Resilient X10 over MPI User Level Failure Mitigation

Sara S. Hamouda
Australian National University, Canberra, Australia
sara.salem@anu.edu.au

Benjamin Herta Josh Milthorpe
David Grove Olivier Tardieu
IBM T.J. Watson Research Center, Yorktown Heights, NY, USA
{bherta,jjmiltho,groved,tardieu}@us.ibm.com

Abstract

Many PGAS languages and libraries rely on high performance transport layers such as GASNet and MPI to achieve low communication latency, portability and scalability. As systems increase in scale, failures are expected to become normal events rather than exceptions. Unfortunately, GASNet and standard MPI do not provide fault tolerance capabilities. This limitation hinders PGAS languages and other high-level programming models from supporting resilience at scale. For this reason, Resilient X10 has previously been supported over sockets only, not over MPI.

This paper describes the use of a fault tolerant MPI implementation, called ULFM (User Level Failure Mitigation), as a transport layer for Resilient X10. By providing fault tolerant collective and agreement algorithms, on demand failure propagation, and support for InfiniBand, ULFM provides the required infrastructure to create a high performance transport layer for Resilient X10.

We show that replacing X10's emulated collectives with ULFM's blocking collectives results in significant performance improvements. For three iterative SPMD-style applications running on 1000 X10 places, the improvement ranged between 30% and 51%. The per-step overhead for resilience was less than 9%.

A proposal for adding ULFM to the coming MPI-4 standard is currently under assessment by the MPI Forum. Our results show that adding user-level fault tolerance support in MPI makes it a suitable base for resilience in high-level programming models.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: [Distributed programming]; D.4.5 [*Reliability*]: [Fault-tolerance]

Keywords X10, Resilience, MPI, User Level Failure Mitigation

1. Introduction

The Asynchronous Partitioned Global Address Space (APGAS) programming model [20] has been demonstrated to enable both scalable high performance [16, 21] and high productivity [19] on a variety of High Performance Computing (HPC) systems and applications. Although originally developed in the context of the X10 language [10], APGAS programming model concepts also underlie a variety of other HPC programming systems including Chapel [3], Habanero [9, 14], Co-Array Fortran 2.0 [22], and UPC++ [23].

X10'16, June 13-17, 2016, Santa Barbara, CA, USA

DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2931028.2931030

Most of these languages delegate process communication to high performance networking layers such as GASNet and MPI. To adapt to a wide variety of systems the X10 runtime uses a network abstraction layer called X10RT. Implementations of X10RT over TCP/IP sockets, MPI, or IBM's PAMI transport are available.

As HPC systems are fairly reliable, it was reasonable for APGAS programming systems to mostly ignore issues related to fault-tolerance and application recovery. However, since projections indicate high failure rates on future systems [8], fault tolerance is growing in importance. Recently, X10 has developed a resilience extension that enables applications to recover from process failures [11, 12]. The initial development of Resilient X10 was primarily concerned with bringing X10 to cloud computing environments. Therefore it only made resilient the portions of the X10 runtime stack that were appropriate for a cloud computing environment. In particular, the X10RT sockets transport was used for communication and experimental evaluations of Resilient X10 scaled to only a few hundred X10 places.

To address the growing need for a standard fault tolerance semantics in MPI, a fault tolerance working group was formed by the MPI forum to assess MPI fault tolerance proposals. MPI-ULFM (User Level Failure Mitigation) is currently the only active proposal for adding fault tolerance to the MPI-4 standard. Since 2012, an OpenMPI implementation of ULFM has been available, and used in a variety of resilient systems [4, 5, 15, 18].

This paper describes the implementation of a resilient MPI transport for X10 using MPI-ULFM. Both Resilient X10 and ULFM adopt a flexible user level fault tolerance approach, resulting in a straightforward integration of the two systems. By providing fault tolerant collective and agreement algorithms, on demand failure propagation, and support for InfiniBand, ULFM provides the required infrastructure to create a high performance transport layer for Resilient X10. For X10 applications that use collective operations, our results show that replacing X10 emulated collectives with native MPI blocking collectives results in significant performance improvement in resilient mode, ranging between 30% and 51%.

We begin by reviewing Resilient X10 in Section 2 and MPI-ULFM in Section 3. The major contributions of the paper are the integration of X10 and MPI-ULFM described in Section 4 and the empirical evaluation of the resulting system in Section 5. Section 6 discusses related work and Section 7 concludes and describes future work.

2. Resilient X10

The X10 programming language [10] has been developed as a simple, clean, but powerful and practical programming system for scale-out computation. Its underlying APGAS programming model [20] is organized around the two notions of *places* and *asynchrony*.

Asynchrony is provided through a single block-structured control construct, `async S`. If `S` is a statement, then `async S` executes `S` in a separate *task* (logical thread of control). Dually, `finish S` executes `S`, and waits for all tasks spawned (recursively) during the execution of `S` to terminate, before continuing. Exceptions escaping from `S` or tasks spawned by `S` are combined in a `MultipleExceptions` instance that is thrown by `finish` upon termination.

A place is an abstraction of shared, mutable data and worker threads operating on the data, typically realized as an operating system process. A single APGAS computation may consist of hundreds or potentially tens of thousands of places. The `at (p) S` permits the current task to change its place of execution to `p`, execute `S` at `p` and return, leaving behind tasks that may have been spawned during the execution of `S`. The termination of these tasks is detected by the `finish` within which the `at` statement is executing.

2.1 Failure Model

Resilient X10 [11, 12] builds on X10 by exploiting the strong separation provided by places to provide a coherent semantics for execution in the presence of failures. It assumes a fail-stop failure model where the unit of failure is the place. A place `p` may fail at any time, with the instantaneous loss of its heap and tasks. A failed place cannot affect the execution of the non-failed places. Any `at (p) S` that was launched from place `q` throws a `DeadPlaceException` (`DPE`). Any future attempt to launch an `at (p) S` from place `q` will also throw a `DPE`. However, any `at (q) S` that was initiated at place `p` before the failure executes to completion. Global refs pointing to objects hosted at `p` now "dangle", but they cannot be dereferenced since an `at (p) S` will throw a `DPE`. The failure of place zero is considered catastrophic, and causes the application to terminate.

2.2 Resilient Finish

When a place `p` fails it may be in the middle of running `at (q) S` statements at other (non-failed) places `q`. The key design decision in Resilient X10 is defining how to track the termination of these "orphan" statements. While `S` has lost its parent place, it still belongs to an enclosing `finish`. Resilient X10 provides a resilient implementation of the `finish` construct that allows orphan tasks to be adopted by the nearest enclosing `finish`.

To maintain the state of `finish` constructs and their enclosing tasks beyond the failure of places, an in-memory resilient store is used. The creation and termination of remote tasks require interactions with this store resulting in a performance overhead. Currently, place zero is used as a `finish` resilient store in Native X10, since it is assumed to never fail. Managed X10 uses a distributed java based resilient store for `finish`. The rationale for the design and implementation of Resilient X10 is discussed in [11, 12].

2.3 Resilient X10RT

X10RT is responsible for inter-place communication. The Resilient X10 runtime assumes that a resilient X10RT: 1) detects place failures, and provides several APIs to inform the runtime about them, propagating place failure information to other places is not required; 2) returns an error when trying to communicate synchronously with known failed places; and 3) preserves the ability for two non-failed places to communicate irrespective of other places failures.

3. MPI User Level Failure Mitigation

ULFM [1] extends the MPI-3 standard by adding fault tolerance semantics to existing operations assuming a fail-stop failure model. It also adds a small set of new interfaces for failure propagation, process agreement and communicator recovery, that can be used to implement a variety of application specific fault tolerance strategies. Table 1 describes the main new interfaces of ULFM.

Table 1: Main ULFM new interfaces

Operation	Description
MPI_Comm_revoke	invalidates the communicator
MPI_Comm_failure_get_acked	retrieves the local list of known failed processes
MPI_Comm_failure_ack	acknowledges the discovery of current failed processes
MPI_Comm_shrink	creates a new communicator by excluding dead processes from a given communicator
MPI_Comm_agree	a resilient consensus algorithm

3.1 Failure notification

ULFM applications need to change the default MPI error handler from `MPI_ERRORS_ARE_FATAL` to `MPI_ERRORS_RETURN`, or to a user specified error handler. After that, ULFM is able to report process failure on a per operation basis using special error codes. A process is notified of a failure once it attempts to communicate with a dead process.

3.2 Failure propagation

By default, ULFM does not perform global notification of process failure, which agrees with Resilient X10 requirements. However, it provides the new operation `MPI_Comm_revoke` that can be used for global propagation if required by an application. After one process revokes a communicator, all other processes sharing the same communicator will receive errors when they call any non-local MPI operation, except `MPI_Comm_shrink` and `MPI_Comm_agree`. To resume the application afterward, a new communicator must be constructed using one of the following recovery mechanisms.

3.3 Communicator Recovery

Communicator recovery strategies are generally classified as shrinking or non-shrinking [15]. In shrinking recovery, a new communicator is created by excluding the dead processes in a given communicator. ULFM provides the new interface `MPI_Comm_shrink` for this purpose. Non-shrinking recovery requires replacing dead processes with new ones, so that the application can restore its state using the same number of processes. The ULFM specification is based on the MPI-3 standard, which supports dynamic process creation. Thus ULFM applications can also apply non-shrinking recovery, although the procedure is complicated as shown in [4].

3.4 Non Blocking Communication Operations

X10 uses non-blocking two sided communication operations for transferring data and active messages between places. Initiating a non-blocking send or receive is done using `MPI_Isend` or `MPI_Irecv` respectively. Checking the completion of a non-blocking operation is done periodically by calling `MPI_Test`. ULFM does not report process failure through initiation calls of non-blocking operations. Instead, failure reporting is postponed until the corresponding completion operation is invoked (i.e. `MPI_Test` or `MPI_Wait`). The same semantics hold for non-blocking collectives.

3.5 Global Failure Detection using `MPI_ANY_SOURCE`

The operation `MPI_Iprobe (source, ...)` is used for checking for incoming messages from a certain source. When the failure of the source is detected, `MPI_Iprobe` returns an error code. A call to `MPI_Iprobe` with the special source parameter `MPI_ANY_SOURCE` returns an error when it detects a failure of *any process in the*

communicator that is not acknowledged by the application. This feature is used by X10 for global failure detection without the need for revoking the communicator.

3.6 Collective and Agreement Operations

ULFM provides non-uniform failure reporting for the collective operations [6]. Depending on the collective implementation, when a process dies, some processes may report the collective as successful, while others may report it as failed. The same behavior occurs in X10's emulated collectives (§ 4.3).

When such disagreement scenarios happen, a *consensus algorithm* can be used to reach a consistent view between processes about a certain state. ULFM adds the interface `MPI_Comm_agree` to provide a failure aware consensus algorithm.

4. X10 ULFM Integration

From the previous sections, we can list a number of agreements between Resilient X10 and ULFM: both assume fail-stop process failure, support user level fault tolerance, and do not require global failure notification by default. That is why we found MPI-ULFM a suitable transport layer for Resilient X10. This section describes the implementation details of the X10 ULFM transport[1].

4.1 OpenMPI ULFM Limitations

Until recently, there has been only one reference implementation for ULFM based on OpenMPI 1.7 [2]. Another implementation over MPICH is currently under development [7]. We used the OpenMPI reference implementation, and found that it has the following limitations: 1) it does not target full node failure; 2) it does not support thread safety; 3) it does not provide non-blocking collectives; and 4) it does not provide one sided RDMA operations. For the first limitation, we performed our experiments by killing MPI processes rather than killing a whole node. For the second limitation, we configured MPI to use the `MPI_THREAD_SERIALIZED` mode, and manually serialized the access to MPI between X10 threads using a shared mutex. The third limitation did not impact X10, because X10's collectives API `Team` can use blocking MPI collectives (see §4.3). The fourth limitation also did not impact X10, because X10 can perform remote memory operations using two sided communication operations.

4.2 Detecting Dead Places

To detect dead places, we registered the error handler in Figure 1 to the default communicator `MPI_COMM_WORLD`. The error handler uses local ULFM operations to acknowledge and query the list of dead places (Lines 4-8). This list is stored in the transport's global state (Lines 10-11) to be used for answering the runtime queries about the status of places. The error handler does not revoke `MPI_COMM_WORLD`. However, each place is guaranteed to eventually detect the failure of any other place due to periodically calling `MPI_Iprobe(MPI_ANY_SOURCE, ...)` to check for incoming messages from other places (see § 3.5).

4.3 Team Collectives

X10 contains a collective API similar to MPI, located in `x10.util.Team`, offering collective operations such as barrier, broadcast, all-to-all, etc. Team attempts to make use of any collective capabilities available in the underlying transport. For transports that provide native collectives, Team maps its operations to the transport collective implementations. For transports that do not provide collectives, such as TCP/IP sockets, Team provides emulated collective implementations.

[1] Our implementation is included in X10 version 2.6, available for download at `https://github.com/x10-lang/x10`.

```
1 void mpiCustomErrorHandler(MPI_Comm* comm,
      int *errorCode, ...) {
2   MPI_Group f_group; int f_size;
3   //Acknowledge & query failed processes
4   MPI_Comm_failure_ack(*comm);
5   MPI_Comm_failure_get_acked(*comm,
        &f_group);
6   MPI_Group_size(f_group, &f_size);
7   int* f_ranks = malloc(...);
8   MPI_Group_translate_ranks(f_group,... ,
        f_ranks);
9   //Update global state
10  global_state.deadPlaces = f_ranks;
11  global_state.deadPlacesSize = f_size;
12 }
```

Figure 1: X10-ULFM custom error handler

An interesting combination arises when the underlying transport supports some, but not all, of the functionality needed by X10. The X10 thread model requires non-blocking operations from the network transport, because there may be runnable tasks in the thread's work queue, and a blocking network call will prevent that runnable work from completing, leading to possible deadlock. MPI-3 offers non-blocking collectives, but other than barrier these are optional, and MPI-2 only supports blocking collectives.For best performance we still want to make use of these, so our implementation calls an emulated barrier immediately before issuing the blocking collective. This allows us to line up all places so that when they reach the blocking operation, they are all in a position to pass through the collective immediately.

Team operations are exposed to the applications as blocking collectives. When a non-blocking MPI collective is used, Team internally uses a `finish` construct to block the calling thread until the completion of the transport's non-blocking collective.

Our experiments show that, in resilient mode, the emulated barrier incurs less performance overhead than `finish`. That is why it is preferable to map Team operations to blocking MPI collectives in resilient mode. In the ULFM transport of X10, all team operations are mapped to blocking MPI collectives.

Team Creation. The collective operation `MPI_Comm_create` is used to create a sub communicator for each team object from the parent communicator `MPI_COMM_WORLD`. Since X10 does not revoke or recover the default communicator after process failure, collective operations over `MPI_COMM_WORLD` will fail if any of the processes is dead. That is why in resilient mode, we use `MPI_Comm_shrink` to avoid accessing dead places while creating a new team communicator, as follows:

```
1 MPI_Comm team_comm, shrunken;
2 MPI_Comm_shrink(MPI_COMM_WORLD,&shrunken);
3 MPI_Comm_create(shrunken,group,&team_comm);
```

Measurements of team creation performance are presented in § 5.4.

4.4 Resilient Iterative Framework

X10 provides a high level checkpoint/restart framework for developing resilient iterative algorithms. The framework provides a resilient executor (Figure 2), and an interface of five methods to be implemented by the target application: `step`, `isFinished`, `checkpoint`, `remake`, and `restore`.

Previously, checkpoint coordination was centralized and relied heavily on `finish`: a coordinator place started a new `finish` block that initiated an activity at each place to execute `checkpoint` (a fan-out `finish`). When these activities terminated successfully, a new fan-out `finish` was created to execute the next application steps. Coordinating restoration was performed in the same way: when a place failed, a coordinator place first invoked the `remake` method to

```
1  do {
2    try {
3      if (restoreRequired())
4        app.remake(...);
5
6      finish for(p in places) at (p) async {
7        while (!app.isFinished()) {
8          //restore
9          if (restoreRequired()) {
10           val status = app.restore(...);
11           setRestoreRequired(false);
12           if (!team.agree(status))
13             throw new Exception("Err:...");
14         }
15         //checkpoint
16         if (is_checkpoint_iteration()) {
17           val status = app.checkpoint(...);
18           if (!team.agree(status))
19             throw new Exception("Err:...");
20           else
21             commit_checkpoint();
22         }
23         //execute a step
24         app.step();
25       }//while !isFinished
26     }//finish
27   } catch (ex:Exception) {
28     setRestoreRequired(true);
29   }
30 } while(restoreRequired());
```

Figure 2: SPMD-style resilient iterative executor

reconstruct the application's distributed objects over a new set of places, then created a fan-out `finish` to invoke `restore`.

The availability of a distributed consensus algorithm in ULFM allows us to avoid the centralized coordination and the repeated fan-outs with every checkpoint and restore. The success of checkpointing or restoration at all places is validated using `Team.agree` which maps to ULFM's `MPI_Comm_agree`.

5. Performance Evaluation

To evaluate the performance of resilient X10 applications over ULFM, we measured: 1) the overhead of ULFM's fault tolerance extensions, 2) Team performance improvement by replacing emulated collectives with MPI collectives, 3) the overhead of place zero resilient `finish` at different scales, and 4) the performance of failure detection, agreement and recovery at different scales.

Applications. Three applications were used in the experiments: LULESH shock hydrodynamics proxy application with domain size 30^3 [16], in addition to two machine learning benchmark programs, PageRank and Linear Regression, from X10's Global Matrix Library [13]. PageRank was initialized using a graph of 1 M vertices per place stored in a sparse matrix with density of 0.001. Linear Regression used a randomly initialized dense matrix of 50 K records per place, each record contained 100 classification features. The three applications were modified to use the iterative framework presented in §4.4. All the applications ran for 50 iterations.

Experiment Setup. The experiments were conducted on the *Raijin* supercomputer at NCI, the Australian National Computing Infrastructure. Each compute node has a dual 8-core Intel Xeon (Sandy Bridge 2.6 GHz) processors, and uses an Infiniband FDR network. We allocated 10 GiB of memory per node, and statically bound each place to a separate core. ULFM was built from source revision ea08943, which includes bug fixes to ULFM 1.1. X10 was built from source revision bcd8f5b, using GCC 4.4.7 for post compilation. The X10 Global Matrix Library used OpenBLAS 0.2.15 configured to use one thread per process (`OPENBLAS_NUM_THREADS=1`). X10 places were configured to use one worker thread per place (`X10_NTHREADS=1`). Resilient X10 mode was enabled in some experiments by setting `X10_RESILIENT_MODE=1`. We also used the environment variable `X10_RESILIENT_FINISH_SMALL_ASYNC_SIZE=1024`, to provide lower finish overhead for messages smaller than 1024 bytes.

All timing results are the average of 30 runs. The 99% confidence intervals for the reported average step times are less than 3.5% of the computed averages.

5.1 ULFM Fault Tolerance Overhead

In order to measure the performance overhead of ULFM's fault tolerance extensions, we installed two configurations of UFLM, one with fault tolerance features, and one without. ULFM with FT was built with the configuration parameters `--enable-mpi-ext=ftmpi --with-ft=mpi`, and executed using the runtime parameters[2] `-am ft-enable-mpi --mca errmgr_rts_hnp_proc_fail_xcast_delay 0`. ULFM without FT was built without these parameters, and thus it is expected to behave like a standard MPI implementation with similar performance to OpenMPI 1.7. The fault tolerant configuration adds extra conditions to validate the communicator status before performing MPI operations. It also adds code segments related to ULFM's new interfaces, which are not invoked in this experiment. Table 2 compares the average step time for the three applications using the two configurations, both running in non-resilient X10 mode. The results show that the used reference implementation adds no overhead to the application performance in a failure free scenario.

Table 2: ULFM's fault tolerance overhead over 1000 places

	Avg. step time		
X10 transport	**LULESH**	**PageRank**	**LinReg**
ULFM without FT	127 ms	83 ms	39 ms
ULFM with FT	127 ms	83 ms	39 ms
Overhead	0%	0%	0%

5.2 X10 Emulated Collectives versus MPI Collectives

Resilient X10 over sockets can use only *emulated* Team collectives. In this section, we measure the performance improvement when emulated collectives are replaced with MPI blocking or non-blocking collectives. Because ULFM does not provide non-blocking collectives, Table 3 compares only two Team implementations for ULFM (Emulated and MPI blocking). To measure the performance of non-blocking collectives, we had to use another OpenMPI implementation that supports non-blocking collectives. We used OpenMPI 1.10.2 configured to use the same thread level as ULFM `MPI_THREAD_SERIALIZED`. Remember from section § 4.3 that Team adds an emulated barrier before calling a blocking collective, and a `finish` construct before calling a non-blocking collective.

We conclude the following from Table 3:

- In non-resilient mode, non-blocking collectives provide better performance than blocking collectives. However, non-blocking collectives result in higher overhead in resilient mode. The internal use of `finish` by Team to wait for the completion of non-blocking collectives is the source of the overhead.
- In non-resilient mode, blocking and emulated collectives have a comparable performance. However, in resilient mode, blocking

[2] The parameter –mca errmgr_rts_hnp_proc_fail_xcast_delay 0 was recommended by ULFM team to reduce the failure detection time.

Table 3: Resilient finish overhead using different Team implementations and different transports over 1000 places

X10 transport	Collective Impl.	LULESH avg. step time non-resilient	LULESH avg. step time resilient	LULESH avg. step time overhead	PageRank avg. step time non-resilient	PageRank avg. step time resilient	PageRank avg. step time overhead	LinReg avg. step time non-resilient	LinReg avg. step time resilient	LinReg avg. step time overhead
OpenMPI 1.10.2	Emulated	131 ms	200 ms	52.7%	84 ms	148 ms	76.2%	39 ms	78 ms	100%
	MPI Blocking	133 ms	142 ms	6.8%	92 ms	92 ms	0%	39 ms	39 ms	0%
	MPI Non-blocking	72 ms	205 ms	184.7%	77 ms	128 ms	66.2%	18 ms	48 ms	166.7%
ULFM with FT	Emulated	127 ms	198 ms	55.9%	76 ms	147 ms	93.4%	39 ms	80 ms	105.1%
	MPI Blocking	127 ms	138 ms	8.7%	83 ms	83 ms	0%	39 ms	39 ms	0%
	Improvement	0%	30.3%	–	-9.2%	43.5%	–	0%	51.3%	–

collectives result in 30% to 51% performance improvement compared to emulated collectives.

- Finally, the emulated barrier called before a blocking MPI collective does not add a significant overhead for resilience.

In PageRank and LinReg, inter-place communication occurs only through Team collective operations. When blocking collectives are used, there is no significant resilience overhead. LULESH is a stencil simulation; in addition to communicating through collectives, each place periodically communicates with its 26 neighboring places to exchange ghost cells. The result is more interactions with place zero resilient store, and higher resilience overhead.

5.3 Resilient Finish Overhead

Figure 3 shows weak scaling results for our three applications using resilient and non-resilient `finish`. Team used ULFM blocking collectives. PageRank and LinReg were executed using multiples of 250 places. LULESH was executed using 343, 512, 729 and 1000 places, because it requires a perfect cube number of places. As the applications scale to larger number of places, the overhead remains almost constant.

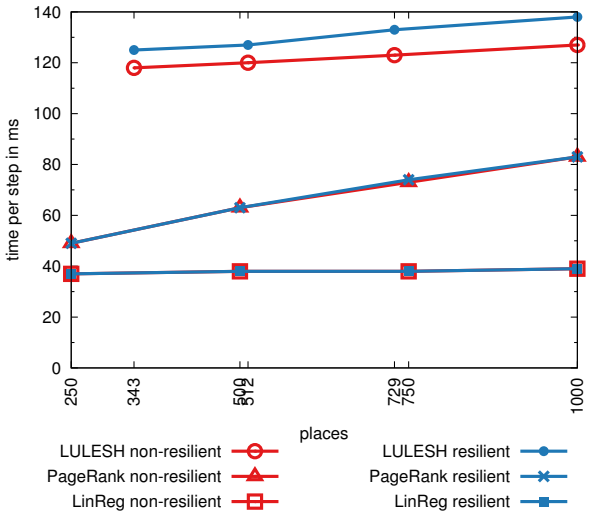

Figure 3: Place zero resilient finish overhead

5.4 Recovery Performance

Figures 4 to 6 show the performance of our applications while recovering from one place failure. A spare place was used to replace the failed one. Application data were checkpointed every 10 steps in a *double in-memory* resilient store. The store keeps a local copy of the data at the owner place, and another copy at a backup place. Place(P/2) was forced to fail by issuing a `SIGKILL` signal at the beginning of the 16th step. When a place fails, all places restore their latest checkpoint from the data copy stored locally; only the spare place copies remote data from the victim's backup place. The figures show the following measurements:

- *Failure detection*: the time between killing a place and receiving an `Exception` at the root place (Figure 2, Line 27). The use of `finish` at Line 6 postpones throwing an exception until all activities in all places detect the failure and terminate. Thus the overhead here is not only from ULFM, but also from the termination detection protocol of `finish`.
- *Team reconstruction*: the time to reconstruct the Team object over a new set of places.
- *Remake*: the time to reconstruct the distributed objects over a new set of places.
- *Data restoration*: the time to reinitialize the application state from the latest checkpoint.
- *Restore agreement*: the time taken to reach a consensus between places on the success or failure of restoring their data.

The figures show that the time for failure detection and team reconstruction scale linearly with the number of places. The completion of a restore agreement depends on the completion of data restoration at all places. That is why when data restoration is slow, the agreement time increases. The times to remake PageRank and LinReg distributed vectors and matrices have better scalability than LULESH. LULESH's remake routine is currently not optimized and suffers low scalability. Each place in LULESH needs to maintain global pointers to ghost cell buffers in all neighboring places. When places die, some of these pointers will dangle and require updating. During remake, each place interacts with all its 26 neighbors to update its global pointers. All these remote operations result in interactions with place zero resilient store, and cause high performance overhead.

6. Related Work

ULFM has been used in a variety of resilient MPI applications with different recovery strategies. Pauli et al. [18] used ULFM for resilient Monte Carlo simulations. Ali et al. [4, 5] implemented exact and approximate recovery methods for 2D and 3D PDE solvers over ULFM. Laguna et al. [15] provided a programmability evaluation for ULFM in the context of a resilient molecular dynamics application. To the best of our knowledge, our work is the first to evaluate ULFM in the context of a high level language.

Panagiotopoulou and Loidl [17] presented a prototype for supporting transparent fault tolerance for Chapel over GASNet. Because GASNet cannot tolerate failures, process failure was simulated using signals without actually killing the transport layer.

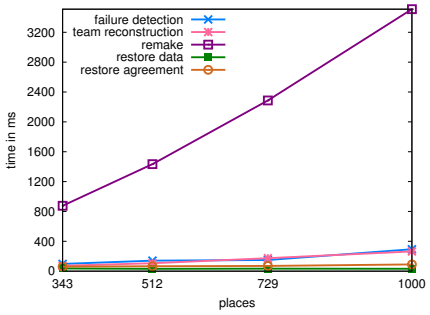

Figure 4: LULESH recovery times

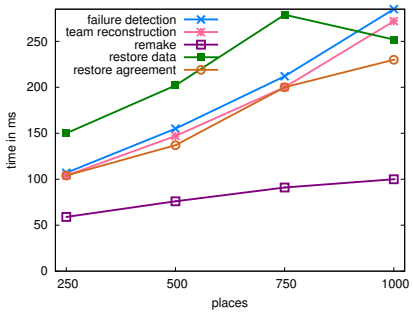

Figure 5: PageRank recovery times

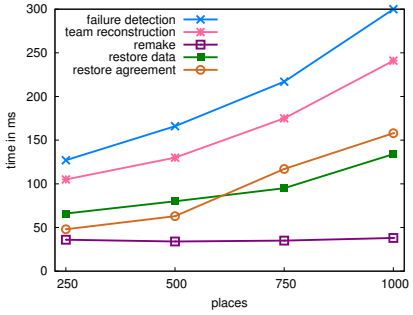

Figure 6: LinReg recovery times

7. Conclusion and Future Work

The MPI-ULFM specification provides a minimalistic user level fault tolerance support to MPI, which makes it a flexible base for resilient high level languages. We used ULFM to provide a resilient high performance transport for X10. Using ULFM, Resilient X10 applications can scale to larger problem sizes on error-prone environments. X10 provides elasticity support over its Managed backend. In the future, we plan to use ULFM's support for non-shrinking recovery to add elasticity support for Native X10.

Acknowledgment

This research was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under Award Number DE-SC0008923, and was undertaken with the assistance of resources from the National Computational Infrastructure (NCI), which is supported by the Australian Government.

References

[1] User Level Failure Mitigation Specification. `https://github.com/mpi-forum/mpi-issues/issues/20`, . Accessed March 13, 2016.

[2] Fault tolerance research hub. `http://fault-tolerance.org/ulfm`, . Accessed January 15, 2016.

[3] Chapel language specification version 0.98. Technical report, Cray Inc., Oct 2015.

[4] M. M. Ali, J. Southern, P. Strazdins, and B. Harding. Application level fault recovery: Using fault-tolerant Open MPI in a PDE solver. In *Proc. Parallel & Distributed Processing Symposium Workshops (IPDPSW)*, pages 1169–1178. IEEE, 2014.

[5] M. M. Ali, P. E. Strazdins, B. Harding, M. Hegland, and J. W. Larson. A fault-tolerant gyrokinetic plasma application using the sparse grid combination technique. In *Proc. International Conference on High Performance Computing & Simulation*, HPCS, pages 499–507, 2015.

[6] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. J. Dongarra. Post-failure recovery of MPI communication capability: Design and rationale. *International Journal of High Performance Computing Applications*, (3):244–254, 2013.

[7] W. Bland, H. Lu, S. Seo, and P. Balaji. Lessons learned implementing user-level failure mitigation in MPICH. In *Proc. 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 1123–1126. IEEE, 2015.

[8] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward exascale resilience. *International Journal of High Performance Computing Applications*, 23(4):374–388, 2009.

[9] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: The new adventures of old X10. In *Proc. 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 51–61, 2011.

[10] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioğlu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proc. 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538, 2005.

[11] S. Crafa, D. Cunningham, V. Saraswat, A. Shinnar, and O. Tardieu. Semantics of (resilient) X10. In *ECOOP 2014–Object-Oriented Programming*, pages 670–696. Springer, 2014.

[12] D. Cunningham, D. Grove, B. Herta, A. Iyengar, K. Kawachiya, H. Murata, V. Saraswat, M. Takeuchi, and O. Tardieu. Resilient X10: Efficient failure-aware programming. *ACM SIGPLAN Notices*, 49(8): 67–80, 2014.

[13] S. S. Hamouda, J. Milthorpe, P. E. Strazdins, and V. Saraswat. A resilient framework for iterative linear algebra applications in X10. In *Proc. 16th Parallel and Distributed Processing Symposium Workshop*, PDSEC 2015, pages 970–979. IEEE, 2015.

[14] V. Kumar, Y. Zheng, V. Cavé, Z. Budimlić, and V. Sarkar. Habanero-UPC++: A compiler-free PGAS library. In *Proc. 8th International Conference on Partitioned Global Address Space Programming Models*, pages 5:1–5:10, 2014.

[15] I. Laguna, D. F. Richards, T. Gamblin, M. Schulz, and B. R. de Supinski. Evaluating user-level fault tolerance for MPI applications. In *Proc. 21st European MPI Users' Group Meeting*, page 57. ACM, 2014.

[16] J. Milthorpe, D. Grove, B. Herta, and O. Tardieu. Exploring the APGAS programming model using the LULESH proxy application. Technical Report RC25555, IBM Research, 2015.

[17] K. Panagiotopoulou and H.-W. Loidl. Towards resilient Chapel: Design and implementation of a transparent resilience mechanism for Chapel. In *Proc. 3rd International Conference on Exascale Applications and Software*, pages 86–91. University of Edinburgh, 2015.

[18] S. Pauli, M. Kohler, and P. Arbenz. A fault tolerant implementation of Multi-Level Monte Carlo methods. In *Parallel Computing: Accelerating Computational Science and Engineering (PARCO)*, volume 13, pages 471–480, 2013.

[19] J. T. Richards, J. Brezin, C. B. Swart, and C. A. Halverson. A decade of progress in parallel programming productivity. *Communications of the ACM*, 57(11):60–66, Oct. 2014.

[20] V. Saraswat, G. Almasi, G. Bikshandi, C. Cascaval, D. Cunningham, D. Grove, S. Kodali, I. Peshansky, and O. Tardieu. The Asynchronous Partitioned Global Address Space Model. In *Proc. Workshop on Advances in Message Passing*, AMP'10, June 2010.

[21] O. Tardieu, B. Herta, D. Cunningham, D. Grove, P. Kambadur, V. Saraswat, A. Shinnar, M. Takeuchi, and M. Vaziri. X10 and APGAS at petascale. In *Proc. PPoPP 2014*, pages 53–66. ACM, 2014.

[22] C. Yang, K. Murthy, and J. Mellor-Crummey. Managing asynchronous operations in Coarray Fortran 2.0. In *Proc. IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 1321–1332, 2013.

[23] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick. UPC++: A PGAS extension for C++. In *Proc. Parallel and Distributed Processing Symposium*, IPDPS, pages 1105–1114, 2014.