SAND2013-9102C

Design and Evaluation of FA-MPI, A Transactional Fault-Tolerant MPI

Amin Hassani, Anthony Skjellum
University of Alabama at Birmingham
1300 University Blvd
Birmingham, AL, 35205
{ahassani,tony}@cis.uab.edu

Ron Brightwell
Sandia National Laboratories
PO Box 5800
Albuquerque, NM 87185-1319
rbbrigh@sandia.gov

Abstract—**It has been predicted that the rapid growth in the size of large scale supercomputers will increase the frequency of failures in such systems. Advanced fault-tolerant methods have evolved to adapt to this high rate of failures, but the behavior of MPI, as the most common communication middleware, is insufficient when confronting these failures. We designed FA-MPI (Fault-Aware MPI) as a set of new extensions to the MPI standard to allow applications to implement a wide range of fault-tolerant methods for their applications. FA-MPI introduces transactions to MPI for the first time, in order to address failure detection, isolation, mitigation, and recovery via application-driven policies. In order to reach the maximum achievable performance of these scalable machines, overlapping communication and I/O with computation through non-blocking operations is necessary. We leverage non-blocking communication operations combined with a set of lightweight transactional *TryBlock* API extensions that can be nested to support multi-level failure detection and recovery. The goal is to support fault-awareness in MPI objects and enable applications to run to completion with higher probability than running on a non-fault-aware MPI. Scalability and fault-free overhead are key concerns and can be managed through changing transaction granularity. In this paper we demonstrate the application of FA-MPI in a simple one dimentional "Game of Life" program. Experimental results with different rates of failure and checkpointing are evaluated. Failure models supported by FA-MPI include but are not limited to process failures, unlike other proposed systems.**

I. Introduction

Resilience is one of the most significant challenges facing large-scale scientific computing. Machines continue to grow in size to achieve higher levels of compute performance, and without significant increases in hardware reliability, platforms will be prone to increased rates of failures. It is unclear that existing strategies for tolerating such failures will continue to be effective. One of the basic tenets of fault-tolerance is replication in "time" and "space." Such strategies have evolved from fault-tolerance in a single machine, to distributed and parallel systems, but programming environments implementing these approaches have largely remained unchanged. An example is message passing middleware based on the Message Passing Interface (MPI) [1]. MPI was initially designed to achieve high performance in parallel systems without regard to fault-tolerance. The MPI model assumes a reliable communication layer in which processes are always able to communicate once they become known. Any failure management is considered the responsibility of the application or the underlying MPI library implementation. The MPI programming interface has no fault-management capability or semantics. MPI does support registration of an error handler on a per-communicator rather than per-function basis to allow applications to recognize and potentially respond to local errors, such as library resource exhaustion or invalid function arguments. Many studies have revealed the impact of fault-tolerance in MPI based on different methodologies like checkpoint/restart [2]–[4], pessimistic logging [5], casual logging [6], optimistic logging [7], and algorithm based fault-tolerance [8] implemented in different levels of the software stack. Some approaches require collaboration of user application while others are transparent. However, existing environments require the engagement of hardware and every layer of the software stack to collaborate for failure management, and the message passing middleware plays an important role.

Adding fault-tolerance support to the MPI Standard has been an important topic in the past few years. The MPI Forum's Fault-Tolerant Working Group (FTWG) has been developing and considering proposals that extend the MPI interface and provide semantics that support fault-tolerance. Two recent proposals are Run-Through Stabilization (RTS) [9] and, the more recent, User-Level Failure Mitigation (ULFM) [10].

In this paper, we present a new standalone approach for fault-tolerance in MPI called Fault-Aware MPI or FA-MPI [11]. The goal of FA-MPI is to extend MPI minimally to support a lightweight transactional model for fault-awareness[1] in message passing middleware. FA-MPI provides the ability to execute a series of operations, wait for them to complete, and disseminate information about any operation failures. This approach allows for the application to control the appropriate granularity of transactions. FA-MPI detects operation failures and broadcasts information about errors through faul-tolerant collective communication functions and provides notification of local and non-local failures to the application via request handles and other MPI objects, such as communicators. For recovery purposes, failures are treated like operation state that can be queried for success. An application can choose to isolate and mitigate failures with the help of FA-MPI by creating smaller communicators, replacing broken communicators with new ones, and then try to recover from the failed state. FA-MPI does not provide any failure recovery policy or semi-automation. The approach is similar to the BSP [12] model,

[1]Throughout the paper we use fault-tolerance and fault-awareness interchangeably, but we leverage the concept of fault-awareness as the first step toward resilience and, ultimately, fault-tolerance.

where asynchrony is reached in "epochs" and barrier synchronization allows the application to enter a known good state. FA-MPI allows different rates of faults to appear in the system, and, instead of masking failures, the application can set the granularity of transactions to handle different failure rates.

Unlike other approaches, the goal of FA-MPI is not to ensure the healthy state of the MPI library (active fault-tolerance) in order to provide continuous operation. Fault-awareness allows MPI to make the application aware of failures, and, consequently, fault-tolerance can be achieved through the application's use of other approaches, including checkpoint/restart, message logging, or ABFT. However, to achieve this level of fault-tolerance in the application, the MPI library itself should be resilient to faults and be able to continue operation after certain failures and helps application to recover.

The remainder of this paper is organized as follows. In the next section, we provide background information on transaction-based fault tolerance. Section III describes the design of FA-MPI, followed by several examples using FA-MPI which are presented in Section IV. In Section V, we present performance results that illustrate the costs of our approach. Relevant related work is discussed in Section VI, and we provide a summary and description of future work in Section VII.

II. Background

Distributed database systems [13] have used transactional models to ensure consistency, integrity, and fault-tolerance for several decades. The goal of a transaction is to ensure data consistency in a system by allowing a mutual agreement between all participants either to accept (commit) or reject a series of changes in system. While in distributed database systems the transaction commit operation ensures data consistency and integrity in saved data, from the viewpoint of a message passing system, a transaction commit operation ensures consistency of a "communication" and/or a "computation" operation. This approach allows the higher-level application to perform a series of computation and communication operations (transactions) and ensure the successful completion of operations at transaction commit. In case of a failure (transaction reject), the application can determine whether or how to recover from a failed state. The transactional model allows applications to do a soft retry, rollback, roll-forward, or perform a restart of the application from a checkpoint if continuing execution is not possible. A transactional model allows *versioning* to be utilized to manage the healthy and failed states of data. Parts of internal data structure of MPI, like communicator objects, as well as other application data can be versioned. Versioning allows multiple snapshots of data to exist throughout the lifetime of the application execution. Versions of data can be kept in ephemeral or persistent storage to allow for various approaches to recovery, including soft retry and checkpoint/restart. The application can discard data that has been corrupted by a failed transaction and revert to a previous known good version.

In a non-distributed system, local failure detection and notification are the initial steps required for fault-tolerance. In a distributed or parallel system, error notification may need to be communicated throughout the system. FA-MPI proposes an approach for failure detection and remote notification inside a *TryBlock*, which will be described in more detail below.

Once the application has been notified of an error, it can take steps to try to continue operation through a recovery procedure. For example, if a transaction fails because of an error in the communication used in an MPI_Allreduce() operation, the communicator may still be used successfully for an MPI_Bcast() operation if, for example, the underlying point-to-point communication pattern avoids failed links or processes. Peer communication operations may still succeed on a communicator with failed ranks if no communication involving those ranks is performed. FA-MPI can help with spawning new processes and create a communicator for the failed ranks and continue point-to-point operations on both communicators. This isolation and mitigation approach allows an application to retry the TryBlock and perform only those operations that previously failed. To retry a failed collective operation, FA-MPI can shrink the failed communicator into a smaller size communicator with only healthy ranks, spawn new processes to form a new communicator, and then merge the two new communicators to form a new, healthy communicator. FA-MPI maintains the single-assignment properties of MPI objects (communicators, windows, and files). Repairing a failed communicator is not our approach. Rather, a failed communicator is replaced with a new, healthy one [2].

We consider recovery as another block of computation and communication that can be handled in a TryBlock even in the presence of faults. FA-MPI provides an environment for a successful multi-level recovery. Partial soft retry, complete soft retry, rollback, rollforward, and checkpoint/restart can be utilized based on application's decision and policies.

The cost of fault-tolerant approaches is an important consideration. Fault-free overhead is defined as the cost of running a fault-aware MPI application, in case of no faults, compared to the non-fault-aware version. We expect that applications using FA-MPI will be able to run longer on larger machines as compare to a non-fault-tolerant version of the application. In order to achieve resiliency, a sacrifice in performance cannot be avoided. We allow applications to run slightly slower, but with enough forward progress to reach the completion of execution more quickly. This is especially important given future exascale machines that may contain more lower-powered processors with higher failure rates as compared to current systems. We allow more failures to be detected by the system, and we use FA-MPI to manage these failures properly. FA-MPI allows the application to control the fault-free overhead by setting the granularity of synchronization systematically. Most of the fault-free overhead resides in transactional operations for starting and ending a transaction – specifically transaction commit, which is a synchronization collective call. We expect that in the failure-free case, the transaction commit performs similar to an MPI_Allreduce() operation with one integer. However, nominal MPI operations will have minimal or no overhead in a practical implementation. It is one of the main features of FA-MPI where failure is not checked/diagnosed/corrected right after each operation so minimal or no overhead is introduced in MPI operations.

[2]We did not provide API extensions for isolation and mitigation of failures in this paper.

III. FA-MPI Design

In this section, we describe the design of FA-MPI at the API level and we provide rationale for our design decisions.

A. *TryBlocks*

FA-MPI extends MPI with a transactional model designed to allow a series of operations to be "tried" and then "committed" when all operations succeed, or be "rolled backward" or "rolled forward" when some operations fail. TryBlock operations are fundamental building blocks upon which the FA-MPI model is based. TryBlock operations model a transaction block inside which several non-blocking communication, computation and I/O operations maybe executed.

MPI_TRYBLOCK_START(comm,flag,tryreq)

IN	comm	communicator (handle)
IN	flag	flag (integer)
OUT	tryreq	tryblock request (handle)

Each TryBlock starts with MPI_TryBlock_start(), which binds a communicator to its request handle. Any communicators (including the communicators associated with window and file objects) used inside a TryBlock should be a proper or improper subset of TryBlock's communicator's group. Violation of this requirement may not produce any error, but such a choice will not guarantee a successful fault-awareness mechanism since it will ignore ranks that are not in TryBlock's communicator's group.

Application might decide that only local errors are sufficient for failure recovery and no global synchronization is necessary. To extend the flexibility of fault-tolerance mechanisms, application can use the "flag" argument to notify the MPI implementation the need for global error dissemination and synchronization transparently or in transactional commit. MPI_TRYBLOCK_GLOBAL (default) and MPI_TRYBLOCK_LOCAL are defined to capture this functionality.

MPI_TRYBLOCK_IFINISH(tryreq,tout,nreq,reqs,stats)

IN	tryreq	tryblock request (handle)
IN	tout	timeout (handle)
IN	nreq	number of input requests (integer)
INOUT	reqs	array of requests (array of handles)
OUT	stats	array of statuses (array of status)

A TryBlock is completed (committed) by MPI_TryBlock_ifinish(). This function is a synchronizing non blocking collective operation that broadcast failures in a fault-tolerant allreduce/allgather over all ranks in the TryBlock's communicator's group. At the end of the transaction, ranks decide consistently to accept or reject the transaction by examining globally returned failures. TryBlock completion allows determination of faulty and failed objects, requests, ranks, and failures associated with each rank. This global knowledge allows the application to define policies to achieve resiliency with the help of FA-MPI. The non-blocking nature of MPI_TryBlock_ifinish() allows multiple TryBlocks to be executed simultaneously and an outer TryBlock sifts out failed and successful TryBlocks. Its request handle can be used to be waited or tested later.

FA-MPI doesn't restrict mechanisms used to implement the semantics of TryBlocks. Any implementation may use a consensus algorithm through piggybacking, gossip, collective, a hybrid algorithm, and/or other methods for broadcasting failure information to live ranks. Some recent publications [14] on implementation of the consensus problem can be used to synchronize failures in TryBlock completion. Transient error dissemination can be achieved using gossip [15] and/or piggybacking [16] either during TryBlocks execution or at the commit phase. Further study is needed to reveal the performance and reliability of transient error dissemination.

TryBlocks can be nested to support multi-level failure detection and recovery. The nested property of TryBlocks is required to achieve high scalability through application of data and task parallelism for smaller communicator groups and multiple user threads. An outer TryBlock can provide global application progress while several nested TryBlocks inside can run in parallel (in different user threads) or serially. The outer TryBlock can sift out successful and failed TryBlocks and progress the application forward or backward based on the recovery policy.

B. *Failure Detection*

Some failures are undetectable, transient, expensive to be detected, or cannot be detected at the moment of operation. Systems with different properties might encounter various types and degrees of faults. FA-MPI does not impose any restriction on failure detection and any local or global failure detector like Heartbeat [17] can be used. In addition, FA-MPI provides an environment which different types of faults in all layers of software and hardware stack can be detected and managed. Applications can help with failure detection through the concept of *failure injection*. FA-MPI proposes a failure injection mechanism at the application level inside TryBlock to allow application and the MPI implementation to collaborate consistently to detect and notify failures and resolve them with each other's help. Coordination can be done by allowing both the application and MPI library to detect and *raise* (inject) errors on requests inside a TryBlock. MPI should trust this information as it was detected by MPI itself. Raising errors on request implies raising error on MPI objects and ranks associated with the request.

This approach allows the application to use an ABFT approach, such as a checksum calculation on the result of a computation or communication, and simply notify other ranks about the failure in TryBlock completion. An MPI implementation can choose to detect any failure and save the associated error state among with the rank responsible in a data structure invisible to user. At least one portable error code will be provided and implementations will be permitted add additional error states specific to their system. All error injection API functions are local.

MPI_REQUEST_RAISE_ERROR(request,code)

IN	request	request with failure (handle)
IN	code	error code (integer)

A few error codes defined already are MPI_ERR_PROCESS_FAILED and MPI_ERR_REQUEST_FAILED. Other error codes can be defined by implementations.

C. Failure Dissemination

In the MPI programming model usually all ranks in a collective operation such as MPI_Allgather() or MPI_Allreduce() need to collaborate on results and even failure of one rank may result in incorrect data. In a parallel or distributed system where a group of processes are involved in executing a task, synchronization is required between the processes. So in addition to failure detection, consistent synchronization regarding failure is important.

Failure recovery decisions should be made locally and the result has to be consistent with all other ranks in the communicator. For example, different ranks in a communicator's group might have different failures associated with their requests and different recovery procedures might be needed for each type of failure. But after failure recovery MPI state in all ranks in a group should be consistent. This requires global error propagation so each rank can view a consistent state of other ranks to make a consistent decision.

Failure dissemination is a modestly heavy operation and not applicable if it needs to be checked after each operation. However, error states should be disseminated consistently. Dissemination of error states to all ranks in a TryBlock's group will be done at the TryBlock's finish call using a fault-tolerant Allgatherv/Allreducev protocol [14] implemented by a consensus agreement.The reason we call the implementation an Allgatherv/Allreducev is because it is a combination of Allgatherv and Allreducev. Each rank has a variable number of locally known failures. Failures can be error codes associated with other ranks so multiple ranks can give opinion on a particular rank. All failure codes from all ranks should be gathered and merged into a final list. This list will have variable length when failure rate in the system changes. At the end this list should be broadcasted to all ranks of the TryBlock's group for global knowledge of failures.

D. Failure Notification

Failures can be revealed to the user after TryBlock's finish call. Failure notification is a bottom-to-up approach and user should be aware of any failure in the system that cannot be recovered at that specific level. Querying for failure is a mechanism for user to retrieve information about local and global failures in the system. We intend to provide the "query for failure functionality" for the user. Different types of failures and associated objects can be queried.

In this experimental model, we provide simple query functions to notify the user of any local or global failure. After getting the list of failed requests, error state for each request shows the type of failure associate with the request. It can be either failure injected by user or failure in the underlying system. The API can also enumerate failed ranks and objects. Although this API is not finalized, our current concept offers a good start for further investigation.

MPI_GET_FAILED_REQUESTS(tryreq,max,count,indices)

IN	tryreq	tryblock's request (handle)
IN	max	maximum size of array indices (handle)
OUT	count	number of failed requests (integer)
INOUT	rank	array of indices to tryblocks's give requests (array of integer)

MPI_GET_FAILED_RANKS(tryreq,franks)

IN	tryreq	tryblock's request (handle)
OUT	franks	group of failed ranks (handle)

MPI_GET_FAILED_OBJECTS(tryreq,max,count,comms)

IN	tryreq	tryblock's request (handle)
IN	max	maximum size of array comms (handle)
OUT	count	number of failed communicators (integer)
INOUT	rank	array of failed communicators (array of handles)

E. Timeout

In future exascale systems, operations have to be marked eventually as complete (successful or failed) through some timeout mechanism. Timeout is an effective mechanism to handle exceptional behaviors, such as delay in response or remote failure. Communication layers normally use timeout as a mechanism for detecting failure. In order to add a fine grain of fault-tolerance in user space, timeout should be added to semantics of message passing middleware. FA-MPI uses timeout semantics to allow application variable granularity for trying (and failing) a transaction so user application does not require to wait "forever." Application can decide to wait infinitely by setting timeout value to zero. In this case operations work as now where there is no timeout. We propose that timeout granularity be no coarser than MPI_Wtick().

Timeout can be treated as not just an error but a warning to the application that this operation has not finished in the specified time. Application can decide to wait longer on operation to finish or utilize other approaches to somehow finish the operation.

MPI_TIMEOUT_SET_TICKS(timeout,ticks)

INOUT	timeout	timeout (handle)
IN	ticks	time out in ticks (integer)

MPI_TIMEOUT_GET_TICKS(timeout,ticks)

IN	timeout	timeout (handle)
OUT	ticks	time out in ticks (integer)

F. Local Completion

TryBlock completion calls need communication operation request handles to perform error detection and notification, but local completion functions like MPI_Wait() destroy request handles on successful return. This behavior is insufficient if the application needs completion of a communication request before the TryBlock's completion call and it needs to check the request's failure state and notify MPI. To take advantage of error notification to MPI implementation, request handles should not be freed at least until the TryBlock's completion call.

There are two solutions for this problem. The first approach is to define a new set of non-blocking completion APIs with two goals in mind. This new set of APIs resembles all MPI completion calls except that these functions do not destroy the request handle. In addition, timeout is used in all of these API as a means to cope with time related failures and deadlock prevention. Request handle can always be deleted using MPI_Request_free().

The reason we use request handles for TryBlock's finish function and not their MPI_Statuses is the fact that we need to check status of operations in progress and not just operations that already completed. For example a timeout error allows the application to wait more time for the operation to finish. If a request handle were deleted, there was no oportunity for application to wait longer for the operation to finish.

MPI_WAIT_LOCAL(request,status,timeout)

INOUT	request	communication request (handle)
OUT	status	status object (status)
IN	timeout	timeout (handle)

MPI_WAITANY_LOCAL(count,requests,index,status,timeout)

IN	count	list length (non-negative integer)
INOUT	requests	array of requests (array of handles)
OUT	index	operation that completed (integer)
OUT	status	status object (status)
IN	timeout	timeout (handle)

MPI_WAITALL_LOCAL(count,requests,statuses,timeout)

IN	count	list length (non-negative integer)
INOUT	requests	array of requests (array of handles)
OUT	statuses	array of status objects (array of status)
IN	timeout	timeout (handle)

MPI_WAITSOME_LOCAL(incount,requests,outcount,indices, statuses,timeout)

IN	incount	requests' list length (non-negative integer)
INOUT	requests	array of requests (array of handles)
OUT	outcount	number of completed requests (integer)
OUT	indices	completed operations indices (array of integer)
OUT	statuses	array of completed statuses (array of status)
IN	timeout	timeout (handle)

The second approach is simpler with fewer API calls, but requires changes in semantics of current MPI completion calls. In this approach, completion calls will free request handle after successful return as normal, but this approach still lacks the timeout semantics needed for failure detection. To solve this problem timeout can be attached to the request using the API below. Any completion call can return with MPI_TIMEOUT failure if timeout happens.

MPI_REQUEST_TIMEOUT_SET(request,timeout)

INOUT	request	request (handle)
IN	timeout	timeout value (handle)

G. Non-Blocking Communication

The path toward exascale and the need for scalable and dependable applications and libraries motivates the use of non-blocking communication calls in message passing systems to achieve higher performance through overlapping computation, communication, and I/O. Non-blocking semantics can be used to help achieve fault-tolerance in MPI applications. As an example, a communication operation should not hang because of a failure in a remote rank. MPI blocking communication operations halt in case of remote failure and a more critical problem with blocking operations is untraceability of the status of operation. As a programming model any resource or reference to a blocking operations is freed after returning from call. This means that there is no mechanism to track what happened in the operation except than an error code on return. However, returning from a blocking operation like MPI_Send() only guarantees that the data buffer can be used and there is no guarantee of data transfer to remote node neither any local failure notification. In other hand a non-blocking call uses a request handle and in concept of fault-tolerance this request handle can be used to save different error states at different stages of operation. This motivates FA-MPI to support only non-blocking communication calls and to ignore legacy blocking APIs. Non-Blocking operations are the most important part elaborated in FA-MPI mechanism. It is the basic idea behind how an MPI implementation and a user can collaborate for fault-tolerant and, specifically, fault-detection. Local errors can be found in a request completion function and global errors can be revealed at the transaction commit. Although non-communicating blocking operations like MPI_Comm_rank() is supported. We expect that tools will be developed to enhance applications with blocking operations automatically or semi-automatically to conform with FA-MPI's transactional fault tolerance capabilities.

H. One-Sided Semantics

One-sided semantics is provided in the MPI standard (since MPI-2) to allow applications to utilize the low latency and high bandwidth capability of RMA engines for remote memory operations. MPI has both blocking (MPI_Put(), MPI_Get(), and MPI_Accumulate()) and non-blocking (MPI_Rput(), MPI_Rget(), and MPI_Raccumulate()) one-sided data transfer operations and we only emphasize on non-blocking API. However, a real problem arises in target synchronization of one-sided semantics. MPI has "active" and "passive" forms of target synchronization. Active target synchronization requires both points of communication to be involved in the synchronization "epoch" while passive synchronization requires only the initiator be involved in synchronization explicitly. MPI does not specify any restriction on when data transfer should start inside a synchronization epoch. Implementations may decide to start data transfer at the end of synchronization. In addition, all synchronization calls are blocking and this violates the fundamental aspects of transactional fault-tolerance because there is no mechanism to track the failure state operations through synchronization functions. In other words, success or failure of synchronization functions should be a part of the failure detection mechanism of FA-MPI, but there is no non-blocking target synchronization API. In future we expect to provide solutions to this problem by either introducing non-blocking target synchronization API or proposing an alternative programming models in FA-MPI using only current versions of one-sided semantics.

I. Files and I/O

Transactional properties of FA-MPI allows I/O operations and files to behave as transactional activities. TryBlocks support files and local and global I/O failures can be reported to user. Many scenarios of I/O faults can be handled using TryBlock. TryBlocks allow full or partial recovery, for instance, in a TryBlock all reads on a file might succeed, but subsequent writes may fail and application can recover/retry the writes on file, or rendering collective I/O maybe invalid while point-to-point I/O still works. Users can employ a copy of a file portion for rollback purposes. Parallel file I/O systems that support shallow copy and copy-on-write semantics could also help support less than complete rollback I/O work. Applications can take advantage of a transactional distributed file system to perform multi-levels checkpointing and recovery. An error code validating that the underlying I/O subsystem cannot continue operating is provided.

IV. Application Examples

Many execution models of computation derived from communication topology and parallel programming semantics can be modeled using FA-MPI to apply fault-tolerance for successful execution. We show simple examples for master-worker and data parallel models. Pseudocode for a master and worker application is shown in Algorithm 1 and 2 respectively. The master can spawn jobs to workers using non-blocking Try-Blocks and handle each worker's failure individually without the need for global synchronization over all ranks. Workers receive a job from Master, compute it and return the results back to the Master in non blocking calls. All this will happen inside a TryBlock for error synchronization.

Algorithm 1: A Master application

```
communication initialization;
create a communicator for each worker;
for i from 1 to number_of_workers do
    MPI_TryBlock_start(comm[i],global,reqs[i]);
    create jobs[i];
    non-blocking send jobs[i];
    non-blocking receive results;
    MPI_TryBlock_ifinish(reqs[i]);
end
while more_work_to_do and still_have_workers do
    MPI_Waitany_local(req,idx);
    if error_occured_in tryblock[idx] then
        do recovery;
        recover jobs[idx];
        MPI_TryBlock_start(comm[idx],global,reqs[idx]);
        non-blocking send jobs[idx];
        non-blocking receive results;
        MPI_TryBlock_ifinish(reqs[idx]);
    else
        free reqs[idx];
        MPI_TryBlock_start(comm[idx],global,reqs[idx]);
        create new jobs[idx];
        non-blocking send jobs[idx];
        non-blocking receive results;
        MPI_TryBlock_ifinish(reqs[idx]);
    end
end
```

Algorithm 2: A Worker application

```
initialization;
create a communicator with master;
while more_work_to_do do
    MPI_TryBlock_start(comm,global,req);
    non-blocking receive job;
    if not more_work_to_do then goto finish compute results;
    non-blocking send results;
    finish: MPI_TryBlock_finish(req);
    if failed then
        do recovery;
        goto start;
    end
end
```

In data parallel models of computation, usually a hierarchy of communicators exists. An example is a 2D grid of communicators for matrix computation, row and column communicators plus MPI_COMM_WORLD, which is the global communicator. Processes in each row or column communicator can apply fault-awareness individually in transactional blocks without enquiring any failure knowledge about other communicators. An outer TryBlock can synchronize failure of all ranks in MPI_COMM_WORLD for a proper checkpoint of the process state. Algorithm 3 shows how a data parallel program can utilize TryBlock on one communicator.

Algorithm 3: A data-parallel application

```
communication initialization;
if restarted then
    load data from last checkpoint (optional);
end
repeat
    while more_work_to_do do
        MPI_TryBlock_start(comm,global,req);
        computation, communication and/or I/O;
        wait for operations to finish;
        inject local errors;
        MPI_TryBlock_ifinish();
        MPI_Wait_local(req, status, timeout);
        if failure happened then
            isolate and mitigate the failure;
            if recovery_needed then break;
        end
        periodically checkpoint;
    end
    if recovery_needed then
        do recovery procedure;
    end
until more_work_to_do or restart_needed;
```

Applications can utilize the TryBlock's communicator to form a fault-tolerance solution only in the group of ranks in the communicator. This allows fine grained fault-management by allowing two or more group of ranks to execute individually and apply fault-tolerance distinctively. Although synchronization between communicators is needed periodically for periodic checkpoints. Figure 1 depicts how multiple communicators can progress in case of failure. Individual communicators can deal with faults disregarding other communicators and multiple communicators can synchronize through a bigger communicator. In order to achieve higher scalability we recommend using global communicators like MPI_COMM_WORLD as few as possible.

V. Experimental Results

In this section we describe how to use FA-MPI in a simple application and how it can be configured to achieve the maximum resiliency and performance at the same time.

A. Game of Life

Conway's "Game of Life" [18] is an inherently data parallel program and is a good educational example used in the HPC community to demonstrate various research results. This program uses a two dimentional matrix of cells. Each cell has two states, either dead or alive. At each evolution of the program, state of a cell is computed base on states of its eight neighbor cells. This property makes decomposing of this evolution like algorithm into a set of localized problems. For simplicity we chose a one dimentional row decomposition of

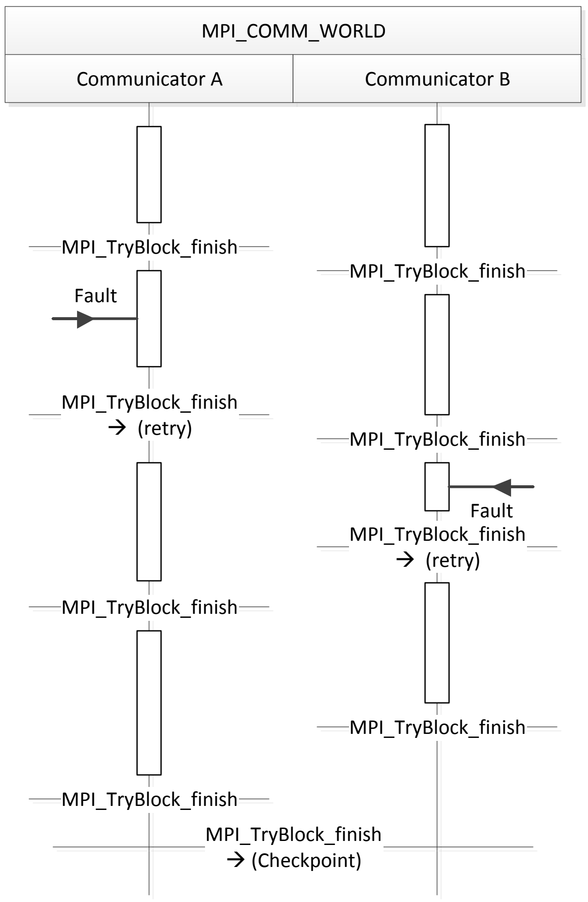

Fig. 1. Multiple communicators progress in case of failure

Algorithm 4: Fault-Tolerant Game of Life with TryBlock support

```
communication initialization;
read local matrix of game of life;
$TryBlock\_step \leftarrow n$;
repeat
    while not finished do
        if evolution mod TryBlock_step equals 0 then
            MPI_TryBlock_start();
        end
        compute current generation;
        communicate border cells with neighbors;
        exponentially inject local errors;
        if evolution mod TryBlock_step == 0 then
            MPI_TryBlock_finish();
            if no failure then
                read the last checkpoint into local buffers;
            else
                checkpoint;
            end
        end
    end
until more_work_to_do or restart_needed;
```

TABLE I. EXPERIMENT SYSTEM SPECIFICATION

CPU	Intel Xeon E5-2620 @2.00GHz
CPU per Node	2
Total Nodes	4
Total Cores	48
Total Threads	96
Memory	132 GB
Network (InfiniBand)	Mellanox MT27500 ConnectX-3
Storage	Raid5

the program so at each evolution a rank will send and recieve buffers to and from its two neighbors (except first and last ranks which have only one neighbor). In these experiments we used the Game of Life program as an example of how transactional concepts of FA-MPI can be applied to a data-parallel application. A matrix of size 4800 by 4800 cells and a maximum of 5000 evolutions is used for the experiments. We decided to have more proof of concept experiments rather than heavy optimizations. Algorithm 4 simply demostrates how we applied TryBlocks and failure injection into Game of Life program. “TryBlock_step” is defined as how often TryBlock operations are executed during multiple evolutions of the program. A TryBlock_step of n shows that TryBlock is executed every nth evolution of the program.

B. System Specification

We used a cluster of 4 nodes with specifications in Table I for experiments. We tried to utilize the total resources available in the system. We implemented FA-MPI as a separate plugin module for OpenMPI 1.7. Tests use the openib BTL (byte transfer layer) and SM (shared memory) layers of OpenMPI. For simplicity, we used a basic system for checkpoint restart by writing and reading directly into local files on each node’s local storage device. Each node is configured with a software raid5.

C. TryBlock Evaluation

We implmented MPI_TryBlock_ifinish() using a fault-tolerant Allgatherv/Allreducev algorithm. We have a linear implementation for MPI_TryBlock_ifinish(), but a log and tree based algorithm will have better performance and will scale better[1]. Figure 2 shows the fault-free overhead of TryBlock by increase in the number of ranks in communicator’s group. Results in Figure 2 indicate linear increase in the execution time of TryBlock, but it is only due to the linear implementation of the TryBlock.

D. Failure Model

For each individual rank we assume an exponential failure rate with failure density function:

$$f(t) = \lambda e^{-\lambda t},$$

[1]For these experiments, our tree based algorithm which has logarithmic performance based on [14] was not ready. We have high confidence that results with log based algorithm and a state of the art checkpoint system will be ready by the time of camera ready submission of this paper.

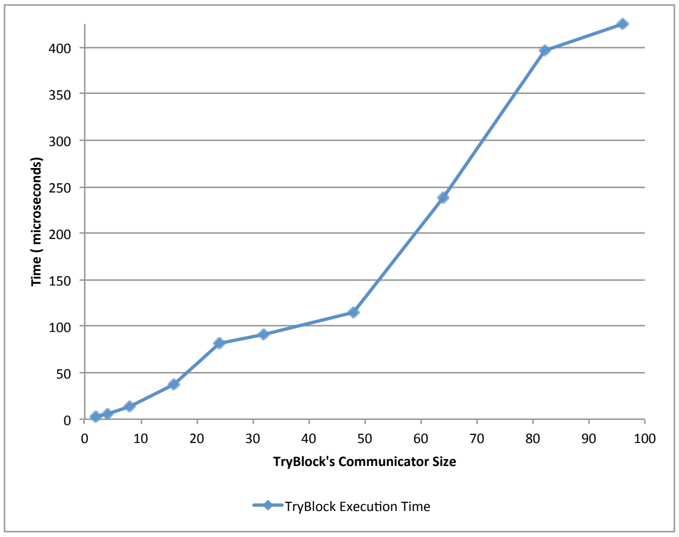

Fig. 2. Performance of TryBlock with linear implementation in fault-free case.

TABLE II. EFFECT OF FAILURE RATE ON PERFORMANCE OF TRYBLOCK. THE EXECUTION TIME IS ASSOCIATED WITH A LOOP OF 20000 CALL TO TRYBLOCK FUNCTIONS.

Node MTTF	Total Failures	Execution Time (secs)
100	0	13.92
2	74	9.94
1	154	11.29
0.1	883	12.56
0.01	5654	10.31
0.002	15914	10.57

and exponential failure distribution:

$$F(t) = 1 - e^{-\lambda t}.$$

Where $F(t)$ is the probability that a rank fails at a time in the next t seconds. mean time to failure is define as:

$$MTTF = 1/\lambda$$

Exponential failure rate is memoryless. In other words time to the next failure is independent of the time of past failures. This model is used to describe the failure rate of many systems. In the remainder of the paper any reference to MTTF means MTTF of one rank or process.

We tried experimenting with different MTTFs for a loop of 20000 times calling TryBlock's start and finish functions. Table II shows the effects of failure rate on the performance of TryBlock. Instead of examining only process failures, we raised errors in the program with rate $F(t)$ at each rank. A failure is raised by MPI_Request_raise_error() on the TryBlock's request with MPI_ERR_TRYBLOCK_INVALIDATE errorcode. As results indicate, failure rate does not affect the performance of TryBlock. This can be caused by the fact that TryBlock implementation is not optimized yet.

In order to show the effects of executing TryBlock followed by a checkpoint, we decreased the TryBlock step and we

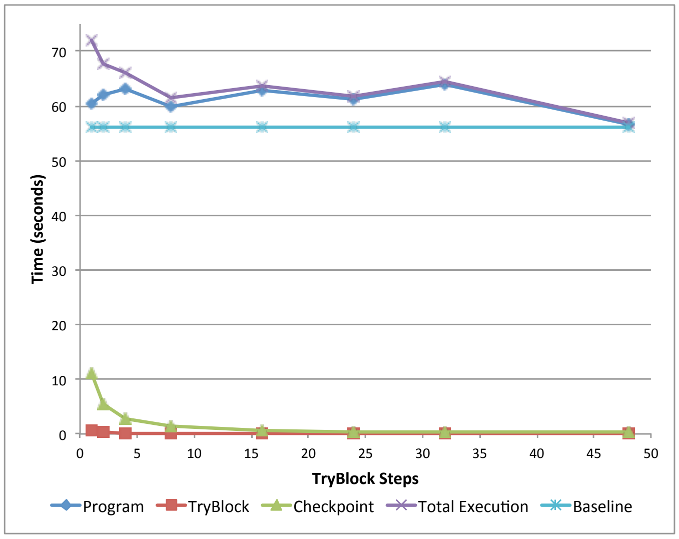

Fig. 3. Effects of the decrease in the TryBlock steps while keeping number of game's evolution constant. 48 Ranks is used for this experiment.

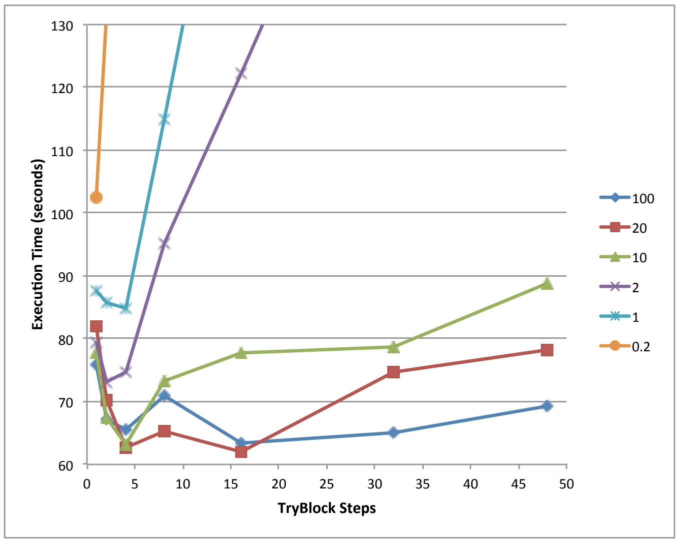

Fig. 4. Effect of different TryBlock steps for systems with different failure rates. MTTF is based on seconds

assumed no failures occur during execution. This resulted in increase in total application execution time as expected. Overal time spend on checkpointing also increased because more checkpoints are repeated more often. Figure 3 shows the results with breakdown of each segment of application.

For the following experiments with failures we did not include failures happening during initialization/finalization and checkpoint/restart phase, Although each one of these sections of program can be protected by TryBlocks and be recovered upon any failure. As we defined earlier even checkpoint/restart recovery is a normal procedure that can be covered by TryBlocks and repeated untill successful or unable to continue anymore.

Figure 4 depicts how systems with different failure rates react to different granularity of TryBlocks. For example a system with MTTF of 10 seconds have the optimal performance when

TryBlock is performed every four steps of application loop. Systems with low MTTF will suffer from higher TryBlock steps. In other words systems with high failure rates need to be controlled with higher TryBlock constraints. The reason that execution time increases with higher TryBlock steps is a result of the need of more repetition of failed evolutions after a failure happens and the application is restarted from checkpoint.

VI. Related Work

User-level failure mitigation (ULFM) [10] is a proposal for a fault-tolerant MPI designed by MPI Forum's fault-tolerance working group (FTWG) and is currently under development. The goal of ULFM is to add fault-tolerant support for MPI to allow implementation of a wide range of fault-tolerant techniques on top of ULFM. It adds a minimal set of API extensions to MPI and is more suitable for libraries providing fault-tolerance rather than user applications. In ULFM's semantics, when an MPI operation fails because of process failure, the user can acknowledge the failure and get the group of locally known failed ranks. Revoking a communicator provides global error propagation and prevents deadlock in blocking point-to-point and collective functions.

Since ULFM encourages libraries to implement complementary fault-tolerant semantics on top of it and it appears logical that we design FA-MPI using ULFM. However, this approach proved difficult because of fundamental difference in how two approaches handle failures. ULFM explicitly supports only process failures and other transient failures are masked to process failure. Although this approach resembles more portability, it restricts application resiliency and scalability by limiting ULFM's capability to make a wider range of recovery policies for different failure models. However, FA-MPI allows more failure types like fail-stop, crash, omission, timing and incorrect computation [19] to be detected, isolated, mitigated and recovered at both the user and MPI level. For example, applications can take advantage of low latency in unreliable network protocols like UDP and handle checksum recovery at application level and decrease jitter in the system. Omission and timing failures can be handled coarse grained and explicitly by using timeout in the application. Byzantine failures is not support by either proposal. Timeout is another aspect of difficulty implementing FA-MPI using ULFM because of lack of timeout semantics in MPI functions.

Agreement function in ULFM provides a means for global per communicator agreement on a value and transparent global failure propagation. MPI_TryBlock_finish() needs a special version of fault-tolerant allgatherv and it can be implemented using a non-fault-tolerant allgatherv followed by ULFM's agreement in a retry loop to cover failures happen during the commit. But this approach introduces a high overhead of $(allgatherv_t + allreduce_t) * retries$ which is not efficient. As we insist on lightweight transactions, a fault-tolerant allgatherv using consensus protocol can solve the problem with much lower overhead.

As mentioned earlier, MPI was not designed with fault-tolerance in mind. So trying to add fault-tolerance functionality for entire API of MPI seems overwhelming. FA-MPI proposes adding fault-tolerance support for only non-blocking communication operations as future system have to use non-blocking semantics to increase scalability and performance by overlapping computation, communication, and I/O.

In ULFM's design, failure propagation is infused with failure recovery by MPI_Comm_revoke(). Revoke is a failure propagation mechanism and if a process detects a failure, it can revoke the communicator. Revoking a communicator causes associated communicators of all ranks in the communicator's group, skipping failed ranks, become invalid permanently for future operations. While this approach might seem reasonable, it restricts the ability of the user to make separate decisions for failure detection and failure recovery. Also, it disables operations like point-to-point which could be finished successfully. Revoke makes decision globally without having global knowledge of the system, However, FA-MPI decides locally and globally based on global knowledge of failures. Except for blocking operations, FA-MPI can be reduced to ULFM by having a TryBlock with local error propagation surrounding each operation. the shrink operation in ULFM creates a new communicator containing only healthy ranks. This can help FA-MPI for recovery purposes but a communicator needs to be revoked before shrinking.

VII. Conclusion and Future Work

FA-MPI is a set of extension APIs for MPI standard to allow achieving fault-awareness using a transactional model. FA-MPI detects, disseminates, and notifies failures and helps the user with isolation, mitigation, and recovery procedures. We expect applications using FA-MPI to run to completion with higher probability than the non-fault-aware versions. In the design of FA-MPI we emphasize the use of lightweight transactions combined with non-blocking operations to achieve a high-level of scalability and resilience in future and current large-scale systems. Non-blocking communication is the most important aspect of message passing middleware that we support. We implemented the system and showed performance results with different rates of failures for different problem sizes. This paper showed how granularity of TryBlocks can be configured to address both resiliency and performance of the program. The proposed API here is designed as preliminary work and is subject to further modifications and enhancements in the future. Further enhancement of the MPI_Status structure is needed to cover extensive support for different failure models. More discussion is needed to target FA-MPI's support of one-sided semantics because MPI lacks the semantics of non-blocking target synchronization at the present. Also not all collective operations in MPI-3 have non-blocking forms.

Acknowledgments

This work was supported in part by the National Science Foundation under grant CCF-1239962 and Sandia National Laboratories under grant PO-1330625. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

[1] Message Passing Interface Forum, “MPI: a message passing interface standard version 3.0,” Tech. Rep., Sep. 2012.

[2] G. Stellner, “CoCheck: checkpointing and process migration for MPI,” in *Parallel Processing Symposium, 1996., Proceedings of IPPS ’96, The 10th International*, 1996, pp. 526–531, cited by 0443.

[3] A. Agbaria and R. Friedman, “Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations,” in *HPDC*, 1999.

[4] S. Sankaran, J. M. Squyres, B. Barrett, and A. Lumsdaine, “The LAM/MPI checkpoint/restart framework: System-initiated checkpointing,” in *in Proceedings, LACSI Symposium, Sante Fe*, 2003, p. 479493.

[5] R. Batchu, Y. S. Dandass, A. Skjellum, and M. Beddhu, “MPI/FT: a model-based approach to low-overhead fault tolerant message-passing middleware,” *Cluster Computing*, vol. 7, no. 4, p. 303315, Oct. 2004, cited by 0034. [Online]. Available: http://dx.doi.org/10.1023/B:CLUS.0000039491.64560.8a

[6] E. Elnozahy and W. Zwaenepoel, “Manetho: transparent roll back-recovery with low overhead, limited rollback, and fast output commit,” *IEEE Transactions on Computers*, vol. 41, no. 5, pp. 526–531, 1992, cited by 0353.

[7] S. Rao, L. Alvisi, H. M. Viny, and D. C. Sciences, “Egida: An extensible toolkit for low-overhead fault-tolerance,” in *In Symposium on Fault-Tolerant Computing*. Press, 1999, p. 4855, cited by 0097.

[8] K.-H. Huang and J. A. Abraham, “Algorithm-based fault tolerance for matrix operations,” *IEEE Trans. Comput.*, vol. 33, no. 6, p. 518528, Jun. 1984. [Online]. Available: http://dx.doi.org/10.1109/TC.1984.1676475

[9] J. Hursey, R. L. Graham, G. Bronevetsky, D. Buntinas, H. Pritchard, and D. G. Solt, “Run-through stabilization: An MPI proposal for process fault tolerance,” in *Recent Advances in the Message Passing Interface*. Springer, 2011, pp. 329–332.

[10] W. Bland, “User level failure mitigation in MPI,” in *Euro-Par 2012: Parallel Processing Workshops*, 2013, pp. 499–504.

[11] A. Skjellum and P. V. Bangalore, “FA-MPI: fault-aware MPI specification and concept of operations,” University of Alabama at Birmingham, Tech. Rep. UABCIS-TR-2012-011912, May 2012.

[12] L. G. Valiant, “A bridging model for parallel computation,” *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.

[13] P. A. Bernstein and N. Goodman, “Concurrency control in distributed database systems,” *ACM Comput. Surv.*, vol. 13, no. 2, p. 185221, Jun. 1981, cited by 1091. [Online]. Available: http://doi.acm.org/10.1145/356842.356846

[14] J. Hursey, T. Naughton, G. Vallee, and R. L. Graham, “A log-scaling fault tolerant agreement algorithm for a fault tolerant MPI,” in *EuroMPI*, 2011, pp. 255–263.

[15] R. van Renesse, Y. Minsky, and M. Hayden, “A gossip-style failure detection service,” in *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, ser. Middleware ’98. London, UK, UK: Springer-Verlag, 1998, pp. 55–70. [Online]. Available: http://dl.acm.org/citation.cfm?id=1659232.1659238

[16] W. Jia and W. Zhao, “Fault-tolerant scaleable multicast algorithm with piggybacking approach on logical process ring,” *Computers and Digital Techniques, IEE Proceedings* -, vol. 145, no. 4, pp. 292–300, 1998.

[17] M. K. Aguilera, W. Chen, and S. Toueg, “Heartbeat: A timeout-free failure detector for quiescent reliable communication,” in *Distributed Algorithms*. Springer, 1997, p. 126140. [Online]. Available: http://link.springer.com/chapter/10.1007/BFb0030680

[18] M. Gardner, *MATHEMATICAL GAMES The fantastic combinations of John Conway’s new solitaire game ”life”*. New York: Scientific American, Oct. 1970.

[19] M. Barborak, A. Dahbura, and M. Malek, “The consensus problem in fault-tolerant computing,” *ACM Comput. Surv.*, vol. 25, no. 2, p. 171220, Jun. 1993. [Online]. Available: http://doi.acm.org/10.1145/152610.152612