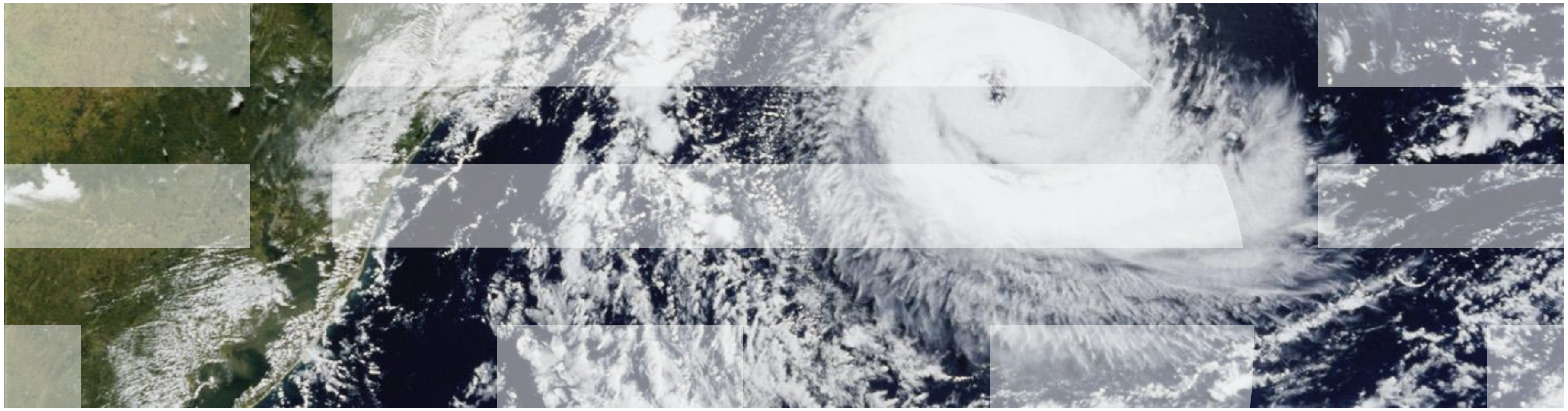
Dave Cunningham, Dave Grove, Ben Herta, Arun Iyengar , Kiyokuni Kawachiya, Hiroki Murata, Vijay Saraswat ,
Mikio Takeuchi, Olivier Tardieu

PPoPP 2014

Resilient X10
Efficient failure-aware programming



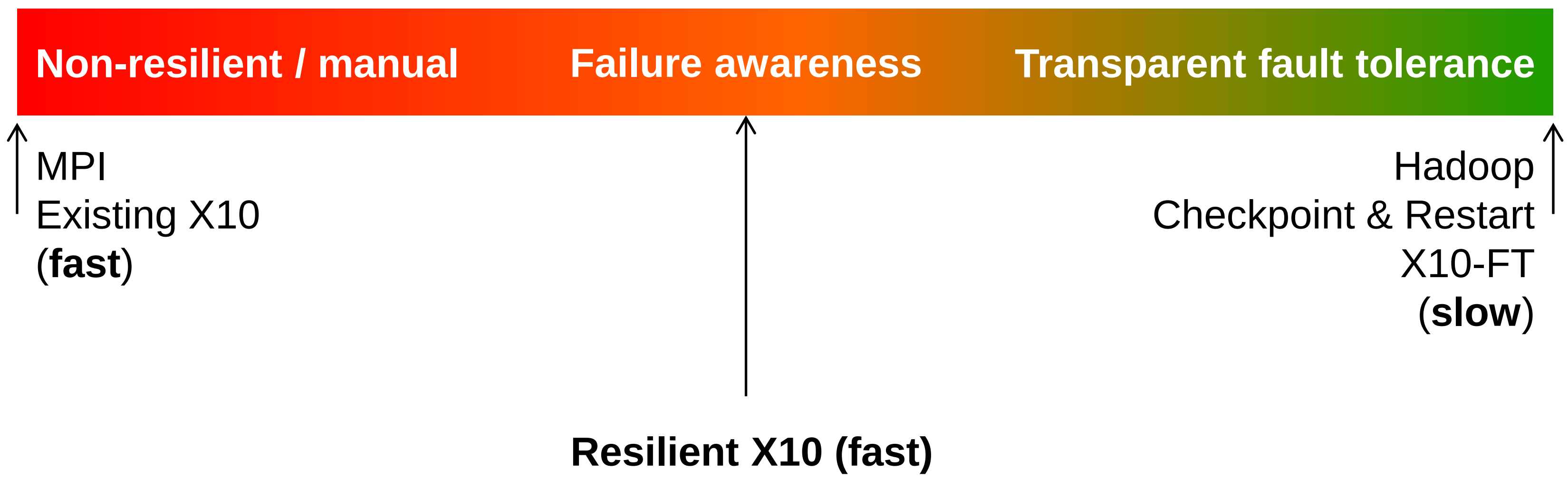
Resiliency Spectrum

Node failure is a reality on commodity clusters

- Hardware failure
- Memory errors, leaks, race conditions (including in the kernel)
- Evictions
- Evidence: Popularity of Hadoop

Ignoring failures causes serial MTBF aggregation:
24 hour run, 1000 nodes,
6 month node MTBF
=> under 1% success rate

Transparent checkpointing causes significant overhead.



Resilient X10 Overview

Provide helpful semantics:

- Failure reporting
- Continuing execution on unaffected nodes
- Preservation of synchronization: *HBI* principle (described later)

Application-level failure recovery, use domain knowledge

- If the computation is approximate: *trade accuracy for reliability (e.g. Rinard, ICS06)*
- If the computation is repeatable: *replay it*
- If lost data is unmodified: *reload it*
- If data is mutated: *checkpoint it*
- Libraries can hide, abstract, or expose faults (e.g. containment domains)
- Can capture common patterns (e.g. map reduce) via application frameworks

No changes to the language, substantial changes to the runtime implementation

- Use exceptions to report failure
- Existing exception semantics give strong synchronization guarantees

X10 Language Overview (Non-distributed features)

- Java-like language
- Developed ~ 10 years (open source)
- Structs (compound value types)
- Reified Generics

```
struct Complex {
    val real : Double;
    val imag : Double;
}
class Test {
    public static def myMethod(p:Complex) : Complex {
        return Complex(p.real+1, p.imag-1);
    }
    public static def myMethod2(p:Rail[Complex]) {
        for (i in 0..(p.size-1)) {
            p(i) = myMethod(p(i));
        }
    }
}
```

- Activities
  - Lightweight threads
  - Exception propagation
  - Atomic construct

```
class Test {
    public static def main(args: Rail[String]) {
        finish {
            async {
                Console.OUT.println("1a");
            }
            async {
                Console.OUT.println("1b");
            }
        }
        Console.OUT.println("2");
    }
}
```

Possible interleavings:

- 1a
- 1b
- 2

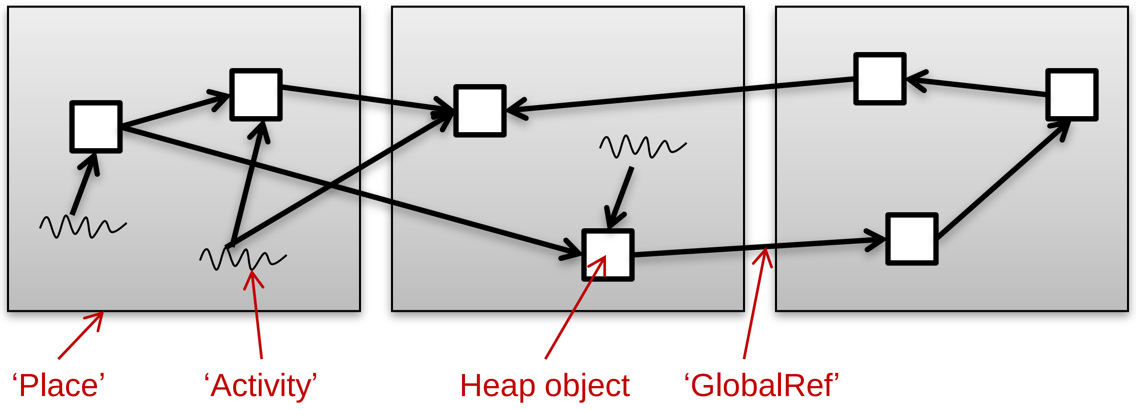
Or:

- 1b
- 1a
- 2

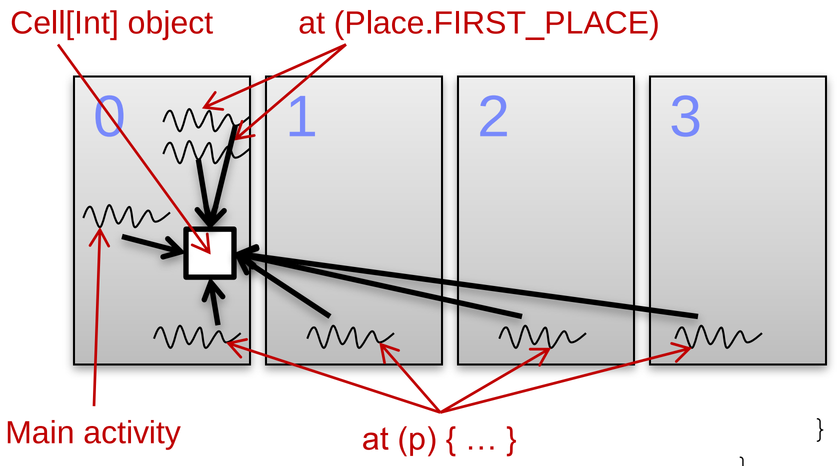
X10 Language Overview (Distributed Features)

- Scales to 1000s of nodes
- Asynchronous PGAS (APGAS)
  - Heap partitioned into 'places'
  - Can only dereference locally
- Explicit communication
- Implicit object graph serialization

```
val x = ...;
val y = ...;
at (p) {
    val tmp = x + y;
}
```

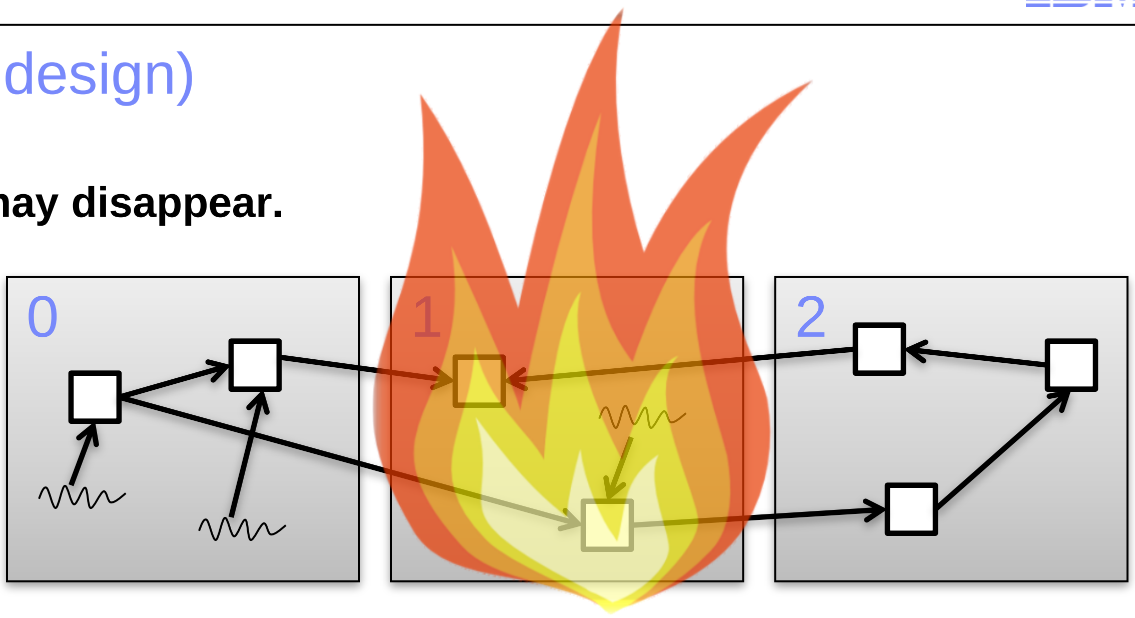


```
class MyClass {
    public static def main(args:Rail[String]):void {
        val c = GlobalRef(new Cell[Long](0));
        finish {
            for (p in Place.places()) {
                at (p) {
                    async {
                        val v = ...; // non-trivial work
                        at (Place.FIRST_PLACE) {
                            val cell = c();
                            atomic { cell(cell() + v); }
        }    }    }    }    }
        // Runs after remote activities terminate
        Console.OUT.println("Cumulative value: "+c()());
    }
}
```



Resilient X10 (Language design)

Sometimes, an arbitrary place may disappear.



Immediate Consequences:

- The heap at that place is lost
- The activities are lost
- Any 'at' in progress immediately terminates with `x10.lang.DeadPlaceException`

(Very similar to java.lang.VirtualMachineError)

Lasting Consequences:

Place will never come back alive.

Can no-longer at (dead_place) {…} – get DeadPlaceException thrown.

GlobalRef[T] to objects at that place may still be dangling…

But type system requires use of 'at' to access that state.

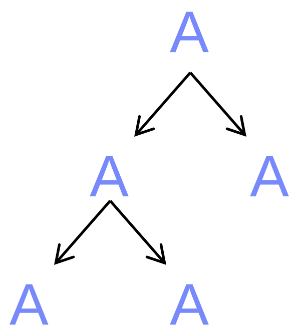
Code can test if a given Place value is dead, get list of alive places, etc.

Resilient X10 Simple Example

Revision of earlier example for failure-reporting X10:

```
class MyClass {
    public static def main(args:Rail[String]):void {
        val c = GlobalRef[Cell[Int]](new Cell[Int](0));
        finish {
            for (p in Place.places()) {
                async {
                    try {
                        at (p) {
                            val v = ...; // non-trivial work
                            at (Place.FIRST_PLACE) {
                                val cell = c();
                                atomic { cell(cell() + v); }  // cell() += v

                            }

                        }

                    } catch (e:DeadPlaceException) {
                        Console.OUT.println(e.place+" died.");
                    }
        }    }    }

        // Runs after remote activities terminate
        Console.OUT.println("Cumulative value: "+c()());
    }
}
```

Special treatment of place 0

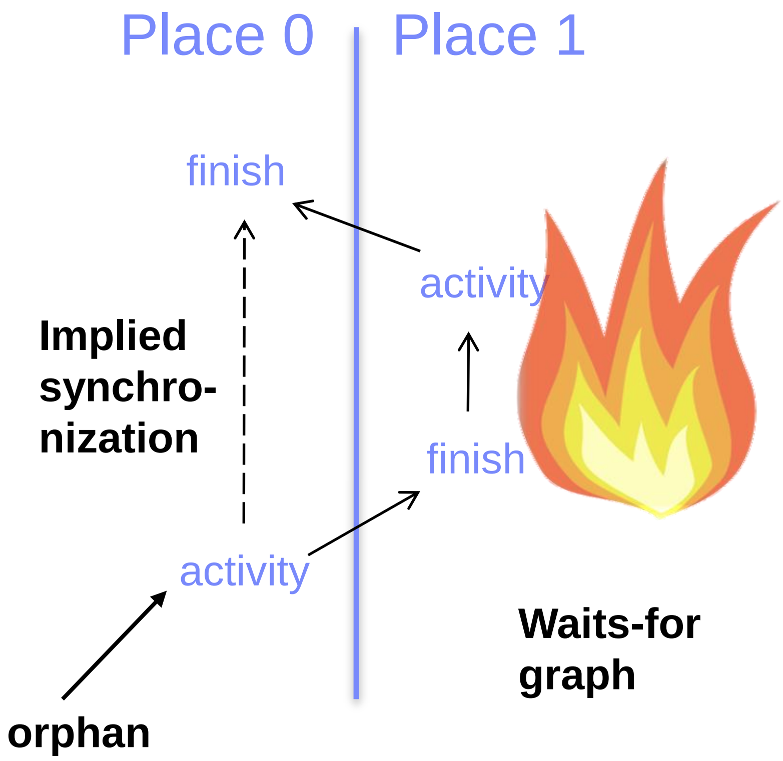


- **Activities are rooted at the 'main' activity at place zero.**
- **If place zero dies, everything dies.**
- **The programmer can assume place 0 is immortal.**
- MTBF of n-node system = MTBF of 1-node system
- Having an immortal place 0 is good for programmer productivity
  - Can orchestrate at place 0 (e.g. deal work)
  - Can do (trivial) reductions at place 0
  - Divide & conquer expressed naturally
  - Can do final result processing / user interface
- However…
  - Must ensure use of place 0 does not become a bottleneck, at scale
- Future work:
  - Transparent fault tolerance for place 0 only (checkpoint the heap).

Happens Before Invariance (HBI) Principle

Failure of a place should not alter the happens before relationship.

```
val gr = GlobalRef(new Cell[Int](0));
try {
    finish at (Place(1)) async {
        finish at (Place(0)) async {
            gr()(10); // A
        }
    }
} catch (e:MultipleExceptions) { }
gr()(3); // B
assert gr()() != 10;
```



A happens before B, **even if place 1 dies**.

Without this property, avoiding race conditions would be very hard.

But guaranteeing it is non-trivial, requires more runtime machinery.

HBI – Subtleties

Relationship between at / finish and orphans

Orphaned activities are *adopted* by the next enclosing *synchronization point*.

```
at (Place(1)) { finish async S } Q    // S happens before Q
finish { at (Place(1)) { async finish async S } Q  }  // S concurrent with Q
```

Exceptions

Adoption does not propagate exceptions:

```
at (Place(1)) {
    try {
        finish at (Place(0)) async { throw e; }
    } catch (e:Exception) { }
}
// e should never appear here
```

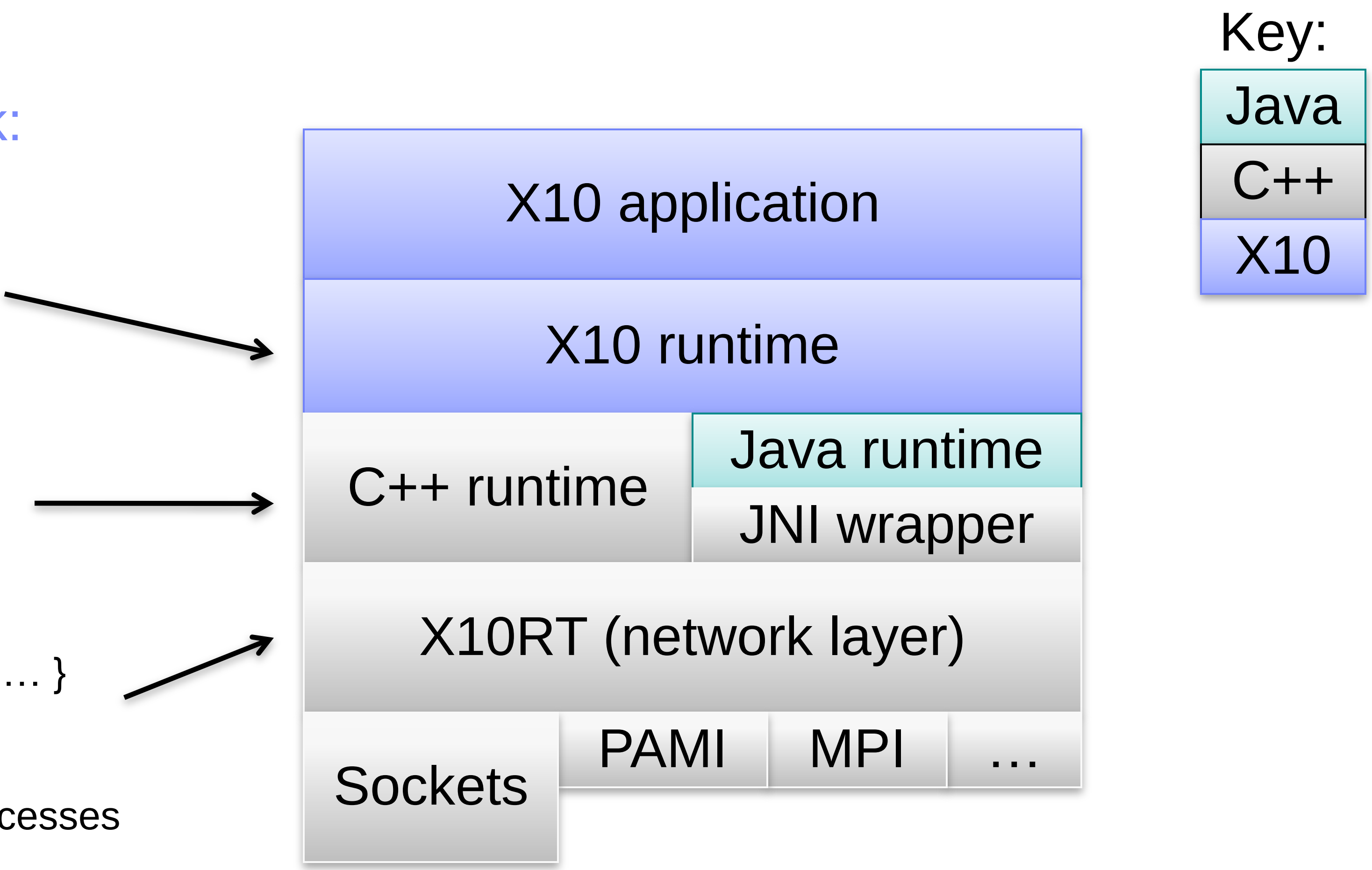
Implementation: X10 Architectural Overview

Runtime stack:

- async { … }
- finish { … }
- at (p) { … }

- OS threads
- Serialization

- at (p) async { … }
- here
- launching processes



Implementing Resilient X10 (X10RT)

Focus on sockets backend

- We have complete control
- Handle TCP timeouts / connection resets gracefully
- Communicate failures up the stack
- Assume no failure during start-up phase (this is short compared to a 24 hour execution)

Changes to X10RT API:

Simple c++ code to send an asynchronous message and wait for a reply (via X10RT API):

```
x10rt_send_msg(p, msgid, buf);
while (!got_reply) {
    x10rt_probe();
}
```

becomes

```
int num_dead = x10rt_ndead();
x10rt_send_msg(p, msgid, buf);
while (!got_reply) {
    int now_dead = x10rt_ndead();
    if (now_dead != num_dead) {
        num_dead = now_dead;
        // account for failure
        break;
    }
    x10rt_probe();
}
```

Implementing Resilient X10 (Finish)

The implementation reduces 'at' to a special case of 'finish'.

Abstractly, finish is a state machine, see paper for details.

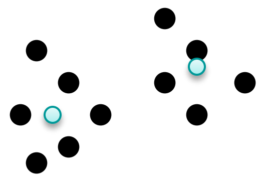
The finish state itself must be resilient, to allow adoption of orphaned activities.

We tried 3 approaches for implementing resilient finish, tested up to 416 places:

- **Store all finish state at place zero.**
  - Simple, makes use of 'immortal' place zero.
  - For finishes logically at place zero in the code, this is optimal anyway.
  - For finishes logically at other places, more communication required.
  - Bottle neck at place zero.
- **Store all finish state in ZooKeeper**
  - Too much overhead.
- **Distributed resilient finish.**
  - Finish state is replicated at one other node.
  - Execution aborted if both nodes die.
  - After optimization for immortal place zero, best all round performance
  - No bottle neck at place zero

See paper for performance results

Application – K-Means (Lloyd’s algorithm)



^ N=11, k=2

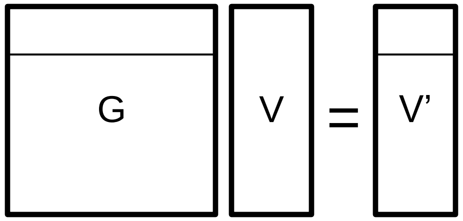
Machine learning / analytics kernel.
Given N (a large number) of points in 4d space (dimensionality arbitrary)
Find the k clusters in 4d space that approximate points’ distribution

•Each cluster’s position is iteratively refined by averaging the position of the set of points for whom that cluster is the closest.
•Very dense computational kernel (assuming large N).
•Embarrassingly parallel, easy to distribute.
•Points data can be larger than single node RAM.
•Points can be split across nodes, partial averages computed at each node and aggregated at place 0.
•Refined clusters then broadcast to all places for next iteration.

Resiliency is achieved via **decimation**
•The algorithm will still converge to an approximate result if only *most* of the points are used.
•If a place dies, we simply proceed without its data and resources.
•Error bounds on this technique explored in Rinard06

Performance is within 90% of non-resilient X10

Application – Iterative Sparse Matrix * Dense Vector



Kernel found in a number of algorithms, e.g. GNMF, Page Rank, …
An N*N sparse (0.1%) matrix, G, multiplied by a 1xN dense vector V
Resulting vector used as V in the next iteration.
Matrix block size is 1000x1000, matrix is double precision

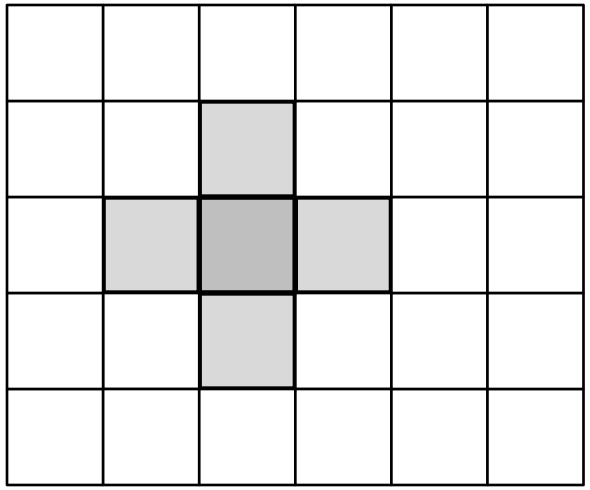
G distributed into row blocks. Every place starts with entire V, computes fragment of V’.
Every place communicates fragments of V to place 0 to be aggregated.
New V broadcast from place 0 for next iteration (G is never modified).

Code is memory-bound, amount of actual computation quite low
Problem is the size of the data – does not fit in node.
G is loaded at application start, kept in RAM between iterations.

Resiliency is achieved by replaying lost work:
•Place death triggers other places to take over lost work assignment.
•Places load the extra G blocks they need from disk upon failure

100x faster than Hadoop
Resilient X10 ~ same speed as existing X10

Application – Heat Transfer



Demonstration of a 2D stencil algorithm with simple kernel
An N*N grid of doubles
Stencil function is a simple average of 4 nearest neighbors

Each iteration updates the entire grid.
Dense computational benchmark
Distributed by spatial partitioning of the grid.
Communication of partition outline areas required, each iteration.

Resiliency implemented via *checkpointing*.
Failure triggers a reassignment of work, and global replay from previous checkpoint.
Checkpoints stored in an in-memory *resilient store*, implemented in X10

Performance can be controlled by checkpoint frequency.
If no checkpoints, performance is the same as existing X10

Conclusions

Resilient X10: A novel point in the design space

- Avoid sacrificing performance
- Re-use exception semantics
- HBI principle ensures that transitive synchronization is preserved after node failure
- Ensure no surprises for the programmer

Implemented and tested at scale

- Implemented ‘finish’ 3 ways, microbenchmarked
- Implemented 3 apps that handle failure in different ways
  - K-Means (decimation)
  - Sparse Matrix * Dense Vector (reload & replay)
  - Stencil (checkpointing)
- Apps are extended from non-resilient versions to handle DeadPlaceException
- Performance close to existing X10, but resilient to a few node failures

Questions?

Resilient Store

•Provide reliable memory so that data can continue to be accessed in the event of a failure
•Disk storage has too much overhead (e.g.Hadoop).

Our solution:

Use Failure-Reporting X10 constructs to replicate data across different places
Manifests as utility library to Failure-Reporting X10 programs
Upon place death, transparently recover data from a backup copy

Different replication methods provide with varying levels of consistency:

Asynchronous updates – weak consistency
Update whenever data changes
Batch updates:

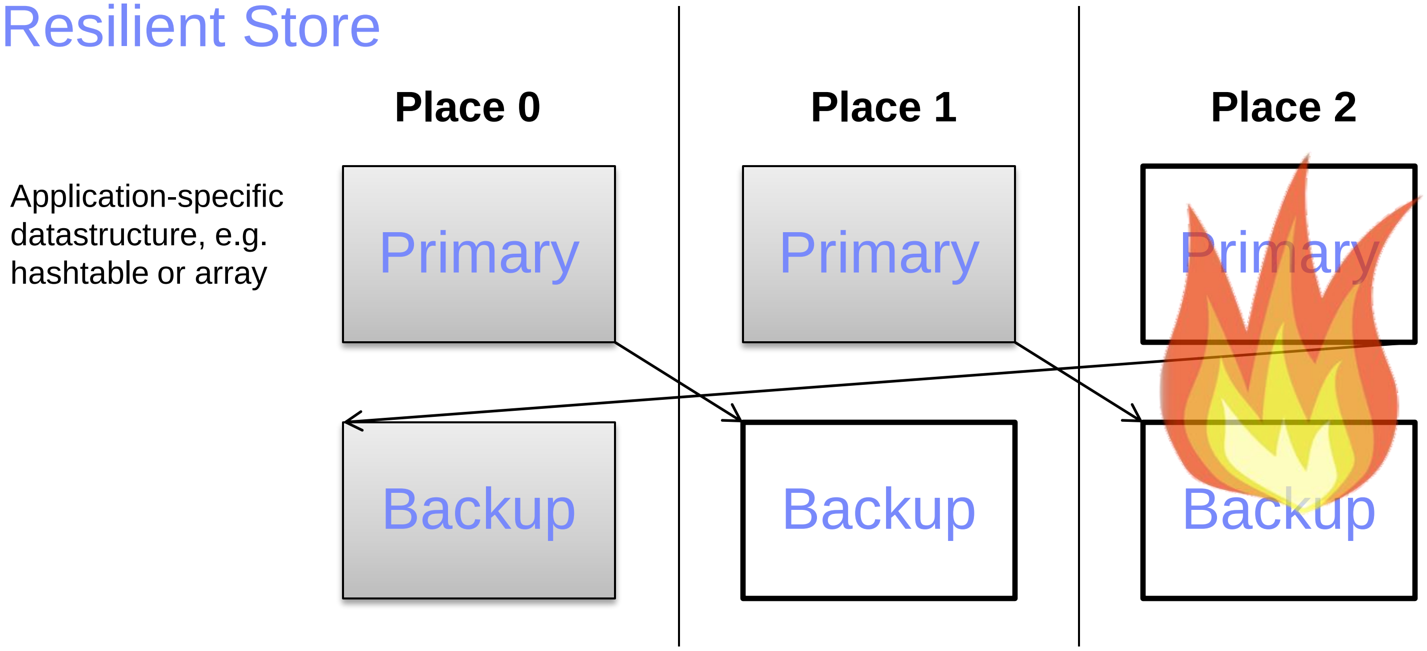
Whenever number of updates exceeds a threshold
Whenever time since last update exceeds a threshold

Synchronous updates – strong consistency, guaranteed never to lose data

Lock all replicas, perform update atomically across all replicas

Synchronous updates result in more consistency but higher overhead.
(These two options correspond to whether or not a ‘finish’ is used in the implementation…)

Resilient Store



Upon failure, must

•Identify lost primary stores, search to find their backups, recover the data
•Invoke application-specific code to distribute this data over remaining stores
•Identify lost backups, recreate from primary at a different place

Status:

Current implementation sufficient for ResilientHeatTransfer
We are now generalizing for other algorithms

Fan-out and task management

- Many applications seem to have a common pattern:
- Start at place zero:
- for (iterations) {
  - Divide work, create an activity at each place, maybe share some input data
  - Each place does its work, and sends result back to place 0
  - Back at place zero, if any place failed, repeat iteration with different work assignment
  - Otherwise aggregate result and use as input for next iteration
- }
- The pattern is an iterative fan-out / reduction with a dynamic work-assignment that reassigns work upon failure.
- We can build a framework to ease writing such frameworks, and ensure it is optimised to e.g. use a tree-based fan out to distribute the communication cost
- Application simply provides the task-breakdown and the reduction operation.
- Kmeans and MatVecMult are a perfect fit for such a framework.
- ResilientHeatTransfer can also benefit from it.
- **Current status: most efficient implementation is in MatVecMult, need to factor it out**

Failure Reporting X10 – Higher level programming

- DistArray: X10 library supporting implicitly distributed (global) arrays
  - Specify a PlaceGroup on creation
  - Implicitly handles the blocking of the array across the places in the group
  - Allows programs to be more concise, if used

- ResilientDistArray: A DistArray that can, on place death:
  - Recover the lost data (explicit checkpointing)
  - Redistribute all the data using a new PlaceGroup
  - (one without dead places in it)

- We hope to be able to deploy this in our example code.

```
val livePlaces = new ArrayList[Place]();
for (pl in Place.places()) livePlaces.add(pl);
var pg:PlaceGroup = new SparsePlaceGroup(livePlaces.toRail());
val A = new ResilientDistArray_BlockBlock_2[Double](n, n, pg, ...);
A.snapshot(); // create a snapshot

for (i in 1..1000) {
    try {
        finish for (pl in A.placeGroup()) at (pl) async {
            for (p in A.localIndices()) A(p) = ... // do distributed calculation
        }
        if (i % 10 == 0) A.snapshot(); // snapshot every 10th iteration
    } catch (e:DeadPlaceException) {
        livePlaces.remove(e.place); // remove the dead place and
        pg = new SparsePlaceGroup(livePlaces.toRail());
        A.restore(pg); // restore from the latest snapshot
    }
} /* for (i) */
```

Failure-reporting X10 (Finish implementation)

Each finish { … } has a resilient runtime state object, containing a set of counters

- live[p] – number of activities executing (or queued) at p
- transit[a,b] – number of activities sent by place a, not yet received by place b (sparse)

Quiescence of finish defined to be when counters are zero, ignoring dead places

Non-zero counters at dead places become DeadPlaceExceptions

User-level exceptions must be handled too

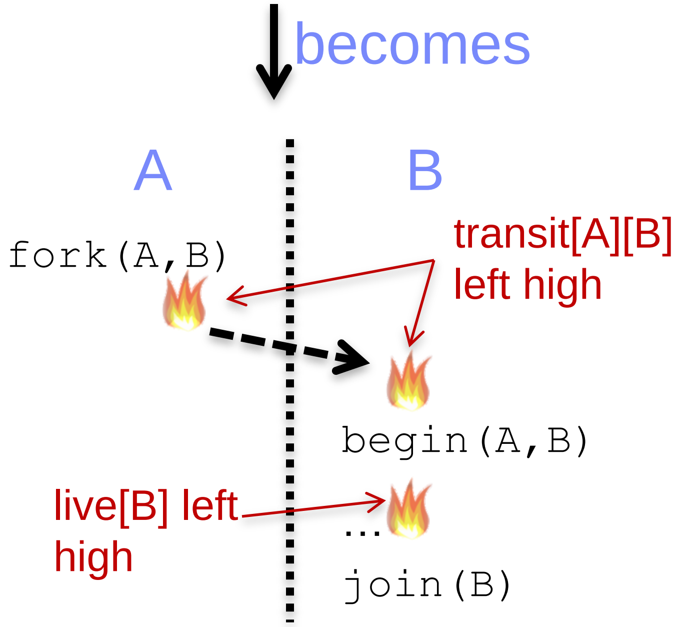
Adoption of activities under dead finishes must be handled

No other real language has a termination algorithm this general…

Operations:

```
fork(a, b)        transit[a][b]++
begin(a, b)       transit[a][b]-- ; live[b]++
join(b)           live[b]--
```

```
// starting from A
at (B) async {

     ...

}
```



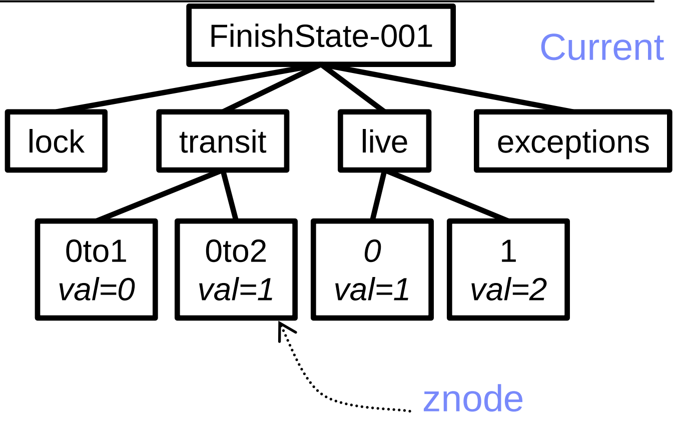
<u>Current implementation (fully implemented):</u>
Place 0 stores all counters, never fails
All fork/begin/join notifications go to place 0 (a bottleneck)
Scales OK on hundreds of nodes
Impractical for thousands of nodes (where resiliency really matters)

Failure-reporting X10 (XRX implementation)

Investigation of ZooKeeper as an external finish database.

- **Initial implementation done** for Managed X10 (X10 on Java)
  - Each counter element is mapped to a znode
  - Exceptions are stored similarly
  - ZooKeeper "Watcher" mechanism notifies waiting activity to test quiescence only when counters are modified (avoids polling)
- In current version, **performance is not good**
  - About 50 times slower than the "Place0" version
  - Many ZK calls are necessary to implement fork/begin/join
  - Each ZK call is not lightweight (each op takes 0.1~1.0 ms)

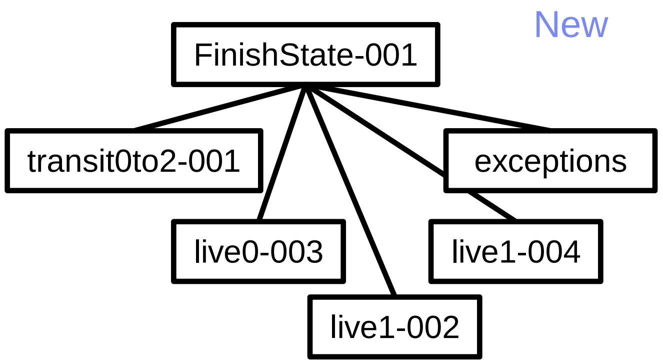
→**Redesigning the znode mapping** to reduce the number of ZK ops



```
val p = here.next();
for (i in 1..100) {
  finish {
    for (j in 1..100) {
      at (p) async {};
    }
  }
}
```

Current mapping

Each element of FinishState arrays is expressed as a znode, which contains a value.
Therefore, `lock;getData;setData;unlock` sequence is necessary to update the value in each node

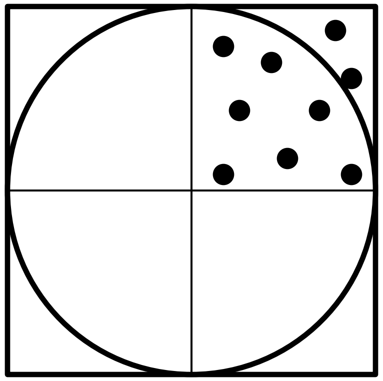


Modify the mapping to utilize "sequential creation" mode:
Express the array value by the *number* of nodes

Reduce number of ZK ops needed per finish op
No need for lock operations

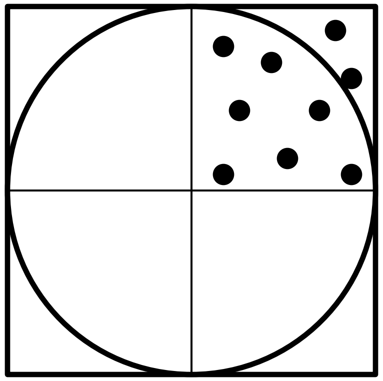
Complete example – Monte Pi

```
public class MontePi {

    static val ITERS = 1000000000 / Place.MAX_PLACES;

    public static def main (args : Rail[String]) {

        val result = GlobalRef(new Cell[Long](0)); // points_in_circle

        finish for (p in Place.places()) async {
            at (p) {
                val rand = new Random(System.nanoTime());
                var total : Long = 0;
                for (iter in 1..ITERS) {
                    val x = rand.nextDouble();
                    val y = rand.nextDouble();
                    if (x*x + y*y <= 1.0) total++;
                }
                val total_ = total;
                Console.OUT.println("Work done at: "+here);
                at (result.home) atomic {
                    result()(result()()+total_));
                }
            }
        }

        val pi = (4.0 * result()()) / (ITERS*Place.MAX_PLACES);
        Console.OUT.println("pi = "+pi);

    }

}
```

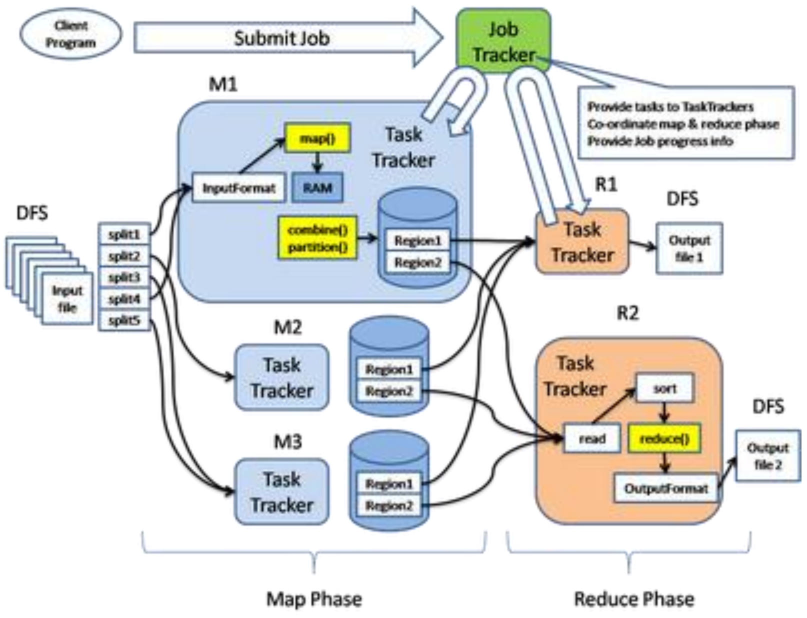


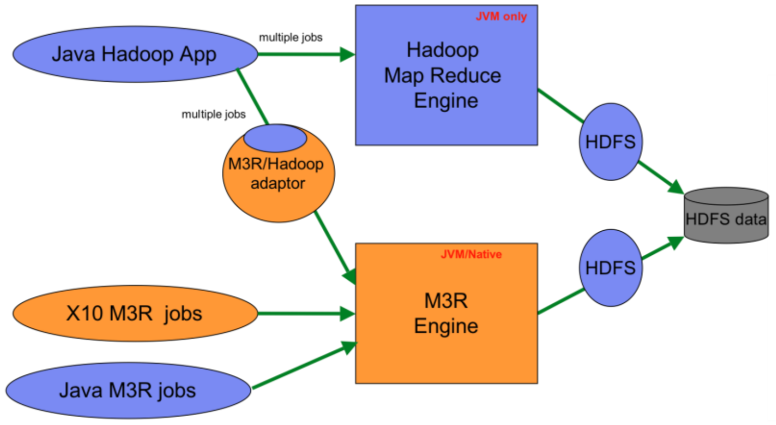
Complete example – Resilient Monte Pi

```
public class ResilientMontePi {

    static val ITERS = 1000000000l / Place.MAX_PLACES;

    public static def main (args : Rail[String]) {

        val result = GlobalRef(new Cell(Pair[Long,Long](0, 0))); // (points_in_circle, points_tested)

        finish for (p in Place.places()) async {
            try {
                at (p) {
                    val rand = new Random(System.nanoTime());
                    var total : Long = 0;
                    for (iter in 1..ITERS) {
                        val x = rand.nextDouble();
                        val y = rand.nextDouble();
                        if (x*x + y*y <= 1.0) total++;
                    }
                    val total_ = total;
                    Console.OUT.println("Work done at: "+here);
                    at (result.home) atomic {
                        result()(Pair(result()().first+total_, result()().second+ITERS));
                    }
                }
            } catch (e:DeadPlaceException) {
                Console.OUT.println("Got DeadPlaceException from "+e.place);
            }
        }

        val pi = (4.0 * result()().first) / result()().second;
        Console.OUT.println("pi = "+pi+"   calculated with "+result()().second+" samples.");

    }

}
```



Case study: M3R (Main Memory Map Reduce)





M3R (today)
- Non-resilient MR on top of X10
- Much smaller implementation
- Hadoop compatibility layer
- Much faster

Hadoop
- Resilient implementation of map reduce
- Ground-up implementation
- Considerable amount of code
- Performance / expressiveness limitations
- No code sharing with other frameworks

M3R (tomorrow)
- Resilient MR on top of Resilient X10
- Still a small implementation
- Still fast
- X10 does a lot of the work

Will be able to implement a range of resilient distributed programming models using X10

Game Plan (2013 - 2014)

1. Failure-reporting X10
   - X10 is internally resilient
   - Applications must handle failures (communicated via exceptions)

2. Resilient frameworks
   - Applications expressed by creating closures and passing to libraries
   - Libraries re-execute code if necessary
   - Libraries handle locality

3. Language support for resilience
   - Detect that code is suitable for execution in a resilient framework
   - Place-independent (can be moved)
   - Idempotent (can run it again)

4. Elastic X10
   - Execution can expand after program starts
   - Unclear that this brings much benefit except for very long executions
   - Will consider possible value of this at a later date…