The Impact of a Fault Tolerant MPI on Scalable Systems Services and Applications

Richard Graham*, Joshua Hursey*, Geoffroy Vallée*, Thomas Naughton* and Swen Boehm*
* Oak Ridge National Laboratory, Oak Ridge, TN USA 37831
Email: {rlgraham,hurseyjj,valleegr,naughtont,bohms}@ornl.gov

***Abstract*—Exascale targeted scientific applications must be prepared for a highly concurrent computing environment where failure will be a regular event during execution. Natural and algorithm-based fault tolerance (ABFT) techniques can often manage failures more efficiently than traditional checkpoint/restart techniques alone. Central to many petascale applications is an MPI standard that lacks support for ABFT. The Run-Through Stabilization (RTS) proposal, under consideration for MPI 3, allows an application to continue execution when processes fail. The requirements of scalable, fault tolerant MPI implementations and applications will stress the capabilities of many system services. System services must evolve to efficiently support such applications and libraries in the presence of system component failures. This paper discusses how the RTS proposal impacts system services, highlighting specific requirements. Early experimentation results from Cray systems at ORNL using prototype MPI and runtime implementations are presented. Additionally, this paper outlines fault tolerance techniques targeted at leadership class applications.**

***Keywords*-MPI; Fault Tolerance; Runtime Environment; Algorithm Based Fault Tolerance; Run-through Stabilization**

I. Introduction

Scientific applications targeting exascale-class High Performance Computing (HPC) machines must be prepared for a highly concurrent computing environment in which system component failure will be a normal event during application execution [1]. As such, application developers are investigating natural and Algorithm-Based Fault Tolerance (ABFT) techniques that will allow them to manage such process and memory failures during application execution. These fault tolerance techniques often allow the application to recover from failure more efficiently than what traditional checkpoint/restart techniques alone can provide.

The Message Passing Interface (MPI) standard supports many of the leadership class HPC applications on existing petascale systems. However, the MPI standard currently lacks semantics and interfaces for sustained application execution in the presence of process failures. The MPI Forum standardization body is working to support ABFT applications and libraries through the efforts of the Fault Tolerance Working Group. The Run-Through Stabilization (RTS) proposal, currently under consideration for MPI 3.0, represents a first step towards a comprehensive fault tolerant MPI standard supporting process fault tolerant applications and libraries. This proposal allows an application to continue execution even when MPI processes fail during execution. The complementary Process Recovery proposal will follow the RTS proposal allowing the application to replace failed processes in their execution environment.

The requirements of a fault tolerant MPI implementation and application will stress the capabilities of existing system services. System services will need to evolve and develop novel interfaces to work efficiently with scalable, fault tolerant applications. The resource manager, scheduler and network must all be prepared to support such applications in the presence of various degrees of system component failure.

This paper is organized as follows with related work discussed in the relevant sections. Section II outlines some of the fault tolerance techniques leadership class applications are experimenting with to prepare for future systems with high failure rates. Section III describes the current RTS proposal under consideration by the MPI Forum which is designed to support those applications. Section IV describes the requirements on the runtime environment to support fault tolerance at the MPI and application levels. Section V discusses cross-cutting requirements that span the entire system software stack highlighting specific requirements, and collaboration opportunities. Before concluding in Section VII, some early experimentation results from Cray systems at Oak Ridge National Laboratory (ORNL) using prototype MPI and runtime implementations are presented in Section VI.

II. Algorithm-Based Fault Tolerance Techniques

This section describes some of the ABFT techniques with which leadership class applications are experimenting. Many of the techniques described in this section are well established in literature, but their use in applications has been limited by the lack of support from underlying system services like MPI. The User-Level Failure Mitigation (ULFM) RTS proposal (described in Section III) is designed to allow applications to experiment with these and other techniques in a portable manner.

A. Faulty Subgroups

Ensemble-style applications divide their work among autonomous groupings of worker processes, where each worker group is given a single task to complete. The ensemble groups are managed by a designated group of manager processes. If each of the worker and manager groups contain only one process each, this style resembles a traditional manager-worker model. More often each of the worker and manager groups

contain a large number of processes working in parallel on the given task.

Since each of the worker groups is autonomous to the other worker groups, the failure of one worker group does not affect the completion of a separate worker group. The manager group must receive notification of the failure of a worker group to determine if the task given to that failed group can be either given to another worker group or discarded. In a 2004 paper by Gropp and Lusk, the authors outlined how a general manager-worker application might achieve a degree of fault tolerance given a high-quality MPI implementation that provides process fault tolerance error handling beyond what is specified by the MPI standard [2]. The ULFM RTS proposal formalizes the expected semantics so such applications can be built portably upon any MPI standard compliant implementation.

When a single process fails in a worker group, the worker group may manage the failure within their group. If the worker group is able to recover from the loss of the peer process then it can do so without the need for external intervention of the manager group. A basic technique for handling process failure in the worker group is to cause all processes in that worker group to fail. To do so an application needs only to set the error handler on the worker group communicator to MPI_ERRORS_ARE_FATAL. Even if the worker group attempts to recover from the loss there are situations in which it might decide to terminate the group due to a set of failures that exceed their failure model. In which case any process in the worker group can call MPI_ABORT on the worker group communicator causing just those processes in that group to terminate.

To understand how MPI communicator error handlers can be used to support fault tolerance in ensemble applications, consider the following simple base case. In this example assume that the failure of a manager is critical to the entire application, and individual worker groups cannot tolerate process failure so failure of a worker process is critical to the worker group. Therefore when a worker group fails it is important that the failure does not cause the remainder of the application to abort due to the cascading nature of the MPI_ERRORS_ARE_FATAL error handler. The application must replace the default, fatal error handler on MPI_COMM_WORLD with, at least, MPI_ERRORS_RETURN. This will prevent the failure of a worker group from triggering the failure of the rest of the application. Since this example assumes the failure of a manager to be critical to the entire application, the manager communicator must set a custom error handler that calls MPI_ABORT on MPI_COMM_WORLD when a process fails in the manager group. Each of the individual worker groups create a group-local communicator and use the default, fatal error handler MPI_ERRORS_ARE_FATAL. From this base case example, an application can start experimenting with more resilient manager groups, and specialized worker groups that can tolerate various sequences of process failure. Such application experimentation is ongoing with a number of applications at ORNL.

The MPI library provides the necessary functionality to contain fatal behavior to just the effected communicators, so the runtime environment does not need to know about the groupings of processes. The runtime environment must only provide the ability to remote terminate a process when directed to do so by the MPI library.

B. Recovery Blocks

Applications that are iterative in nature may find the concept of a recovery block to be a useful fault tolerance programming construct to apply to their application. The recovery block model breaks up the code into a section of code that performs some calculation followed by an acceptance test [3]. If the test fails the application knows that the preceding section of code executed incorrectly, in the context of this discussion, possibly due to process failure – though other types of errors can be detected and handled this way. An application may choose to terminate the execution, rollback modified data and retry the previous section of code, or retry the previous calculation using a different technique. The notion of recovery blocks is similar to transactions in database systems, but in the context of the ULFM RTS proposal the user is responsible to rolling back modified data when the acceptance test fails, if necessary.

Applications will need to use a fault tolerant agreement operation when constructing acceptance tests that are able to provide the same pass/fail information to all processes even in the presence of process failure. The MPI_COMM_AGREE operation defined by the ULFM RTS proposal provides the application with a fault tolerant agreement collective operation that returns a logical *AND* over the input flag value. Figure 1 presents a sketch of a recovery block with associated acceptance test. In the simplified code sketch the only error handled is of process failure, and since collectives are guaranteed to complete and never hang in the presence of process failure it is acceptable to jump directly to the acceptance test rendezvous point at the first sign of error.

Application developers experimenting with this fault tolerance technique must be careful to manage the overhead of the synchronization in this model. Some applications may find that they are able to use the nonblocking MPI_COMM_IAGREE to overlap the execution of the acceptance test with the setup for the next recovery block.

C. Linear Algebra Libraries

ABFT dense linear algebra operations are possibly the most well studied of all ABFT techniques [4], [5], [6], [7]. The Fault Tolerant Linear Algebra (FT-LA) project is encapsulating many of these techniques into a library with a similar set of functionality as ScaLAPACK [8].

Most of the linear algebra ABFT techniques use a data encoding technique similar to diskless checkpointing to recover data lost to process failure [9]. A checksum of the data in a slice of the matrix is stored in a spare process. When a process failure occurs, the checksum data is used to recover the lost data and the failed process is, often, replaced.

```
1  int rc, allsucceeded;
2
3  // Recovery Block
4    rc = MPI_Allreduce( ..., comm );
5    if( MPI_ERR_PROC_FAILED == rc ) {
6      goto acceptance_test;
7    }
8    rc = MPI_Allreduce( ..., comm );
9    if( MPI_ERR_PROC_FAILED == rc ) {
10     goto acceptance_test;
11   }
12
13 // Acceptance Test
14 acceptance_test:
15   // Check result of computation
16   // The return code in this example.
17   allsucceeded = (MPI_SUCCESS == rc);
18   // Agree upon acceptance test
19   MPI_Comm_agree(comm, &allsucceeded);
20   // If failed, then the allsucceeded will be 'false'
21   if( !allsucceeded ) {
22     // Start recovery action
23   }
```

Fig. 1. Example of a Recovery Block.

Since these algorithms can be complex to implement correctly and efficiently, encapsulating them in a library allows an application to take full advantage of the fault tolerant algorithms without incurring the overhead of providing the functionality themselves. Further encapsulation of other ABFT techniques will make it easier for applications to make the transition from fault-unaware to fault-aware and, even, fault-tolerant algorithms. Therefore libraries like FT-LA are a step in the right direction for application developers concerned about failures.

III. Run-Through Stabilization Proposal for MPI

The MPI Forum's Fault Tolerance Working Group is charged with defining a set of semantics and interfaces to enable fault tolerant applications and libraries to be portably constructed on top of the MPI interface. The Run-Through Stabilization (RTS) proposal allows an application to continue running and using MPI even when one or more processes in the MPI universe fail without replacing those failed processes. For the most part, process recovery can be achieved using the dynamic process management interface in MPI. However since this is a cumbersome task with the existing interfaces, the working group is pursuing a process recovery proposal to complement the RTS proposal. For the purposes of this paper, we will focus our discussion on the RTS proposal and, in particular, the simplified version of the proposal entitled: *User-Level Failure Mitigation (ULFM)*.

The RTS proposal has evolved over time as the working group incorporates feedback from the user and MPI developer communities. The ULFM version of the RTS proposal simplifies previous versions [10] to provide only the essential functionality necessary to build stronger fault tolerance capabilities as third-party libraries on top of MPI (such as the original variations). The interfaces and semantics represented in the ULFM RTS proposal were strongly influenced by previous work in the area of fault tolerant interfaces for MPI. The FT-MPI [11] and MPI/FT [12] projects approached fault tolerance in slightly different ways. The ULFM RTS proposal was designed to support most, if not all, of the functionality provided by these, and other projects through third-party libraries building upon the exposed interfaces and semantics.

The ULFM RTS proposal assumes *fail-stop* process failure meaning that an MPI process permanently stops communicating with other MPI processes, and its internal state is lost [13]. Other types of faults not currently addressed by the MPI standard (i.e, reliable message delivery), like Byzantine failures [14], are left to the application to address, as necessary.

The application is notified of a process failure once it attempts to communicate directly (e.g., point-to-point operations) or indirectly (e.g., collective operations) with the failed process through the return code of the function, and error handler set on the associated communicator. The ULFM RTS proposal does not change the default error handler of MPI_ERRORS_ARE_FATAL, so to use these semantics the application must explicitly change the error handler to, at least, MPI_ERRORS_RETURN on all communicators involved with fault handling in the application.

For point-to-point operations, direct communication between two active processes is unaffected by the failure of other, non-participating processes. For example, if process A fails, process B can still send messages to process C, and vice versa. It is not until process B tries to communicate with process A that an error is raised.

For MPI_ANY_SOURCE receive operations the semantics are slightly different than those for directed point-to-point operations. For MPI_ANY_SOURCE receives the MPI implementation cannot determine if a new process failure is important to the correct completion of the receive operation. As such, upon MPI internal notification of a process failure, blocking, unmatched MPI_ANY_SOURCE receive operations will complete in error (MPI_ERR_PROC_FAILED). Asymmetrically, unmatched nonblocking MPI_ANY_SOURCE receive operations will *not* complete in error, but rather raise a warning error of MPI_ERR_PENDING. The asymmetry is necessary so as not to violate the message order matching guarantees provided by MPI. When the user receives the warning error of MPI_ERR_PENDING they can either cancel the request, or acknowledge the failure and continue waiting on the receive. The MPI_COMM_FAILURE_ACK and MPI_COMM_GET_FAILURE_ACKED[1] operations allow the application to acknowledge a set of failures, and

[1]Alternative function names are being consider for many of the new MPI functions. As such, some of the functions may change their names before acceptance into the MPI standard. No changes are expected to their semantics.

access the current group of acknowledged failures. After acknowledgement, the pending request can be passed to a test or wait operation without raising an error until a subsequent, unacknowledged process failure occurs.

Collective operations must be *fault-aware*, meaning that they will not hang in the presence of failures [15]. To preserve failure-free performance, collective operations are not required to provide uniform return codes. Such a synchronization requirement would severely impact the performance of many collective operations (e.g., MPI_BCAST). It is important to notice that since communicator creation calls are also collective they are also not required to provide uniform return codes, which might result in a partially created communication object.

Since applications often need to reason about the uniform completion of an operation, like communicator creation, the ULFM RTS proposal provides a fault tolerant agreement algorithm in the form of the MPI_COMM_AGREE operation [16]. This special collective operation will tolerate existing and emerging process failure to reach agreement on a specified boolean value, and return uniformly at all processes. The agreement operation is useful in determining the success or failure of a single or set of operations in a recovery block (See Section II-B).

Depending on when a failure occurs, when it is notified to each process, and the communication pattern of the application it is possible for the application to struggle to determine safe rendezvous recovery points, the absence of which can often lead to application deadlock. The ULFM RTS proposal provides a communicator *invalidation* operation that revokes the context of the communicator to assist the application in regaining control in such circumstances. The MPI_COMM_REVOKE operation is a local operation that propagates a signal to all other alive processes in the communicator indicating that the context has been revoked. All outstanding and future operations on that communicator will immediately return MPI_ERR_REVOKED, with the exception of MPI_COMM_SHRINK and MPI_COMM_AGREE.

Once a communicator has been revoked, a new communicator containing only the alive processes from the previous communicator can be created using the MPI_COMM_SHRINK operation. The shrink operation creates a new communicator, and, unlike other communicator creation operations, it uniformly returns at all alive calling processes with a valid communication object even in the presence of process failure.

Though this section has focused on communicators, semantics and interfaces are defined by the ULFM RTS proposal for the entire MPI specification including one-sided windows and file handles.

IV. Runtime Requirements

The ULFM RTS proposal enables applications to continue running and using the HPC system after losing one or more processes in the MPI universe. HPC system services largely have not had to support such fault tolerant applications and middleware libraries. As such, these system services will need to evolve and develop novel, flexible interfaces to work efficiently with scalable, fault tolerant applications. This section outlines many of the needs that stem from the ULFM RTS proposal, and also highlights additional features that would further enhance such a fault tolerant HPC environment.

In all of the cases below, proper documentation of exposed interfaces and expected behavior when process failure is encountered is paramount to building a supporting infrastructure for fault tolerant applications. All of the system services from low-level network drivers to MPI libraries must work in concert to provide the quality of service that a fault tolerant application expects.

An HPC runtime system should support different end-user scenarios to facilitate alternate uses. This includes the separation of mechanisms and policies for the target platform. The runtime should provide interfaces that enable the users to define policies appropriate for their use cases. This enables the runtime to be reused for different purposes. The focus of this paper is on the MPI case, but the runtime should be usable for other common HPC tasks like debuggers, etc.

The MPI library implementation relies on a runtime system that provides several fundamental capabilities to enable application level fault tolerance. A reliable "out-of-band" communication subsystem should be provided. This aids in the bootstrapping of higher-level HPC communication services. There should also be some support for monitoring of the parallel execution context. The detection and notification of failures at the runtime level provides an essential service for the MPI library. The runtime infrastructure should also support robust process management and clean-up when abnormal conditions do arise. Additionally, the runtime infrastructure itself should be resilient in order to guarantee application execution despite the occurance of failures at large scale.

There must also be interfaces to facilitate recovery (e.g., restart process). These recovery mechanisms will be used by the MPI implementation to restore the system level aspects of the failed tasks. Fault tolerance at the runtime level must be clearly separated from fault tolerance at the MPI level. This division clarifies the semantics for runtime interfaces, i.e., "contracts". For example, the transition between runtime managed resilience and application managed fault tolerance that is assumed during the transition between runtime initialization (`rte_init()`) and MPI_Init(). After the MPI function returns, the application and MPI library govern the policies for how failures should be managed.

Part of the challenge to provide a runtime that can be used for fault tolerance at the MPI level is to provide an internal infrastructure that is scalable and a set of capabilities that can be used for the implementation of fault tolerance mechanisms. For instance, a tree-based architecture is a typical choice for the design of a scalable runtime system, where MPI tasks are the leaves of the tree, and runtime infrastructure the inner nodes of the tree: it allows reduction at the level of tree nodes for scalable communications and enable a fair level of parallelism based on a arity of the tree. For instance, on Cray platforms, Application Level Placement Scheduler

(ALPS) is used for the deployment of processes across the compute nodes. It forms a tree-based control network among the ALPS daemons on the compute nodes [17]. This tree is rebalanced based on the number of nodes used when launching remote tasks. However, standard tree-based architectures have limitations with respect to fault tolerance. To address this issue, a solution is to make the distinction between the infrastructure used during application initialization (typically up to the successful execution of MPI_INIT) and the infrastructure used by the application until its finalization (typically from MPI_INIT to MPI_FINALIZE). During the first "bootstrap" phase, a basic tree-based infrastructure is deployed and is then used to startup more advanced architecture such as a binomial graph (BMG) based topology [18], which guarantee communications despite the failure of communication channels by setting up redundant links between the different processes used to instantiate the runtime infrastructure. With such a runtime architecture, a route can always be found for communications between MPI tasks (via redundant communication channels). Therefore, the application can still be efficiently deployed at scale and it is guaranteed that applications can successfully run despite the failures of communication channels or processes internally used by the runtime infrastructure. Failures at the application level can then be handled by the ULFM RTS proposal at the MPI level, as described in Section III.

V. Cross-cutting Requirements

There are some system services that have an effect on all the layers of the HPC software stack. These cross-cutting services influence the applications, MPI layer, and runtime in different ways. In this section we highlight capabilities for three of these services that are important to support fault tolerance in an HPC context.

A. Network

The main purpose of an MPI library is to provide an abstraction from the high performance interconnects in HPC machines, and to provide tuned versions of common communication patterns to applications. These abstractions have enabled applications to remain portable over the generations of HPC systems that they must target.

To provide effective fault tolerance capabilities to the application, the MPI library relies on a responsive network driver interface that is both highly efficient, and has well defined semantics and interfaces for managing failures. The primary focus of the ULFM RTS proposal is on process failure, but such process failure is often caused by the failure of resources supporting the execution of the process. Machine failure will cause the loss of one or more processes, as will the failure of a switch. From the perspective of an MPI application these are all process failure situations since MPI does not expose a concept of a machine or switch to the application.

When a connection to a remote process fails the network driver should communicate to the MPI implementation as much information regarding this failure as possible. If a single process has failed, then this might be handled differently than if a switch failed and the remote processes might still be alive. In the latter case the MPI library might take further action to determine if the network has become segmented and if the remote processes are still reachable via another route. So the more informative and reliable the error information from the network driver the more flexibility the MPI library has when determining the most effective and appropriate fault detection algorithm for that system. The timeliness of this information is also important to providing fast notification of process failure to the application.

The ULFM RTS proposal introduces an revocation operation that revokes the communication context of a communicator. This operation will immediately complete in error (MPI_ERR_REVOKED) all outstanding operations on the communicator. The network must have the capability to support such a request from the MPI implementation. If the network is managing all of the request matching in hardware, then it should expose the ability to remove all of the requests that match a given communicator. Since matching of MPI messages includes the associated communicator, this might be as simple as removing all registered requests that match an appropriate bitmask.

Fault tolerant applications that use MPI_ANY_SOURCE receives (e.g., manager/worker style applications) will need the ability to properly cancel these requests. Further, other fault tolerant applications that may not need to call the revoke operation may, instead, require the ability to cancel pending send operations. Though MPI_CANCEL is an existing MPI operation, it is known to be difficult to support. The MPI library takes the responsibility for managing the subtle semantics of MPI_CANCEL, but the underlying network driver should expose the ability to remove a single, given request from the matching queue when asked to do so by the MPI library.

Network provided collective operations must also be aware that process failure may occur during the collective operation. The ULFM RTS proposal requires that these collective operations never hang in such a situation. As such, may hardware collective implementations may need a feedback mechanism to flush the collective when an error has been encountered.

B. Resource Manager

An HPC runtime provides the interface between the resource manager and the MPI layer. Most HPC resource managers assume that once a process failure occurs that the application job should be terminated so as not to waste resources on an application that is not capable of handling such an error and will likely hang. This assumption has been guided by the MPI standard which defaults to a fatal approach to process failures which would necessitate such a global rule for job termination. The ULFM RTS proposal does not replace the default fatal behavior. The proposal defines what should happen when the MPI library encounters a process failure *and* the application has asked for the opportunity to handle such errors. The application communicates to the MPI library that it is willing to handle process failures by replacing the default MPI error

handler. Additionally, the runtime can use these resource manager interfaces to improve how it manages resilience for the parallel job.

The resource manager should expose an interface to the runtime that allows the MPI library to communicate whether or not it is able to handle process failure. To remain backwards compatible with prior MPI implementations and existing, fault-unaware applications such an interface should default to the existing fatal behavior. An MPI implementation that is fully compliant with the ULFM RTS proposal would use this interface to indicate to the resource manager that it should hand control for job termination in abnormal circumstances to the MPI library. It is possible that the MPI library encounters a sequence of process failures that it is not able to recover from. In such a situation, the resource manager should expose an interface for the MPI library to abort the job, and let the runtime and resource manager clean up.

Today, resource managers must be fault tolerant to continue providing service when one or more machines in the HPC system are down for maintenance. When those machines fail, the resource manager recovers its own communication mechanism, and reports the failure to the scheduler so no new jobs will be scheduled on that resource. This resilient communication network, and notification service could greatly benefit a fault tolerant MPI implementation. The MPI implementation must detect process failure (node failure is seen to the application as one or more processes failing together), and notify all alive processes of the failure. If the MPI implementation and the resource manager are both doing failure detection and notification in parallel this increases the noise on the system, load on the network, and overhead seen by the application. A high-quality resource manager should expose this resilient communication network and failure notification service to the MPI library and runtime to reduce these overheads.

It has been shown that some applications need fewer resources at the start and end of their execution cycle [19], and other applications need a certain proportion of processes for execution (e.g., power of two number of processes). A fault tolerant application that recovers from a set of process failures may decide to remove other alive processes to regain the proper proportion of process and/or to increase locality among the remaining processes. In such a situation, potentially large numbers of functioning machines are idle while the application continue execution. The resource manager should expose an interface to allow the application and MPI library to return resources to the scheduler when they are no longer needed for the application. Optionally, a resource manager might also expose a set of interfaces to allow the MPI library to request more resources to replace failed resources during execution.

The resource manager has the authoritative view of the resources. Therefore, it is best equipped to provide information to the runtime for bootstrapping the parallel job. The resource manager should provide an interface to this global information at a "node" local level, which will enhance the startup of tasks on the system. This can also help to distribute the initialization of services and enhance the overall scalability for the runtime and ultimately the application.

C. Scheduler

As mentioned in the previous section, it is useful for the MPI library to have the ability to return resources to the scheduler that are no longer needed, and, potentially, request replacement resources. The MPI library will communicate these needs through the resource manager interfaces, but the scheduler has the authority on how this request is handled so must be aware of the potential for such requests.

The scheduler must realize that fault tolerant applications (and even some not fault tolerant applications [19], [20]) can benefit from a more dynamic scheduling environment. Such applications are more aware of the changing system resources and able to adapt accordingly. The scheduler is able to increase throughput when it allows an application to return functioning, but no longer needed resources back to the available pool of resources.

The application might request replacement resources be added to their allocation to assist in the recovery of their application execution. This request is communicated from the application through the MPI library and subsequently the resource manager before reaching the scheduler. The scheduler should immediately respond with whether or not it can service such a request, and, if it can, then the estimated time until the resources are available. Once the resources are available the scheduler should communicate their availability to the MPI library so that it know where it should launch replacement processes, in cooperation with the application.

To more efficiently handle such requests for more resources, the scheduler might consider retaining a small pool of spare resources shared between all jobs to act as replacements for failed resources. This spare pool can service short, small (possibly preemptible) jobs ensuring that they are readily available when a fault tolerant application requires them for recovery. So when an application requests more resources the turn around time for that request can be diminished.

Based on the expected time to resource availability an application can decide how to proceed with recovery. If the expected time is short, the application might decide to wait for the resources to become available before recovering. If the expected time is long, the application might decide to continue execution with the reduced number of processes and then rebalance once the resources become available. The key concept here is that the application is being provided sufficient information to make informed decisions, and interfaces to communicate their needs to the various system services.

VI. Early Experimentation Results

The testing for our preliminary experiments were done on two Cray XK6 systems at ORNL. The machines, *Chester* and *Jaguar*, are the development and production platforms, respectively, for the Oak Ridge Leadership Computing Facility (OLCF). Jaguar has completed the first phase of an upgrade that will result in the *Titan* supercomputer. The current version of Jaguar (April 2012) is comprised of 200 cabinets containing

18,688 nodes with AMD Opterons processors, each with 16 cores, for a total of 299,008 cores. In addition, 960 of Jaguar's 18,688 compute nodes contain a NVIDIA graphical processing unit (GPU). The nodes have 32GB of memory per node, with 2GB per core. The Chester development machine is a single cabinet Cray XK6 of 80 nodes (16 cores per node, each with a NVIDIA GPU). Both Chester and Jaguar use the Gemini interconnect.

A. ULFM RTS Prototype Performance

The NetPIPE benchmark was used to assess the 1-byte latency and bandwidth impact of the modifications necessary for the ULFM RTS prototype compared with a build of Open MPI without fault tolerance support. For these tests we used the Chester machine at ORNL. The shared memory 1-byte latency incurred a 1% overhead between the unmodified versus modified Open MPI implementation. The bandwidth overhead was negligible for shared memory on this machine. In testing of the Gemini interconnect, no noticeable difference in performance was seen between the unmodified versus modified Open MPI implementation.

Prior work has demonstrated with the previous, expanded RTS proposal that collective overhead is minimal over communicators that include masked failed processes [15]. The ULFM RTS proposal only supports collectives over dense communicators where all processes are active. As such we have been able to use unmodified fault-unaware collective operations that exit at the first sign of process failure. An out-of-band notification service will force an error at processes that are indirectly dependent upon a failed process, as described in [15]. Since the fault-unaware collective operations were unchanged, so was the performance characteristics of these collective operations for the ULFM RTS proposal.

Prior work has shown that the MPI_COMM_AGREE operation can be implemented such that it has similar complexity as a MPI_ALLREDUCE operation [16].

B. Runtime Performance

This section presents preliminary performance evaluation of the runtime system developed at ORNL. In this document, we focus on the time to set up a BMG topology in order to have redundant communication channels, as well as runtime scalability.

Our runtime prototype supports the deployment of a BMG topology between the processes of the runtime infrastructure. This topology has been proven to be a good candidate for fault tolerance thanks the redundant links it sets up. We set up the BMG topology on top of the tree-topology used for the bootstrapping runtime infrastructure. The current implementation is based on a centralized implementation that identify the set of missing communication channels when mapping the BMG topology on top of the tree topology. Table I shows the time required to perform this mapping: the nodes are the nodes of the topology, i.e., all MPI ranks as well as all processes of the runtime infrastructure.

Nodes	Time (in seconds)
1024	0.19
2048	0.25
4096	0.47
8192	1.21
16384	3.10
32768	12.68
65536	65.36
131072	297.95

TABLE I
TIME TO MAP A BMG TOPOLOGY

The runtime supports deployment via different resource managers, which on Cray is based on ALPS [17]. ALPS is used to launch applications on Cray compute nodes from within a job allocation. As such, the runtime startup leverages information available in ALPS for the initialization of a tree topology used during job startup. The runtime uses an on-node resource manager interface that provides information about the control network tree that ALPS maintains internally when launching the tasks on the compute nodes. This node-local interface reduces the overhead to configure the base agents used in the runtime system. Additionally, the startup of these base agents can be done using a single ALPS launch command (i.e., a single `aprun` invocation). Once this base agent is available the runtime distributes the additional information (deployment and connection topology information) to start the target executable for the job.

VII. CONCLUSION

Scientific applications are investigating natural and ABFT techniques that will allow them to more efficiently manage process failure in exascale systems. The MPI Forum's Fault Tolerance Working Group is working to support ABFT applications and libraries through proposals like the ULFM RTS proposal. This proposal, currently under consideration for MPI 3.0, allows an application to continue execution even when MPI processes fail during execution. The MPI libraries and runtime environments support fault tolerant applications will stress the capabilities of most existing system services. This paper presents a discussion of how these various system services will need to evolve and develop novel interfaces to work efficiently with scalable, fault tolerant applications.

To withstand the projected failure rates of exascale systems the entire software stack must be resilient and working in concert. This level of integration and cooperation is necessary to provide the application with every opportunity to efficiently manage failures as they emerge during normal execution so they can best harness exascale class machines.

ACKNOWLEDGMENTS

Research sponsored by the Mathematical, Information, and Computational Sciences Division, Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.

REFERENCES

[1] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, "Toward exascale resilience," *International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 374–388, 2009.

[2] W. Gropp and E. Lusk, "Fault tolerance in message passing interface programs," *International Journal of High Performance Computing Applications*, vol. 18, no. 3, pp. 363–372, 2004.

[3] B. Randell, "System structure for software fault tolerance," in *Proceedings of the international conference on reliable software*. New York, NY, USA: ACM Press, 1975, pp. 437–449.

[4] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers*, vol. 33, no. 6, pp. 518–528, 1984.

[5] Z. Chen and J. Dongarra, "Algorithm-based fault tolerance for fail-stop failures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 12, pp. 1628–1641, 2008.

[6] Y. Du, P. Wang, H. Fu, J. Jia, H. Zhou, and X. Yang, "Building single fault survivable parallel algorithms for matrix operations using redundant parallel computation," *International Conference on Computer and Information Technology*, pp. 285–290, 2007.

[7] J. Langou, Z. Chen, G. Bosilca, and J. Dongarra, "Recovery patterns for iterative methods in a parallel unstable environment," *SIAM Journal of Scientific Computing*, vol. 30, no. 1, pp. 102–116, 2007.

[8] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou, "Algorithm-based fault tolerance applied to high performance computing," *Journal of Parallel and Distributed Computing*, vol. 69, no. 4, pp. 410 – 416, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731508002141

[9] J. S. Plank, K. Li, and M. A. Puening, "Diskless checkpointing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 10, pp. 972–986, October 1998.

[10] 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 *EuroMPI 2011: Proceedings of the 18th EuroMPI Conference*, Santorini, Greece, September 2011.

[11] G. E. Fagg, E. Gabriel, Z. Chen, T. Angskun, G. Bosilca, J. Pjesivac-Grbovic, and J. J. Dongarra, "Process fault-tolerance: Semantics, design and applications for high performance computing," *International Journal for High Performance Applications and Supercomputing*, vol. 19, no. 4, pp. 465–478, 2005.

[12] 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, pp. 303–315, Jan 2004.

[13] R. D. Schlichting and F. B. Schneider, "Fail-stop processors: An approach to designing fault-tolerant computing systems," *ACM Transactions on Computing Systems*, vol. 1, pp. 222–238, August 1983.

[14] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, 1982.

[15] J. Hursey and R. Graham, "Analyzing fault aware collective performance in a process fault tolerant MPI," *Parallel Computing*, vol. 38, no. 1-2, pp. 15–25, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819111001414

[16] 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: Proceedings of the 18th EuroMPI Conference*, Santorini, Greece, September 2011.

[17] *Cray XT^{TM} System Management*, S-2393-22 ed., Cray, Jul. 2009. [Online]. Available: http://docs.cray.com/books/S-2393-22

[18] T. Angskun, G. Bosilca, and J. Dongarra, "Binomial graph: A scalable and fault-tolerant logical network topology," in *International Symposium on Parallel and Distributed Processing and Applications*. Springer, 2007, pp. 471–482.

[19] T. Armstrong, Z. Zhang, D. Katz, M. Wilde, and I. Foster, "Scheduling many-task workloads on supercomputers: Dealing with trailing tasks," in *Many-Task Computing on Grids and Supercomputers (MTAGS), 2010 IEEE Workshop on*, November 2010, pp. 1 –10.

[20] R. Sudarsan, C. Ribbens, and D. Farkas, "Dynamic resizing of parallel scientific simulations: A case study using LAMMPS," in *Computational Science – ICCS 2009*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2009, vol. 5544, pp. 175–184. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-01970-8_18