A Resilient Framework for Iterative Linear Algebra Applications in X10

Sara S. Hamouda*, Josh Milthorpe†, Peter E. Strazdins*, Vijay Saraswat†
*Australian National University †IBM T.J. Watson Research Center
sara.salem@anu.edu.au, jjmiltho@us.ibm.com, peter.strazdins@cs.anu.edu.au, vsaraswa@us.ibm.com

***Abstract*—The Global Matrix Library (GML) is a distributed matrix library in the X10 language. GML is designed to simplify the development of scalable linear algebra applications. By hiding the communication and parallelism details, GML programs are written in a sequential style that is easy to use and understand by non expert programmers.**

Resilience is becoming a major challenge for HPC applications as the number of components in a typical system continues to increase. To address this challenge, we improved GML's adaptability to process failure and provided a mechanism for automatic data recovery. As iterative algorithms are commonly used in linear algebra applications, we also created a checkpoint/restore framework for developing resilient iterative applications using GML.

Using three example machine learning applications, we demonstrate that this framework supports resilient application development with minimal additional code compared to a non-resilient implementation. Performance measurements in a typical cluster environment show that the major cost of resilient execution is due to resilient X10 itself, and that the additional cost due to our framework is acceptable.

***Keywords*-X10; Resilience; Global Matrix Library; Checkpoint/Restart; Iterative Algorithms**

I. Introduction

Specialized matrix libraries are commonly used for rapid development of statistical and linear algebra applications. Systems such as MATLAB and R provide scientists with simple vector and matrix routines that enable them to easily build linear algebra applications and to simulate mathematical models in their domains.

The huge increase in the size of datasets of interest motivates the need for distributed matrix libraries that can exploit large scale clusters. However, distributed processing can be difficult for non-expert programmers to handle correctly due to the explicit handling of matrix distribution, data sharing, task coordination and load balancing. Ensuring resilient execution of applications adds further complexity as the size of computing clusters continues to increase.

Many researchers have used scalable fault tolerant data flow systems such as Hadoop [1] to implement resilient data intensive applications. Although the MapReduce model [2] applied by Hadoop is simple to use for aggregate operations, it is less suitable for implementing matrix operations [3], as the need to cast the matrix algorithms as map and reduce operations (such as in HaMa [4]) results in inefficient parallel algorithms [5]. Hadoop is also not suitable for implementing iterative algorithms [6], which are commonly used in linear algebra applications. Mahout [7] is a Hadoop based framework that provides high level building blocks for developing machine learning algorithms, but it inherits Hadoop's performance deficiencies of handling iterative algorithms [8]. Spark [9] and HaLoop [6] are data flow systems that use in-memory data caching for more efficient iterative processing, however, these systems do not expose a matrix-based data model to the application, which makes them harder to use than specialized matrix libraries.

In this paper, we describe a matrix library written in the X10 language that addresses the main challenges associated with distributed matrix computations: simplicity, performance, resilience and support for iterative computation.

X10 is a parallel programming language that is designed to bridge the gap between productivity and scalability. X10 supports resilience by detecting the failure of a portion of the system and notifying the application through an exception; this allows users to implement their own application level fault tolerance techniques to guard against fail-stop process failures [10]. X10 version 2.5.1 introduced support for dynamic process creation (Elastic X10). With resilience and elasticity, X10 programmers are able to implement applications that can adapt to the loss or addition of resources during the runtime.

X10 Global Matrix Library (GML) is a distributed matrix library that provides a rich set of matrix formats and routines, which makes it an attractive foundation for scalable linear algebra applications, as well as a potential compilation target for high-level array languages [11]. By hiding the details of communication and parallelism, GML programs are written in a sequential style that is easy to use and understand by non-expert programmers. Although GML inherits the performance and the efficient support for iterative algorithms from the X10 language, early versions were not resilient; failure of a single process caused the whole GML application to fail.

In this paper, we describe enhancements to GML to enable the development of resilient linear algebra applications with minimal impact on productivity. We improved GML's adaptability to the loss of resources by providing different restoration mechanisms that handle mapping the data partitions to a dynamically changing number of processes. As iterative algorithms are commonly used in linear algebra applications, a framework for developing resilient iterative applications was also implemented.

Our specific contributions are:

- A framework for developing resilient distributed linear algebra applications.
- Experimental evaluation of the performance overhead of our framework using different restoration modes.
- Experimental evaluation of the performance overhead of Resilient X10 bookkeeping activities.

II. The X10 Language

X10 is an Asynchronous Partitioned Global Address Space (APGAS) language. It simplifies distributed memory programming by exposing a single address space partitioned into regions called *places*. A place in the X10 language is an abstraction for an operating system process that contains a collection of data and threads operating on these data. The X10 type `x10.lang.Place` represents a place in the APGAS model, and `x10.lang.PlaceGroup` represents a collection of places.

Adding asynchrony to the PGAS model gives the programmer the control to dynamically create asynchronous tasks. The statement **async** `S` executes `S` in a new asynchronous task on the current place. `S` can also create other child async tasks. For synchronization, the **finish** construct is used to block the current task until all the nested tasks enclosed within the finish scope terminate. To access remote data or execute on a different place, the **at** construct is used. **at**`(p)S` executes the statement `S` at place `p`.

A task on a specific place can hold a reference to a (possibly) remote object in a value of type `x10.lang.GlobalRef`. However, the remote object can not be accessed directly; a running task must change to the place that holds the remote object using the **at** statement, then apply the operator '()' on the `GlobalRef` to access the referenced object. For example, the following code accesses a remote object of type `MyClass` using a global reference:

```
val gr:GlobalRef[MyClass] = ...;
at(gr) { val localObj:MyClass = gr(); }
```

This has the advantage of making the cost of remote access explicit to the programmer.

Similar to `GlobalRef`, values of type `x10.lang.PlaceLocalHandle` (PLH) are used to hold a reference to a family of objects, one per place in a `PlaceGroup` specified in the constructor. As with `GlobalRef`, the **at** statement and the '()' operator must be used to access these remote objects.

X10 can be compiled into C++ code (Native X10) or Java Code (Managed X10). For the communication layer, the user can currently select between sockets, MPI, or PAMI.

A. Resilient X10

Originally, a crash of a single place in X10 would cause the whole application to abort or to wait indefinitely for the completion of tasks on the crashed place. The failure rate is expected to increase in future systems as the number of components in the system increases; this motivated the development of a resilience extension for X10 [10]. Instead of waiting for the completion of tasks on a crashed place, the **finish** construct has been modified with extra bookkeeping activities to be able to detect the death of places, the termination of the orphan tasks, and to throw a `DeadPlaceException`. This gives the programmer the flexibility of handling this exception by a method suitable for the application.

The current version of X10 only supports resilient execution over the sockets communication layer. It is also based on the assumption that Place Zero is immortal, which means that a failure of Place Zero will cause a failure of the whole application. This is not a problem in practice, because even though the probability of some place failing increases with the number of places, the probability of a specific place failing (such as Place Zero) remains constant.

III. X10 Global Matrix Library

The Global Matrix Library (GML) provides powerful yet simple matrix and vector APIs that hide from the programmer the internal parallelism and distribution of the computation. Using GML interfaces, parallel and distributed linear algebra applications are written in a simple sequential style. GML is an object-oriented library; different implementations of matrices and vectors implement the same interface, so that client code is insulated from the details of the implementation. The choice of the particular implementation class is relevant only when the matrix or vector is created. We envisage, in future work, a layer above GML where this decision can be made automatically (through annotations supplied by the user, or through code analysis).

Listing 2 shows a GML implementation for the PageRank algorithm which is described in the pseudocode in Listing 1.

A. Matrix and Vector Classes

Table I lists some single place and multi-place classes available in GML. The classes in the "Duplicated" column store duplicates of the same vector or matrix data at multiple places, while the classes in the "Distributed" column partition a single data structure across multiple places.

The `Vector` class represents a single column of elements. `DupVector` represents a vector that is duplicated at a number of places. `DistVector` represents a vector partitioned into a number of segments, where each place holds one segment.

GML provides a variety of matrix representations by specializing the abstract `x10.matrix.Matrix` class. For single-place matrices, `DenseMatrix` represents a matrix in full dense storage format, and `SparseCSC` and `SparseCSR` represent sparse matrices stored in compressed-sparse-row and compressed-sparse-column formats, respectively.

For the multi-place matrices, `DupDenseMatrix` and `DupSparseMatrix` provide the functionality to duplicate a matrix in a number of places. The distributed classes provide abstractions for distributed block matrices. `DistSparseMatrix` and `DistDenseMatrix` partition the matrix by assigning one block to each place. In contrast, `DistBlockMatrix` assigns one or more blocks to each place. For doing so, it uses the `x10.matrix.distblock.BlockSet` class which is a container for a set of blocks. Allowing places to hold a set of blocks instead of a single block facilitates restoring the computation after a place failure by remapping the same matrix blocks among the remaining places without the need to repartition the matrix (more details in Section IV). Fig. 1 provides a graphical representation of the `DistVector`, `DupVector` and `DistBlockMatrix` classes.

Many matrix algorithms can be formulated as block algorithms, to exploit the parallelism available in multi-core and many-core systems. GML internally uses block algorithms to implement the matrix operations. The `x10.matrix.block.Grid` class is used by the distributed matrix classes to partition a matrix into blocks.

B. Creating GML Objects

All GML classes provide the factory method `make()` to create GML objects using the user specified configuration parameters. For example, the `DistBlockMatrix.make()` method used in Listing 2 (Line 3) allows the user to specify: the matrix dimensions (m,n), the data grid configuration (the number of row blocks and column blocks), and the place grid configuration (the number of row places and column places), which is used to map blocks to places.

Listing 1: PageRank pseudocode

```
for (z in 1..k) {
  $\mathbf{P} = \alpha\mathbf{GP} + (1-\alpha)\mathbf{EU}^T\mathbf{P}$
}
```

Listing 2: PageRank Implementation using GML

```
1  /* Input Data */
2  var m:Long, n:Long, rowBlocks:Long,
       colBlocks:Long, rowPlaces:Long,
       colPlaces:Long, alpha:Double;
3  val G:DistBlockMatrix = DistBlockMatrix.
       make(m, n, rowBlocks, colBlocks,
       rowPlaces, colPlaces);
4  val P:DupVector = DupVector.make(n);
5  val U:DistVector = DistVector.make(n,
       ...);
6  /*Temp Data*/
7  val GP:DistVector = DistVector.make(n,
       ...);
8
9  /* Data initialization code omitted */
10
11 /* PageRank Iterations */
12 for (1..k) {
13     GP.mult(G, P).scale(alpha);
14     val UtP1a = U.dot(P) * (1-alpha);
15     GP.copyTo(P.local()); // gather
16     P.local().cellAdd(UtP1a);
17     P.sync(); // broadcast
18 }
```

C. GML Resilience Limitations

1) No Tolerance for Place Failure: Previous versions of GML did not tolerate the loss of any place during the computation. Losing a place left the PLHs with dangling references to the dead places. The factory methods used a *fixed* place group containing all the original places to create the distributed objects. Also, the collective operations were designed to operate on all places regardless of their status. Due to these limitations, GML applications could not tolerate a place failure.

Table I: GML Vector and Matrix Classes

	Single-Place	Multi-Place	
		Duplicated	**Distributed**
Vector Classes	Vector	DupVector	DistVector
Matrix Classes	DenseMatrix SparseCSC SparseCSR	DupDenseMatrix DupSparseMatrix	DistDenseMatrix DistSparseMatrix DistBlockMatrix

2) No Built-in Mechanism For Restoring GML objects: There was no built-in mechanism to restore the lost portion of a GML object that was held by a failed place. The application had to be started from the beginning after the occurrence of any failure. Coding for fault tolerance was entirely the responsibility of the programmer, which reduced productivity.

IV. Resilient GML

In this section, we describe our improvements to GML, which enable the development of applications that can tolerate failures of one or more places (other than Place Zero) during execution.

A. Using Arbitrary Place Group

1) GML Dynamic Place Distribution: Using a fixed non-configurable place group limited the adaptability of the GML objects to the environment changes (loss or addition of

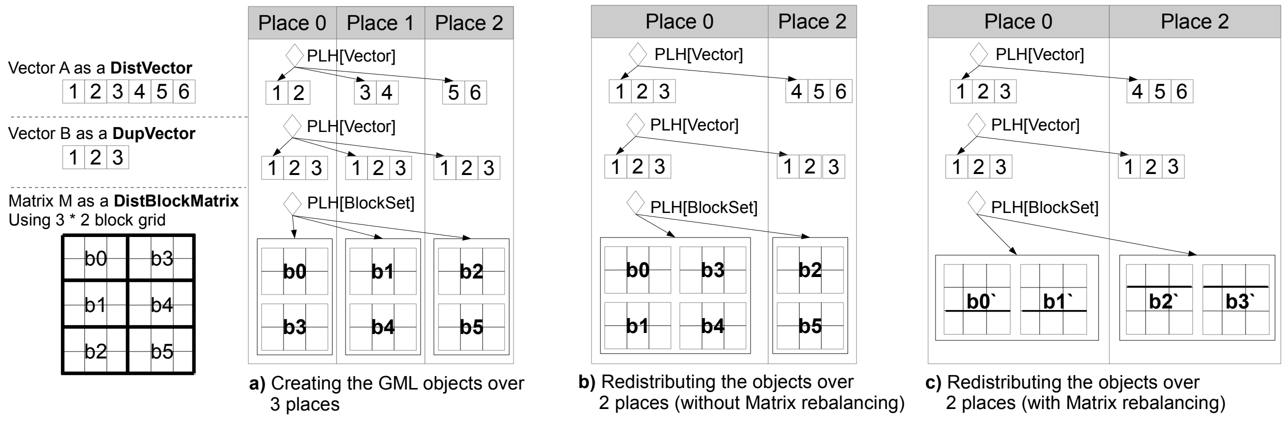
Place 0
Place 1
Place 2
Vector A as a DistVector
Vector B as a DupVector
Matrix M as a DistBlockMatrix Using 3 * 2 block grid
PLH[Vector]
PLH[BlockSet]
a) Creating the GML objects over 3 places
b) Redistributing the objects over 2 places (without Matrix rebalancing)
c) Redistributing the objects over 2 places (with Matrix rebalancing)

Figure 1: Graphical Representation for the Distribution of the DupVector, DistVector and DistBlockMatrix

places). To overcome this problem, we allowed the use of arbitrary place groups for the construction of the multi-place classes. Instead of using all the available places, the user can specify a subset of places for creating a GML object.

The object's distribution can also be modified during execution; the `remake(newPlaces:PlaceGroup)` method was added to the GML objects for this purpose.

2) Dynamic Distribution and Load Balancing: For the duplicated classes in Table I, changing the `PlaceGroup` simply means duplicating the vector/matrix on a different number of places. However, changing the `PlaceGroup` for the distributed classes requires careful handling for load balancing.

Classes that assign one block to each place, such as `DistDenseMatrix`, `DistSparseMatrix`, and `DistVector`, must recalculate the data grid to generate new blocks equal in number to the size of the new `PlaceGroup`. On the other hand, classes that assign more than one block per place (i.e. `DistBlockMatrix`) do not necessarily need to recalculate the data grid. If the data grid is kept the same after distribution, only the block-to-place mapping needs to be updated. However this can lead to load imbalance when using a different number of places.

To ensure an even data distribution between places, GML requires repartitioning the object based on the size of the new place group. Fig. 1 shows an example for distributing a `DupVector`, a `DistVector` and a `DistBlockMatrix` over a smaller number of places. The difference in load balance resulting from either keeping or changing the data grid of a `DistBlockMatrix` is shown in Fig. 1-b and Fig. 1-c, respectively.

Changing the data grid results in better load balancing but it also alters the blocks' configurations (the number and the dimensions of the blocks), which complicates the restore operation, as will be described in Section IV-B.

Listing 3: GML Objects Snapshot/Restore API

```
interface Snapshottable {
  def makeSnapshot():Snapshot;
  def restoreSnapshot(snapshot:Snapshot):
      void;
}
```

B. Snapshot/Restore Mechanism for GML objects

When the `remake(newPlaces:PlaceGroup)` method is called to redistribute a GML object, it destroys the previously allocated memory for the object and reallocates memory on a new `PlaceGroup`. In order to restore a GML object's data, we added the capability to save a snapshot of the object into a resilient store, and then use the saved snapshot to restore the object's data. The snapshot/restore capability is provided through the `Snapshottable` interface shown in Listing 3.

1) Snapshot Capability: A `Snapshottable` GML object implements the `makeSnapshot()` method which returns a `Snapshot` object. The `Snapshot` object stores the GML object's state as key/value pairs, where key is the index of the place in the `PlaceGroup`, and value is the portion of the object's data stored on that place. When the GML object is restored to a reduced number of places, the identifiers of the remaining places will remain unchanged, but the index of some places will be shifted due to filtering out the dead places.

The `Snapshot` object applies a double in-memory storage mechanism for the key/value pairs; a copy of each key/value pair is saved in the local place, and a backup copy is saved in the next place. The cost of saving data to a `Snapshot` object is uniform from any place; it equals to the cost of saving the local copy plus the cost of saving the remote copy. However, the cost of loading data is not uniform, as it depends on whether the requested data exists on the local memory of the loading place.

2) Restore Capability: The restore capability is added to the GML objects by implementing the `restoreSnapshot (Snapshot)` method from the `Snapshottable` interface.

Restoring a duplicated class (such as `DupVector`) is done by concurrently loading a duplicate for each place from the `Snapshot` object given the new index of the place as the key.

Restoring a distributed class (such as `DistVector`) is done by concurrently loading each place's data partition from the `Snapshot` object. The performance of restoring a distributed class depends on whether the object is using the same or different data grid after a failure. If the object's partitioning is unchanged, this means that the blocks on the new distribution have the same configurations as the blocks on the old distribution (before the failure). Each place restores its portion of the object's data by copying whole blocks from the snapshot object to the GML object (block-by-block restore). This is applicable when the object is restored using the same number of places. It is also applicable for the `DistBlockMatrix` when the data grid remains the same (see Fig. 1-b).

Restoring a distributed object is more complex when the object's data grid is changed. In that case, the number and the dimensions of the blocks will be different from the block configuration at the snapshot time (see Fig. 1-c). A single block on the new distribution can overlap with many other blocks on the old distribution. Each place needs to calculate these overlapping regions to restore its data portion by copying sub blocks from the overlapping places. For the sparse matrices, the non-zero elements for the overlapping regions must be counted to determine the space required for the new sparse block, which adds more overheads for restoring a repartitioned sparse matrix.

V. The Resilient Iterative Application Framework

In this section, we describe a framework that simplifies the development of resilient iterative algorithms. We also describe three restoration modes provided by the framework to give the user the flexibility to select how the application should adapt to the loss of places.

The framework applies the Coordinated Checkpoint/ Restart technique. In this technique, all the participating processes periodically pause their processing in order to create a consistent checkpoint for the application. Should one process fail, the application can be recovered by restarting the application from the latest checkpoint. In coordinated checkpointing, it is not necessary to save more than one checkpoint for the application; after a new checkpoint is successfully taken, the old one can be deleted.

For some applications, it might be difficult to specify where in the program a checkpoint should be taken. However, for the iterative algorithms, a checkpoint is usually taken either at the beginning or at the end of the iteration body.

The checkpointing interval should be carefully chosen to balance the trade-off between the checkpointing overhead and the recovery overhead. Young's formula may be used to determine the checkpointing interval: $\sqrt{2 * \text{Time}_{\text{checkpoint}} * \text{MTTF}}$, where MTTF is the mean time to failure [12].

A. Framework Description

The implementation of the framework is based on three main modules: 1) the application resilient store, 2) the iterative application programming model, and 3) the application executor.

1) Application Resilient Store: In Section IV-B, we explained how the `Snapshot` class is used to store the state of a *single* GML object. In order to create a coherent checkpoint for a GML application, it is important to take a snapshot for *every* object that contributes to the application's state. Creating a checkpoint for the application should be done in an atomic manner; the application snapshot is considered valid only if the snapshots of all the involved GML objects are successfully created. To make this process easier for the application developer, the framework provides the `AppResilientStore` class shown in Listing 4, which creates consistent application snapshots, restores the application from the latest snapshot, and deletes the old unneeded snapshot(s).

Listing 5 shows an example for using the `AppResilientStore` to snapshot/restore the PageRank program in Listing 2. Taking a snapshot for the application is done in lines 3-7. First `startNewSnapshot()` is called, followed by calling `save()` or `saveReadOnly()` for each GML object contributing to the application's state, followed by calling the `commit()` method. The `saveReadOnly()` method is used with the objects that do not change during the computation. If there is an existing snapshot for a read-only object, the `saveReadOnly()` method will reuse this snapshot, otherwise, a snapshot will be created. Restoring the application state using a different place group is done in lines 9-14. The multi-place objects are first updated to use the new place group `(newPlaces)` by calling the `remake()` method (lines 10-13), then a single call to the `restore()` method is used to restore all the saved GML objects (`G, U, and P`) in parallel.

2) Iterative Application Programming Model: The GML iterative application framework requires the application developer to follow a specific programming model for developing an iterative algorithm. By limiting the programming model, it becomes easier to provide fault tolerance for the application with a higher level of transparency from the programmer. For example, limiting the programming model to the simple MapReduce model allows Hadoop to provide transparent fault tolerance for its applications.

Listing 4: Application Resilient Store API

```
class AppResilientStore {
  /* Array of application snapshots */
  val snapshots:Rail[AppSnapshot];

  def startNewSnapshot();
  def save(obj:Snapshottable);
  def saveReadOnly(obj:Snapshottable);
  def commit();
  def cancelSnapshot();
  def restore();

  class AppSnapshot{
    val map:HashMap[Snapshottable,
        Snapshot];
  }
}
```

Listing 5: Using AppResilientStore to snapshot/restore the PageRank Program in Listing 2

```
1  val store:AppResilientStore;
2  // Take an application snapshot
3  store.startNewSnapshot();
4  store.saveReadOnly(G);
5  store.saveReadOnly(U);
6  store.save(P);
7  store.commit();
8  // Restore an application snapshot
9  val newPlaces:PlaceGroup;
10 G.remake(..., newPlaces);
11 U.remake(..., newPlaces);
12 P.remake(newPlaces);
13 GP.remake(..., newPlaces);
14 store.restore();
```

Our framework requires the application developer to implement the following four methods:

`isFinished():Boolean` - this method should evaluate the algorithm's termination condition (for example by checking the number of completed iterations or by checking a convergence condition).

`step():void` - this method should contain the body of the iterative algorithm which will be repeated until `isFinished()` evaluates to true.

`checkpoint(store:AppResilientStore):void` - this method should contain the steps required to save the states of the GML objects into the `AppResilientStore` object passed as a parameter (i.e. Lines 3-7 in Listing 5).

`restore(newPlaces:PlaceGroup, store: AppResilientStore, snapshotIter:Long):void` - this method should contain the steps required to rollback the application to the state of the snapshot iteration (the third parameter). The restore method should use the provided place group (first parameter) and the application resilient store (second parameter) to remake and restore the used GML objects (i.e. Lines 9-14 in Listing 5).

Table II: Lines of code comparison between the non-resilient and resilient versions of the benchmark programs

Application	Non-resilient Total	Resilient Total	Resilient Checkpoint	Resilient Restore
LinReg	66	96	10	16
LogReg	166	222	11	20
PageRank	72	94	7	10

We applied this programming model to three GML benchmarking applications. Table II shows a lines of code (LOC) comparison between the non-resilient versions and the resilient versions. For the resilient code, the table shows the total LOC, the `checkpoint` method LOC, and the `restore` method LOC. All applications had 3 lines of code for the `isFinished` method. The remaining lines of code that are not shown in the table are related to defining application specific variables, data initialization, and the iterative algorithm, which are almost unchanged for the purpose of resilience. The numbers in Table II show that the programming effort required to add resilience support is minimal. Further simplification can be done in the future by providing annotations to mark the snapshot objects to eliminate the need for defining the `checkpoint` and `restore` methods.

3) Iterative Application Resilient Executor: The resilient executor module is the component that executes GML applications following our framework's programming model: it executes the `step()` method in a loop, it calls the `isFinished()` method to check the termination condition of the loop, it calls the `checkpoint()` method according to the provided checkpoint interval, and it calls the `restore()` method when a place failure is detected.

B. Restoration Modes

The resilient executor provides three modes for restoring the application that define how the application should adapt to the loss of places.

1) Shrink Mode: In this mode, the executor restores the application using the remaining live places. If the application is using a `DistBlockMatrix` object, restoring the matrix will be done using the same block partitioning. It provides faster block-by-block restore, but can lead to load imbalance.

2) Shrink-Rebalance Mode: In this mode, the executor restores the application using the remaining live places. If the application is using a `DistBlockMatrix` object, the matrix will be repartitioned to provide even load balancing for the remaining places.

3) Replace-Redundant Mode: In this mode, a failed place will be replaced by a spare place. At the application start time, the user can select to create a number of redundant places to be used in case of failure. Although starting redundant places reduces the utilization of the allocated system resources, this approach has the advantage of allowing the executor to restore the application using the same number of places. As a result, there will be no need to rebalance the load between places, because the load distribution will remain the same after the failure. When the number of failed places exceeds the number of redundant places, the executor will have to apply the shrink mode or the shrink-rebalance mode based on the user's choice.

In the future, we plan to add a fourth mode (Replace-Elastic) that utilizes the new elasticity feature of X10. Using elastic X10, we can dynamically create new places to replace the failed ones instead of creating redundant places in advance.

VI. Related Work

Many programming languages, systems and libraries can be used to implement linear algebra algorithms with different levels of simplicity, efficiency and resilience. In the following, we review some of the options ranging from low level languages, such as MPI, to high level matrix languages.

A. Message Passing Systems

MPI is the de facto standard for developing high performance applications including linear algebra algorithms. With the reliability issues of large scale clusters, lots of efforts have been made towards fault-tolerant MPI. FT-MPI [13] and OpenMPI-ULFM[14] are examples for MPI implementations that provide APIs to rebuild or shrink the MPI communicator after a failure. FMI [15] is another example of a message passing runtime that provides automatic recovery for MPI communicators using spare processes. MPI's low level programming model is complex to use, especially for large applications with complex algorithms. As a result, research into higher level programming models (like APGAS and MapReduce) is gaining more interest.

The Global Arrays (GA) library [16] applies the Global Address Space model by extending MPI with shared array abstractions. Fault tolerance support has been added to GA through checkpoint/restart [17], matrix encoding techniques [18], and redundant communication [19]. Like GA, GML provides distributed data structures and methods to manipulate them, however, unlike GA, GML is implemented in a high level APGAS language. Providing a resilient simple to use matrix library for X10 expands the language's applicability, and makes X10 a more convenient alternative for MPI.

B. Data Flow Systems

The MapReduce data flow model [2] provides a high level abstraction for distributed processing. The simplicity of the programming model, in addition to the scalability and the transparent fault tolerance support are the main reasons behind the wide adoption of MapReduce based systems such as Hadoop. In spite of the success achieved by MapReduce systems in supporting aggregate computations, the MapReduce model is not a natural fit for matrix computations [3]. In addition, implementing iterative algorithms as repeated calls to MapReduce jobs is inefficient because of the encountered I/O overhead due to reloading the intermediate data from reliable storage at each iteration [6].

Spark [9] is a data flow system that provides reliable and efficient support for iterative algorithms by storing the intermediate data in memory. Spark's fault tolerance approach is based on recording the coarse-grained transformations done on the data partitions to enable recomputing any lost partition. Similar to Hadoop, the Spark programming model is not matrix based, thus it is more suitable for aggregate computations than for matrix based computations.

C. Matrix Based Languages and Libraries

R and MATLAB are matrix processing frameworks that are widely used by researchers for prototyping small scale matrix algorithms [8]. There are some attempts to improve the parallelism and scalability of such frameworks. Presto [3] overcomes the scalability problem of R by extending it with a distributed array abstraction *darray* and with APIs for partitioning and repartitioning the distributed arrays for automatic rebalancing. Presto requires the programmer to be aware of the distributed nature of the matrix algorithm and to express the algorithm by iterating over the individual array blocks. Although this approach provides high flexibility for implementing complex distributed matrix algorithms, it comes with the cost of reduced productivity.

MadLinq [5] is a distributed matrix library for the .NET language. The parallel execution of the block matrix computation is done through a data flow engine, which transforms the computation into a task DAG (Directed Acyclic Graph). Although generating a task DAG can improve the performance and processor utilization by applying pipelined based scheduling, it can also introduce extra overheads for matrix computations with regular parallelism patterns.

VII. Experiments

To evaluate the performance of our resilient linear algebra application framework, we measured: 1) the overall performance of three applications running in non-resilient mode, 2) the overhead due to the bookkeeping activities of resilient X10, 3) the checkpoint overhead, and 4) the restore overhead for a single failure using the different restoration modes.

All timing experiments were conducted on a SoftLayer cluster of 11 nodes hosted at IBM Almaden Research Center. Each node has a single four-core 2.6 GHz Intel Xeon E5-2650 CPU with 8 GB of memory; four Native X10 places were run per node with a single worker thread per place (`X10_NTHREADS=1`). The X10 compiler and runtime were built from source version 2.5.2, and OpenBLAS version 0.2.13 was used for BLAS operations, also with a single thread per place (`OPENBLAS_NUM_THREADS=1`). Our full source code is freely available at http://x10-lang.org as part of GML version 2.5.2.

A. *Resilient X10 Overhead*

To measure the overhead due to resilient X10, we ran three GML benchmark codes – Linear Regression, Logistic Regression and PageRank – under both non-resilient and resilient X10. All are iterative codes; we performed 30 test runs of 30 iterations each, and we report the mean, min and max times across all runs.

The Linear Regression (LinReg) benchmark trains a linear regression model against a set of labeled training examples; in these experiments the training examples are stored as a dense `DistBlockMatrix`. We trained a model of 500 features with a training set size of 50,000 examples per place (weak scaling) on 2 to 44 places. Fig. 2 shows the time per iteration for LinReg with non-resilient X10 compared to resilient X10.

With standard non-resilient X10, the time per iteration increases from 60 ms on two places to 180 ms on 44 places. With resilient X10, time per iteration increases more rapidly with number of places from 60 ms to 400 ms, an overhead of up to 120%. The increasing cost of resilient X10 with number of places is due to communication with place 0 for activity bookkeeping, which has previously been identified as a scalability bottleneck for place-zero-based resilient **`finish`** [10].

The Logistic Regression (LogReg) benchmark trains a binary classification model against a set of labeled training examples stored as a dense `DistBlockMatrix`. Fig. 3 shows the time per iteration for LogReg with non-resilient X10 compared to resilient X10.

With standard non-resilient X10, the time per iteration increases from 110 ms on two places to 295 ms on 44 places. The overhead of resilient X10 is relatively less with LogReg than with LinReg: time per iteration increases from 110 ms to 595 ms, an overhead of up to 100%.

The PageRank benchmark computes a measure of centrality for each node in a linked network, the structure of which is represented in GML as a sparse `DistBlockMatrix`. We performed 30 iterations of the PageRank algorithm for a network of 2M edges per place (weak scaling) on 2 to 44 places. Fig. 4 shows the time per iteration for PageRank with non-resilient X10 compared to resilient X10.

With standard non-resilient X10, the time per iteration increases from 38 ms on two places to 360 ms on 44 places. As there are fewer **`finish`** constructs used in PageRank than in LinReg or LogReg, the overhead of resilient X10 is minimal: time per iteration increases from 38 ms to 370 ms, an overhead of less than 5%.

B. *Checkpointing Overhead*

We next examine the cost of performing checkpoints using the resilient application framework. We ran the resilient

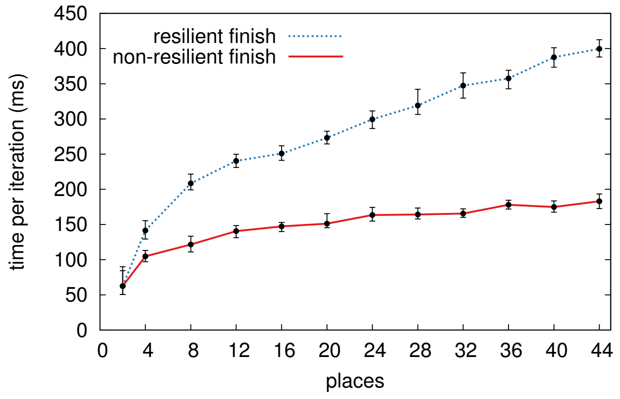

Figure 2: Linear Regression: resilient X10 overhead

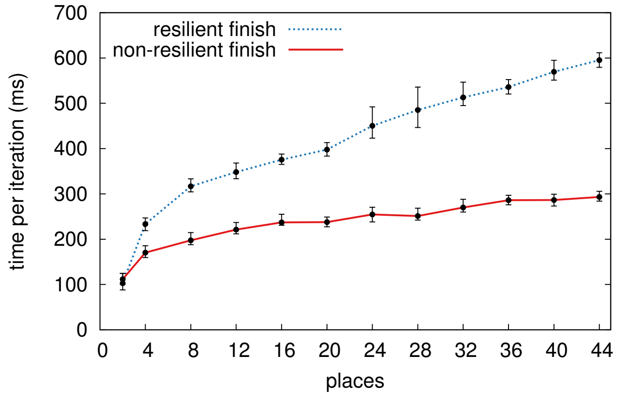

Figure 3: Logistic Regression: resilient X10 overhead

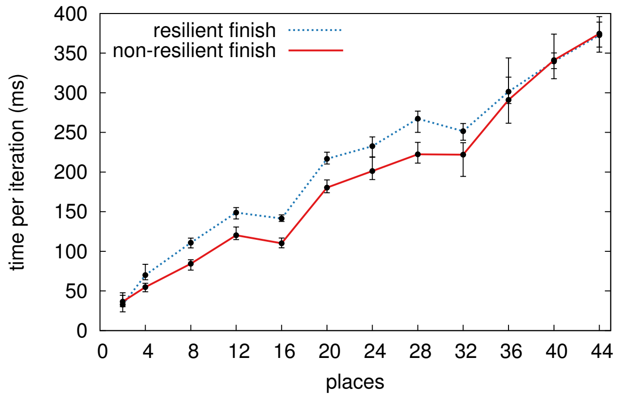

Figure 4: PageRank: resilient X10 overhead

GML applications taking a checkpoint once every ten iterations (three checkpoints per run). Table III shows the mean time per checkpoint across 30 runs for the same problem sizes used in the weak scaling experiments described above.

Table III: Time per checkpoint for resilient GML apps

	Mean checkpoint time (ms)		
places	LinReg	LogReg	PageRank
2	1284	1288	241
4	1819	1819	332
8	1917	1945	358
12	2292	2354	451
16	2289	2361	462
20	2293	2368	469
24	2336	2350	478
28	2356	2385	480
32	2377	2415	500
36	2358	2401	516
40	2474	2510	522
44	2464	2534	534

At 44 places, checkpoint times for both LinReg and LogReg are around $2.5\,\mathrm{s}$, the time required to compute around 6 and 4 iterations, respectively. The checkpoint time for PageRank on 44 places is $534\,\mathrm{ms}$, the time required to compute around 1.5 iterations. For all three applications, time per checkpoint increases by less than 20% from 12 to 44 places, which suggests that the distributed checkpoint algorithm is scalable. All of the three applications use a read-only matrix as input. Using the `saveReadOnly()` with such objects reduces the overall checkpointing overhead as their snapshot will be created only once (in the first checkpoint).

C. Restore Overhead

We now consider the increase in runtime due to checkpointing, restoration and (optionally) rebalancing when running in each of the three different restoration modes – shrink, shrink–rebalance, and replace-redundant – described in §V-B.

Fig. 5 shows the total runtime for 30 iterations of LinReg, where checkpoints are taken every 10 iterations and a single place failure occurs at iteration 15. The total runtime includes the overhead of resilient X10, checkpointing, restoration and (for the shrink-rebalance mode only) rebalancing. The total time for non-resilient execution without failure is also shown as a baseline. Fig. 6 and Fig. 7 show the corresponding timings for LogReg and PageRank.

Table IV shows the percentage of the total time consumed by the checkpoint and the restore operations on 44 places for the experiments in Fig. 5, 6 and 7. The replace-redundant and the shrink modes produce lower overhead on the overall execution time compared to the shrink-rebalance mode. The shrink-rebalance mode has the highest overhead due to vector/matrix repartitioning and having to copy multiple sub-blocks to restore a single block.

Table IV: Percentage of total time consumed by checkpoint (C%) and restore (R%) operations on 44 places

	Shrink		Shrink-Rebalance		Replace-Redundant	
Application	C%	R%	C%	R%	C%	R%
LinReg	32	18	25	22	36	7
LogReg	26	15	19	22	27	16
PageRank	10	7	10	10	11	4

VIII. Conclusion and Future Work

In this paper we presented a resilience extension to X10 Global Matrix Library. Resilient GML objects can be mapped to a dynamically changing number of processes, and can be saved and restored using the same number or fewer processes. Using X10's resilient runtime and the snapshot/restore capability of GML objects, resilience support can be added to GML programs with minimal programming effort.

Although the paper is about a library in the X10 language, our approach is also applicable to other languages. The resilient application framework is generic enough to be easily implemented in other languages or reused by other X10 libraries such as ScaleGraph [20].

In the future, we plan to use Resilient GML in more resilient applications, improve the performance and compare it with other frameworks. We also plan to add other fault tolerance techniques, and to use Elastic X10 to achieve fast restoration without sacrificing process utilization.

Acknowledgment

A number of researchers at IBM have contributed ideas or code to the material presented in this paper. The initial version of GML was developed by Juemin Zhang (with Vijay Saraswat). We also wish to thank Kiyokuni Kawachiya, Arun Iyengar and Min Li.

References

[1] T. White, *Hadoop: the definitive guide*. O'Reilly Media, Inc., 2009.

[2] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber, "Presto: distributed machine learning and graph processing with sparse matrices," in *Proc. 8th ACM European Conference on Computer Systems*. ACM, 2013, pp. 197–210.

[4] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng, "HAMA: An efficient matrix computation with the MapReduce framework," in *Proc. 2nd International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2010, pp. 721–726.

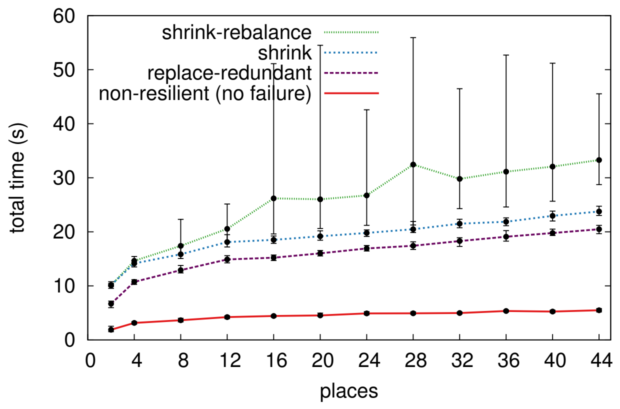

Figure 5: Linear Regression: total runtime with a single failure using different restoration modes

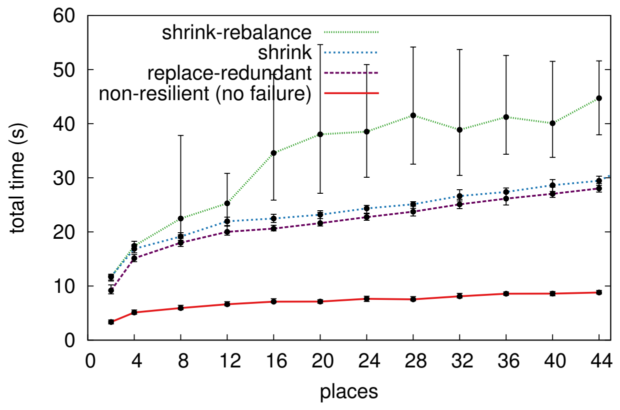

Figure 6: Logistic Regression: total runtime with a single failure using different restoration modes

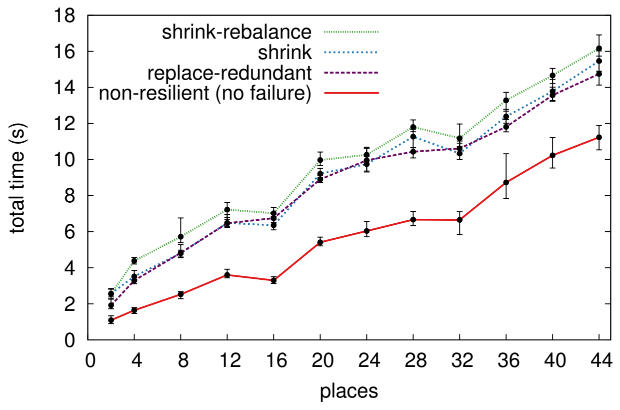

Figure 7: PageRank: total runtime with a single failure using different restoration modes

[5] Z. Qian, X. Chen, N. Kang, M. Chen, Y. Yu, T. Moscibroda, and Z. Zhang, “MadLINQ: large-scale distributed matrix computation for the cloud,” in *Proc. 7th ACM European Conference on Computer Systems*. ACM, 2012, pp. 197–210.

[6] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, “HaLoop: Efficient iterative data processing on large clusters,” *Proc. VLDB Endowment*, vol. 3, no. 1-2, pp. 285–296, 2010.

[7] “Apache Mahout,” http://mahout.apache.org, accessed January 21, 2014.

[8] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska, “MLI: An API for distributed machine learning,” in *Proc. 13th International Conference on Data Mining (ICDM)*. IEEE, 2013, pp. 1187–1192.

[9] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *Proc. 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, 2012, pp. 15–28.

[10] 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,” in *Proc. 19th ACM SIGPLAN Symposium on Principles and Practice Of Parallel Programming*. ACM, 2014, pp. 67–80.

[11] D. Grove, J. Milthorpe, and O. Tardieu, “Supporting array programming in X10,” in *Proc. ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ACM, 2014, pp. 38–43.

[12] J. W. Young, “A first order approximation to the optimum checkpoint interval,” *Communications of the ACM*, vol. 17, no. 9, pp. 530–531, 1974.

[13] G. E. Fagg and J. J. Dongarra, “FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world,” in *Recent advances in parallel virtual machine and message passing interface*. Springer, 2000, pp. 346–353.

[14] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. J. Dongarra, “An evaluation of user-level failure mitigation support in MPI,” in *Proc. EuroMPI’12*, 2012, pp. 193–203.

[15] K. Sato, A. Moody, K. Mohror, T. Gamblin, B. R. de Supinski, N. Maruyama, and S. Matsuoka, “FMI: Fault tolerant messaging interface for fast and transparent recovery,” in *Proc. 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 1225–1234.

[16] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, “Global Arrays: A nonuniform memory access programming model for high-performance computers,” *The Journal of Supercomputing*, vol. 10, no. 2, pp. 169–189, 1996.

[17] V. Tipparaju, M. Krishnan, B. Palmer, F. Petrini, and J. Nieplocha, “Towards fault resilient Global Arrays,” *Parallel computing: architectures, algorithms, and applications*, pp. 339–345, 2008.

[18] N. Ali, S. Krishnamoorthy, M. Halappanavar, and J. Daily, "Tolerating correlated failures for generalized Cartesian distributions via bipartite matching," in *Proc. 8th ACM International Conference on Computing Frontiers*. ACM, 2011.

[19] N. Ali, S. Krishnamoorthy, N. Govind, and B. Palmer, "A redundant communication approach to scalable fault tolerance in PGAS programming models," in *Proc. 19th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 2011, pp. 24–31.

[20] M. Dayarathna, C. Houngkaew, and T. Suzumura, "Introducing ScaleGraph: an X10 library for billion scale graph analytics," in *Proc. ACM SIGPLAN X10 Workshop*. ACM, 2012, pp. 6:1–6:9.