[4] E.R. Berlekamp, J.H. Conway and R.K. Guy, *Winning Ways for Your Mathematical Plays*, Vol. 1, *Games in General* (Academic Press, New York, 1982).

[5] L. Birov et al., PMLP home page, PMLP Web page; http://hpcl.cs.msstate.edu/pmlp (current August 2002).

[6] N. Boden, D. Cohen, R. Felderman, Al Kulawic, C. Seitz, J. Seizovic and W. Su, Myrinet: A gigabit-per-second local area network, IEEE Micro 15(1) (1995).

[7] R. Buyya (ed.), *High Performance Cluster Computing: Architectures and Systems*, Vol. 1 (Prentice Hall, Upper Saddle River, NJ, 1999).

[8] D.E. Collins, A. George and R. Quander, Achieving scalable cluster system analysis and management with a gossip-based network service, in: *Proc. 26th Annual IEEE Conference on Local Computer Networks* (IEEE CS Press, Los Alamitos, CA, 2001) pp. 49–58.

[9] R.P. Dimitrov, Overlapping of communication and computation and early binding: Fundamental mechanisms for improving parallel performance on clusters of workstations, Ph.D. thesis, Department of Computer Science (May 2001); http://library.msstate.edu/etd/show.asp?etd=etd-04092001-231941.

[10] R.P. Dimitrov and A. Skjellum, An analytical study of MPI middleware architecture and its impact on performance, Parallel Computing (November 2002), submitted.

[11] E.N. Elnozahy, D. Johnson and Y.M. Yang, A survey of rollback-recovery protocols in message passing systems, Tech. report TR98-1662, Computer Science Department, Cornell University, Ithaca, NY (1998).

[12] G. Fagg and J. Dongarra, FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world, in: *Proc. 7th European PVM/MPI Users' Group* (Springer, Berlin, 2000) pp. 346–353.

[13] M. Hayden, The ensemble system, Tech. report TR98-1662, Computer Science Department, Cornell University, Ithaca, NY (1998).

[14] K.H. Huang and J.A. Abraham, Algorithm-based fault tolerance for matrix operations, IEEE Transactions on Computers 33 (December 1984) 518–528.

[15] Infiniband; http://www.infinibandta.org (current October 2002).

[16] D.L. Katz et al., Applications development for a parallel COTS spaceborne computer, in: *Proc. 3rd Annual Workshop on High-Performance Embedded Computing* (Lincoln Laboratory, Lexington, MA, 1999).

[17] Y. Ling, J. Mi and X. Lin, A variational calculus approach to optimal checkpoint placement, IEEE Computer 50(7) (2001) 699–708.

[18] M. Litzkow et al., Checkpoint and migration of UNIX processes in the Condor distributed processing system, Tech. report No. 1346, Department of Computer Science, University of Wisconsin-Madison, Madison, WI (1997).

[19] B.B. Mandelbrot, *The Fractal Geometry of Nature*, 2nd edn. (Freeman, San Francisco, CA, 1982).

[20] Message Passing Interface (MPI) Forum, *Message Passing Interface Standard 1.1*, MPI Forum, http://www.mpi-forum,ord/docs/mpi-11.ps.Z (current September 2002).

[21] Message Passing Interface (MPI) Forum, *Message Passing Interface Standard 2.0*, MPI Forum, http://www.mpi-forum.ord/docs/mpi-20.ps.Z (current September 2002).

[22] MPI Software Technology Inc., MPI/Pro(tm) for Linux, Product pages, http://www.mpisofttech.com/products/mpi_pro_linux/default.asp (current September 2002).

[23] MPI Software Technology Inc., VSI/Pro(tm) products page, Product pages, http://www.mpi-softtech.com/products/vsipro/default.asp (current September 2002).

[24] V.P. Nelson, Fault-tolerant computing: Fundamental concepts, IEEE Computer 23(7) (1990) 19–25.

[25] J.S. Plank and W.R. Elwasif, Experimental assessment of workstation failures and their impact on checkpointing systems, in: *Proc. 28th Int. Fault-Tolerant Computing Symposium* (IEEE CS Press, Los Alamitos, CA, 1998) pp. 48–57.

[26] S. Rao, L. Alvisi and H.M. Vin, Egida: An extensible toolkit for low-overhead fault-tolerance, in: *Proc. 29th Int. Fault-Tolerant Computing Symposium* (IEEE CS Press, Los Alamitos, CA, 1999) pp. 48–55.

[27] R. Renesse, Y. Minsky and M. Hayden, A gossip-style failure detection service, in: *Proc. of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)* (1998).

[28] G. Stellner, CoCheck: Checkpointing and process migration for MPI, in: *Proc. 10th Int. Parallel Processing Symposium* (IEEE CS Press, Los Alamitos, CA, 1996) pp. 526–331.

[29] S.J. Wang and N.K. Jha, Algorithm-based fault tolerance for FFT networks, IEEE Transactions on Computers 43(7) (1994) 849–854.

[30] K.F. Wong and M. Franklin, Checkpointing in distributed systems, Journal of Parallel and Distributed Systems 35(1) (1996) 67–75.

Rajanikanth Batchu is M.S. graduate from the Department of Computer Science and Engineering at Mississippi State University in 2003. He is currently completing an internship in the telecommunications industry. This student research included fault-tolerant and security aspects of high performance clusters and interfacing issues of parallel mathematical libraries. His current interests include scheduling in real-time systems, practical solutions of dispatching systems and design issues of fault-tolerant middleware. He received his Bachelors degree in computer science and engineering from Osmania University, Hyderabad, India in 1999.

Yoginder S. Dandass is an Assistant Professor in the Department of Computer Science and Engineering at Mississippi State University. He received his Ph.D. degree from Mississippi University in 2003 and M.S. degree from Shippensburg University in 1996. His research interests include real-time scheduling, operating systems, fault-tolerant and high performance message-passing middleware, and computer security. Prior to joining Mississippi State University, Mr. Dandass worked as an information technology consultant from 1989 to 1997.

Anthony Skjellum is a Ph.D. graduate of the California Institute of Technology in 1990 (after completing B.S. and M.S. degrees at Caltech in 1984 and 1985, respectively). His multidisciplinary Ph.D. work was in concurrent dynamic simulation, in chemical engineering with computer science minor, included work on message-passing systems. This work contributed to the MPI-1 and MPI-2 standards, and Dr. Skjellum's group initiated the first implementation of MPI – MPICH – with Argonne National Laboratory in 1994. Dr. Skjellum subsequently worked at the Lawrence Livermore National Laboratory in high performance computing from 1990 to 1993, and was a Professor of computer science and engineering from 1993 to 2003 at Mississippi State University. He is currently a Professor and Chair of the Department of Computer and Information Sciences at the University of Alabama at Birmingham. Dr. Skjellum has also spun-off MPI Software Technology, Inc. (now Verari Systems Software, Inc.), a high performance middleware company in 1996. His on-going research includes message-passing, mathematical libraries, QoS for parallel computing, and component-object-aspect-model-based high performance computing.

Murali Beddhu is currently an Assistant Professor at the University of Southern Mississippi. He received his Ph.D. in general engineering from Mississippi State University in 1992. His research interests include algorithms and numerical schemes for numerical grid generation, computational fluid dynamics, and machine vision. His work experience includes eight years in developing advanced CFD algorithms as a Research Engineer at the CFD Lab at Mississippi State University, focusing on free surface and rotating flows, and nearly two years as a Senior Software Engineer at MPI Software Technology Inc., focusing on middleware related topics. Dr. Beddhu was also a part-time lecturer in the Department of Aerospace Engineering at Mississippi State University. He received his M. Tech. in aerospace engineering from Indian Institute of Technology, Kanpur, India in 1988, and his B. Tech. in aeronautical engineering from Madras Institute of Technology, Madras, India in 1986.