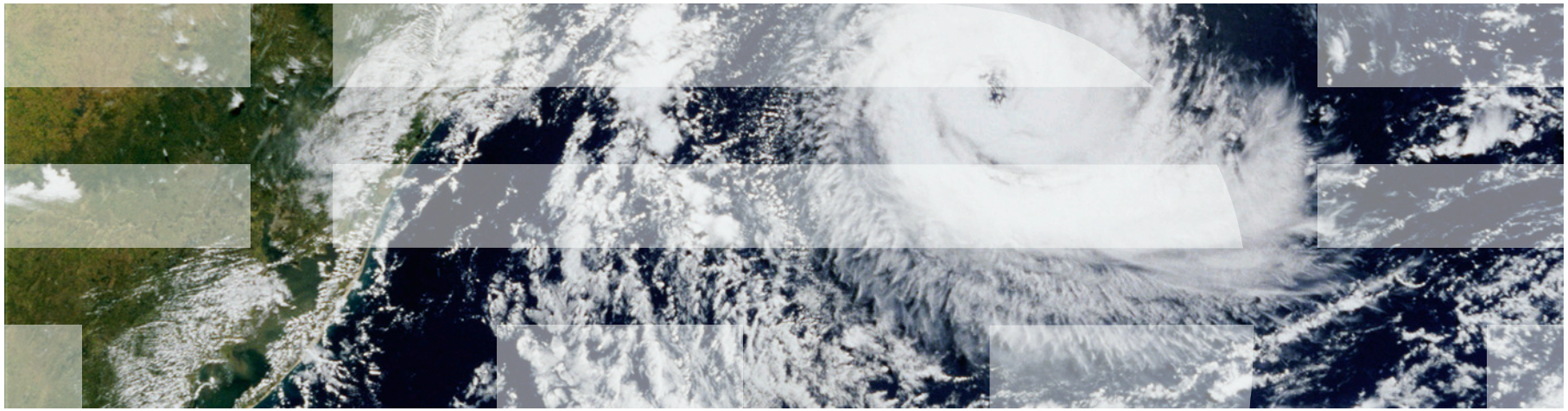
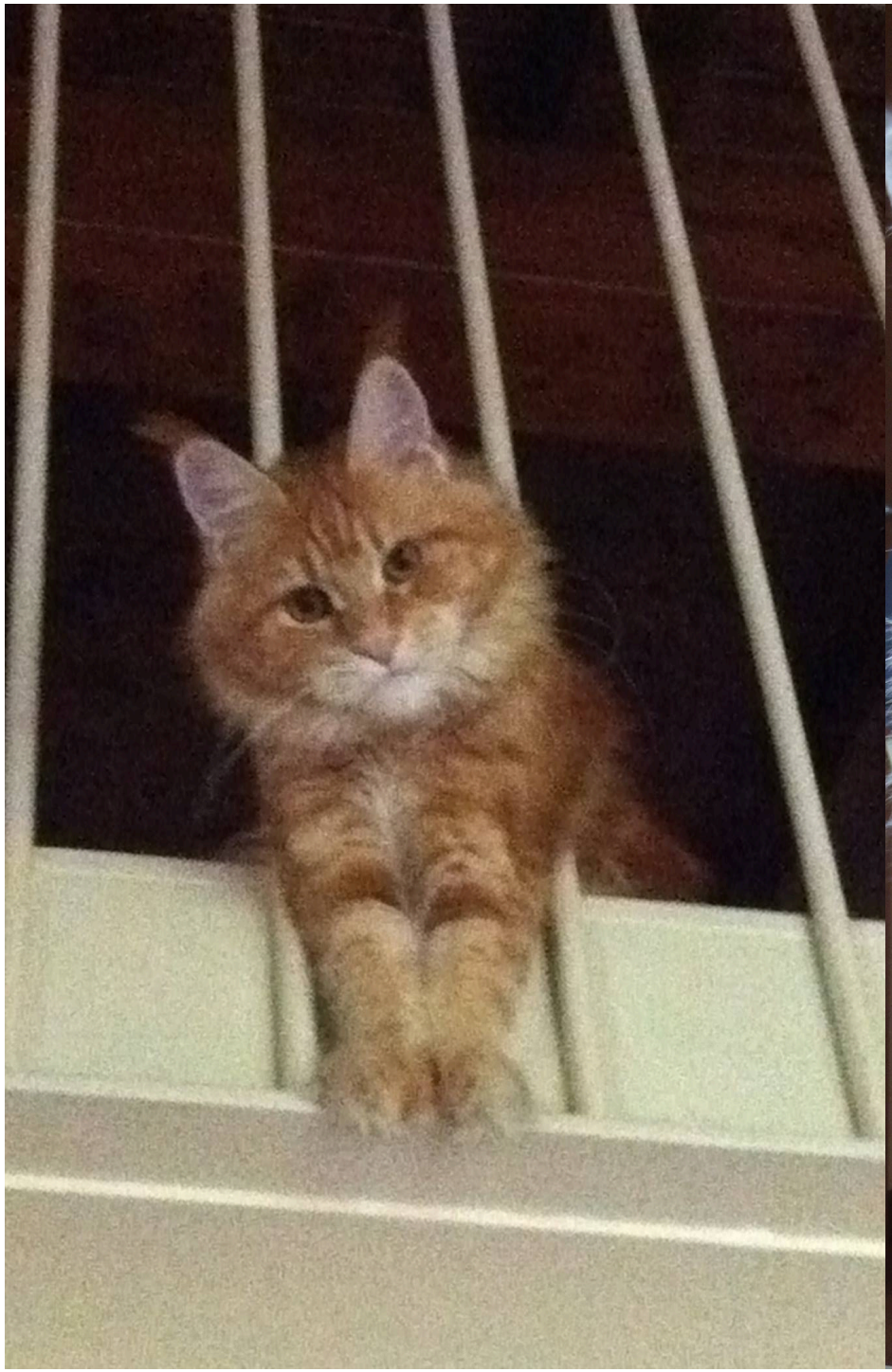
Dave Cunningham (now Google), Dave Grove, Ben Herta, Arun Iyengar , Kiyokuni Kawachiya, Hiroki Murata, Vijay Saraswat , Mikio Takeuchi, Olivier Tardieu, Joshua Milthorpe (ANU, IBM) Sara Salem Houda (ANU), Silvia Crafa (U Padua)

Resilient X10
Efficient failure-aware programming

PPoPP 2014 “Resilient X10: Efficient failure-Aware Programming”
ECOOP 2014 “Semantics of (Resilient) X10”
http://x10-lang.org

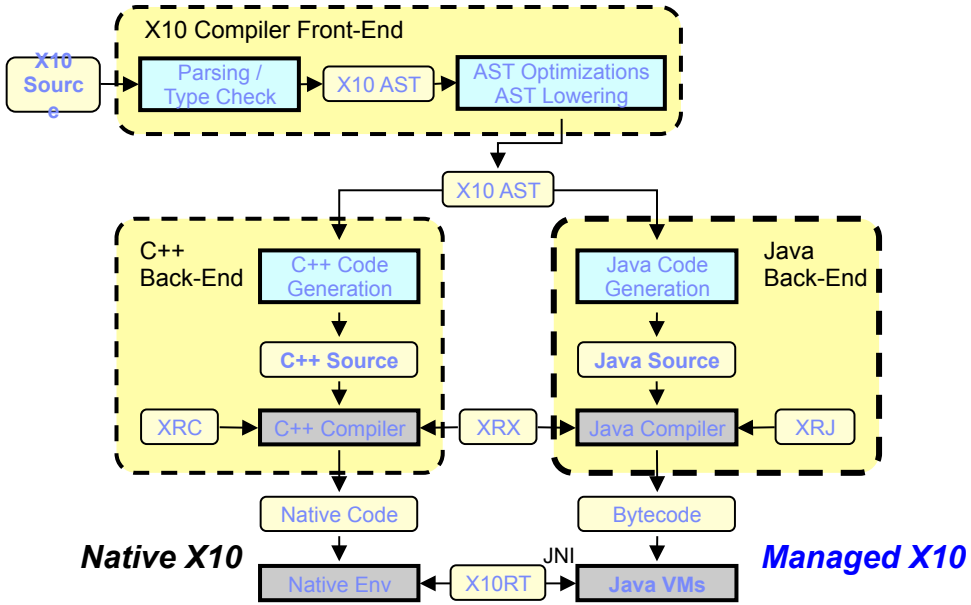


Funded in part by AFOSR



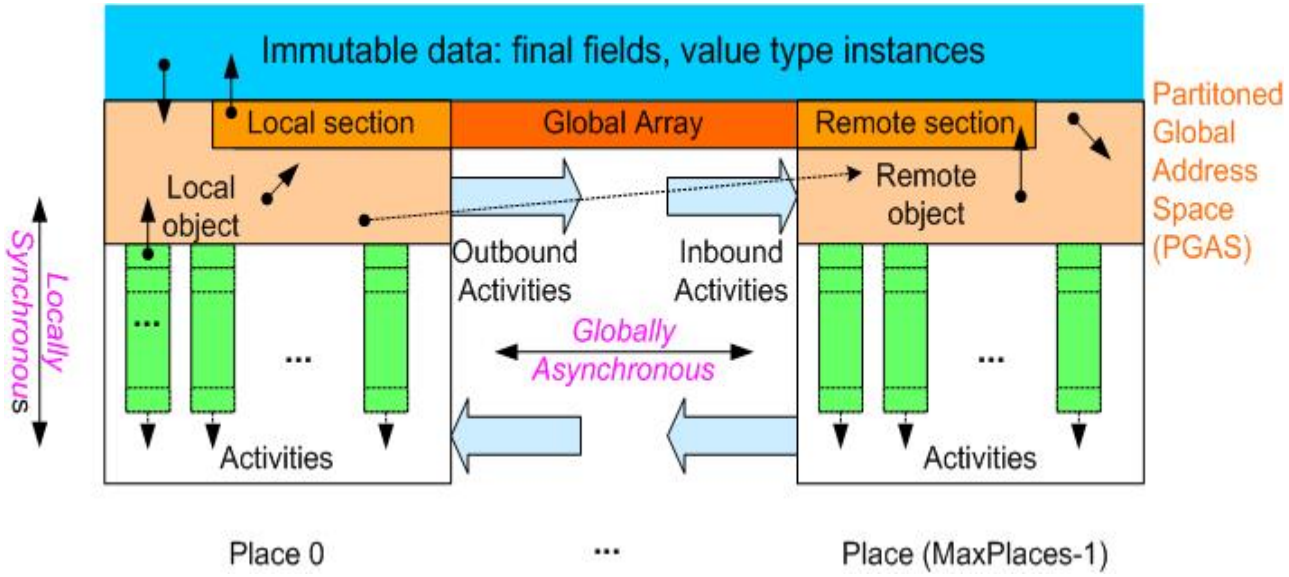
X10: Place-centric, Asynchronous Computing

- Language for at-scale computing being developed at IBM Research (since 2004). Funded by DARPA, IBM.
- Focused on High Performance Computing, and (since 2010) scale-out analytics
- Compiles to C++
- Compiles to Java, interoperates
- Linux, AIX, MacOS, Cygwin, Blue Gene / x86, x86_64, PowerPC, GPUs, modern interconnects
- Eclipse based IDE, with remote execution support



Java-like productivity, MPI-like performance at scale

X10 and the APGAS model



Asychrony • **async S**	**Atomicity** • **when (c) S**	**Global data-structures** **points, regions, distributions, arrays**
Locality • **at (P) S**	**Order** • **finish S** • **clocks**	

- Five basic constructs
  - async S – run S as a separate activity
  - at (P) S – switch to place P to run S.
  - finish S – execute S, wait for termination
  - when (c) S – execute S when c, atomically
  - clocked async, clocked finish support barriers

APGAS Idioms

Remote procedure call

```
v = at(p) evalThere(arg1, arg2);
```

Active message

```
at(p) async runThere(arg1, arg2);
```

Divide-and-conquer parallelism

```
def fib(n:Long):Long {
  if(n < 2) return n;
  val f1:Long;
  val f2:Long;
  finish {
    async f1 = fib(n-1);
    f2 = fib(n-2);
  }
  return f1 + f2;
}
```

SPMD

```
finish for(p in Place.places()) {
  at(p) async runEverywhere();
}
```

Atomic remote update

```
at(ref) async atomic ref() += v;
```

Computation/communication overlap

```
val acc = new Accumulator();
while(cond) {
  finish {
    val v = acc.currentValue();
    at(ref) async ref() = v;
    acc.updateValue();
  }
}
```

Some additional constructs -- Clocks

APGAS barriers

- synchronize dynamic sets of tasks

`x10.lang.Clock`

- anonymous or named
- task instantiating the clock is registered with the clock
- spawned tasks can be registered with a clock at creation time
- tasks can deregister from the clock
- tasks can use multiple clocks
- split-phase clocks
  - clock.resume(), clock.advance()
- compatible with distribution

```
// anonymous clock
clocked finish {
  for(1..4) clocked async {
    Console.OUT.println("Phase 1");
    Clock.advanceAll();
    Console.OUT.println("Phase 2");
  }
}
// named clock
finish {
  val c = Clock.make();
  for(1..4) async clocked(c) {
    Console.OUT.println("Phase 3");
    c.advance();
    Console.OUT.println("Phase 4");
  }
  c.drop();
}
```

Some additional constructs -- Collecting Finish

```
public class CollectPi {
  public static def main(args:Rail[String]) {
    val N = Long.parse(args(0)), P = Long.parse(args(1));
    val result = finish(Reducible.SumReducer[Double]()) {
      for(1..P) async {
        val myRand = new Random();
        var myResult:Double = 0;
        for (1..(N/P)) {
          val x = myRand.nextDouble();
          val y = myRand.nextDouble();
          if (x*x + y*y <= 1) myResult++;
        }
        offer myResult;
      }};
    val pi = 4*result/N;
    Console.OUT.println("The value of pi is " + pi);
  }}
```

Key Semantic Properties

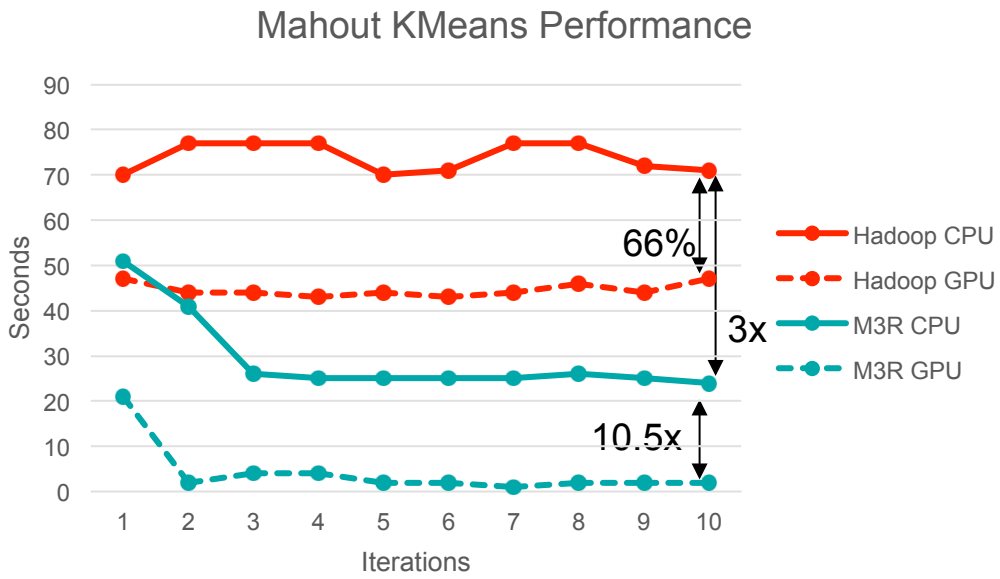
- Programs with multiple places, (collecting) finish, async, atomic, at, clocks are deadlock-free. (Concur '05)
  - –Only when is not permitted
- Determinacy for "clocked final" programs.
- (And C10 …)
- Formal operational semantics, plus equational theory (ECOOP '14)
- "Hard hat" object initialization – no escape of "this" during construction (ECOOP '12)
- Precise syntactic characterization of Happens Before relationship for polyhedral programs (PPoPP '13)
  - – Allows precise statement-specific, instance-specific analysis of races
- And the clean design that permits Resilience to be "just added in"…

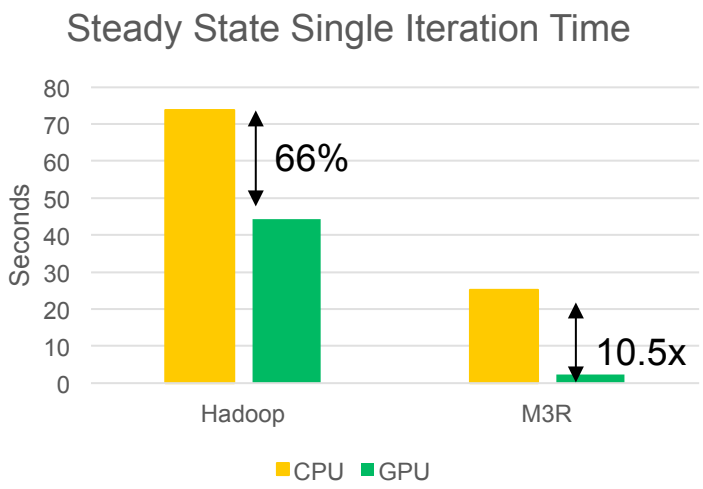
Key Applications

- Global Matrix Library
  - Distributed, partitioned implementation of operations on sparse/dense matrices / vectors
- Global Load Balancing Library (PPoPP '11, PPAA '14)
- Scalegraph – X10 graph library (scalegraph.org)

- M3R -- Main Memory Map Reduce (VLDB'12)
  - Open source M3RLite (~200 lines)
  - Open source BSP engine (~300 lines)
- X10 in IBM products
  - M3R
  - X10 for in-memory scale-out analytics
  - "…introducing rich time modeling, reasoning and analytics to detect and respond to intricate patterns and trends; innovative global analytics to extract valuable insights over populations of business entities in real-time;…"

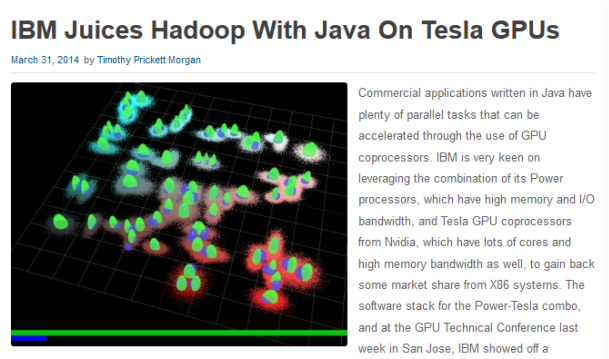
M3R Performance: Mahout KMeans on Power8+GPU





With M3R, the real computation becomes the dominant cost enabling GPU acceleration of KMeans mapper to yield 10.5x speedup

Execution Environment		K-Means Parameters	
Host:	Power8 (8286-42A)	**N:**	9.6M
OS:	Ubuntu 14.04	**D:**	2
GPU:	K40m	**K:**	64
Hadoop:	1.0.3	**Mappers:**	12
HDFS:	in RAM	**Reducers:**	1
M3R:	2.5.0.1		



IBM Juices Hadoop With Java On Tesla GPUs

March 31, 2014 by Timothy Prickett Morgan

Commercial applications written in Java have plenty of parallel tasks that can be accelerated through the use of GPU coprocessors. IBM is very keen on leveraging the combination of its Power processors, which have high memory and I/O bandwidth, and Tesla GPU coprocessors from Nvidia, which have lots of cores and high memory bandwidth as well, to gain back some market share from X86 systems. The software stack for the Power-Tesla combo, and at the GPU Technical Conference last week in San Jose, IBM showed off a

GPU implementation & experimental evaluation by Keith Campbell, HAL Lab
Demo at NVIDIA GPU Tech. Conf. March, 2014
Article: http://www.enterprisetech.com/2014/03/31/ibm-juices-hadoop-java-tesla-gpus/

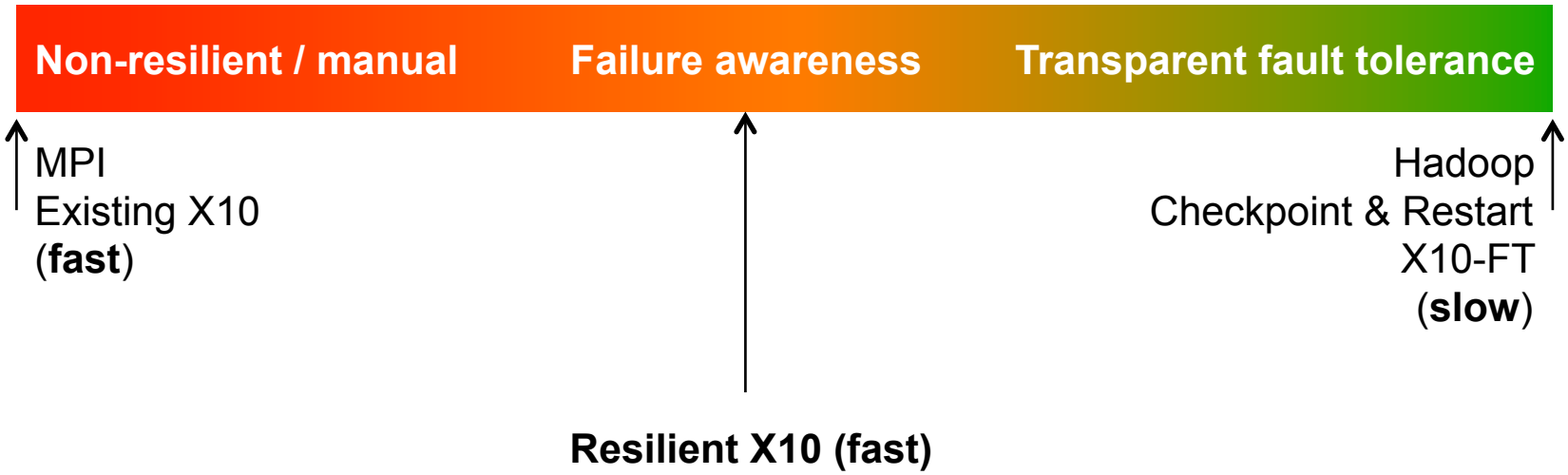
Resiliency Spectrum

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

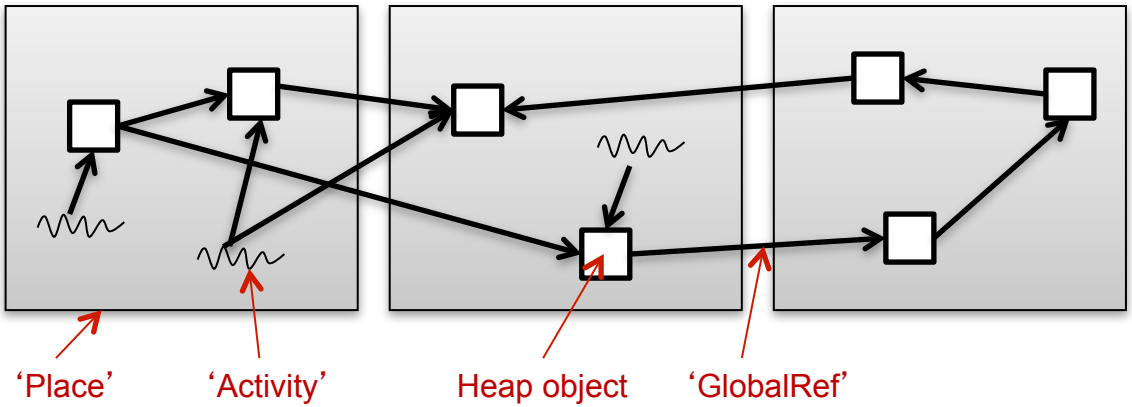
- If the computation is approximate: *trade accuracy for reliability (e.g. Rinard, ICS06)*
- If the computation is repeatable: *replay it*
- If lost data is unmodified: *reload it*
- If data is mutated: *checkpoint it*
- Libraries can hide, abstract, or expose faults (e.g. containment domains)
- Can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

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

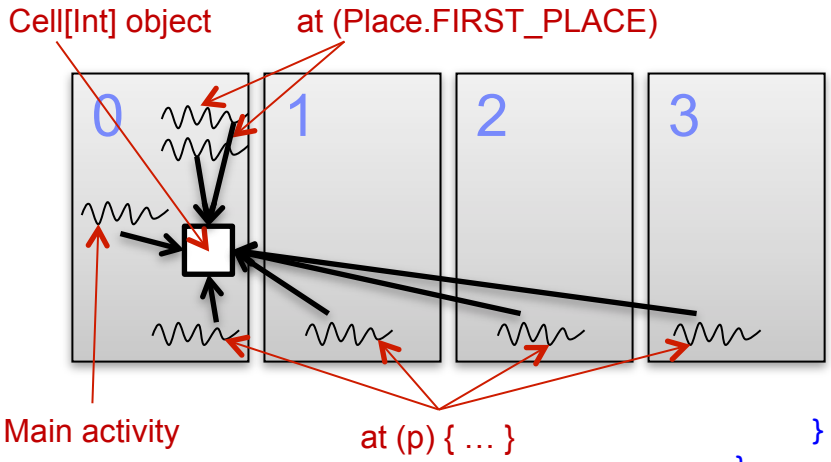
- Use exceptions to report failure
- Existing exception semantics give strong synchronization guarantees

X10 Language Overview (Distributed Features)

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



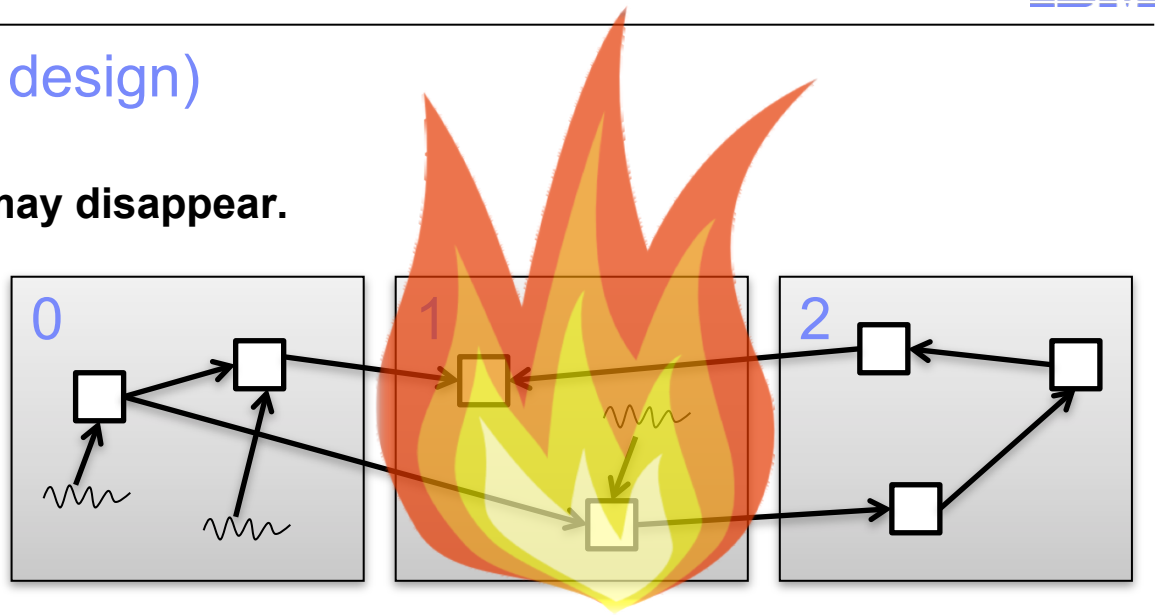

```
val x = ...;
val y = ...;
at (p) {
    val tmp = x + y;
}
```




```
class MyClass {
    public static def main(args:Rail[String]):void {
        val c = GlobalRef(new Cell[Long](0));
        finish {
            for (p in Place.places()) {
                async {
                    at (p) {
                        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

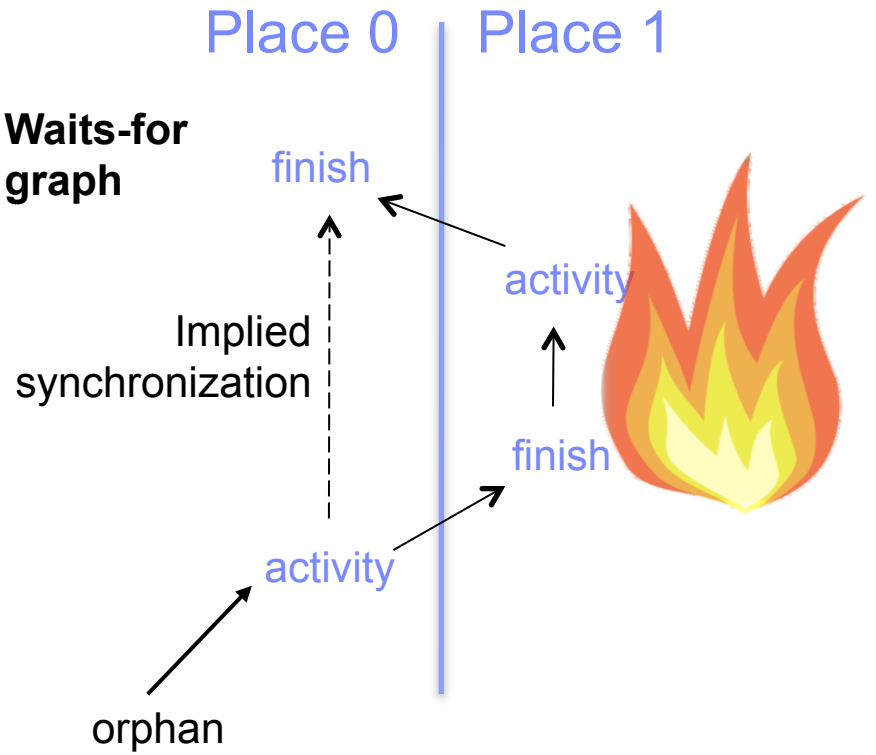
Revision of earlier example for failure-reporting X10:

```
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()());
    }
}
```

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
```

TX10

Semantics of (Resilient) X10 [ECOOP 2014]

S.Crafa, D.Cunningham, V.Saraswat, A.Shinnar, O.Tardieu

object id

global object id

exception

$$\textit{Values} \quad v ::= o \mid o\$p \mid \mathsf{E} \mid \mathsf{DPE}$$

$$\textit{Expressions} \quad e ::= v \mid x \mid e.f \mid \{f{:}e, \ldots, f{:}e\} \mid \texttt{globalref}\, e \mid \texttt{valof}\, e$$

$$\textit{Statements} \quad s ::= \texttt{skip;} \mid \texttt{throw}\, v \mid \texttt{val}\, x = e\ s \mid e.f = e; \mid \{s\ \ t\}$$

$$\texttt{at}(p)\texttt{val}\, x = e\, \texttt{in}\, s \mid \texttt{async}\, s \mid \texttt{finish}\, s \mid \texttt{try}\, s\, \texttt{catch}\, t$$

$$\overline{\texttt{at}(p)\, s} \mid \overline{\texttt{async}\, s} \mid \texttt{finish}_\mu\, s$$

$$\textit{Configurations} \quad k ::= \langle s, g\rangle \mid g$$

error propagation and handling

$$\textit{Global heap} \quad g ::= \emptyset \mid g \cdot [p \mapsto h] \qquad \textit{Local heap} \quad h ::= \emptyset \mid h \cdot [o \mapsto (\tilde{f}_i : \tilde{v}_i)]$$

Semantics of (Resilient) X10

- Small-step transition system, mechanised in Coq
- **non in ChemicalAM style** (better fits the centralised view of the distributed program)

$$\langle s, g\rangle \longrightarrow_p \langle s', g'\rangle \mid g' \qquad \langle s, g\rangle \xrightarrow{\mathsf{E}\times}_p \langle s', g'\rangle \mid g' \qquad \langle s, g\rangle \xrightarrow{\mathsf{E}\otimes}_p \langle s', g'\rangle \mid g'$$

Proved in Coq

Async failures arise in parallel threads
and are caught by the inner `finish` waiting for their termination

```
finish {async throw E   async s2}
```

Proved in Coq

Synch failures lead to the failure of any sync continuation
leaving async (remote) running code free to terminate

```
{async at(p)s1   throw E   s2}
```

Semantics of (Resilient) X10

- Small-step transition system, mechanised in Coq
- **non in ChemicalAM style** (better fits the centralised view of the distributed program)

$$\langle s, g\rangle \longrightarrow_p \langle s', g'\rangle \mid g' \qquad \langle s, g\rangle \xrightarrow{\mathsf{E}\times}_p \langle s', g'\rangle \mid g' \qquad \langle s, g\rangle \xrightarrow{\mathsf{E}\otimes}_p \langle s', g'\rangle \mid g'$$

Absence of stuck states

(the proof can be run, yielding an interpreter for TX10)

Semantics of **Resilient** X10

smoothly scales to node failure, with

- global heap is a partial map: ***dom(g)* collects non failed places**
- executing a statement at failed place results in a DPE
- place shift at failed place results in a DPE
- remote exceptions flow back at the remaining finish **masked** as DPE

contextual rules modified accordingly

(Place Failure)

$$\frac{p \in dom(g)}{\langle s, g\rangle \longrightarrow_p \langle s, g \setminus \{(p, g(p)\}\rangle}$$

$$\frac{p \notin dom(g)}{\begin{array}{r}\langle \texttt{skip}, g\rangle \xrightarrow{\text{DPE}\otimes}_p g \\ \langle \texttt{async}\, s, g\rangle \xrightarrow{\text{DPE}\otimes}_p g \\ \langle \texttt{at}(p)\, s, g\rangle \xrightarrow{\text{DPE}\otimes}_q g\end{array}}$$

Semantics of Resilient X10

- **Happens Before Invariance**
  - failure of place q does not alter the happens before relationship between statement instances at places other than q

$\overline{\texttt{at}}(0)\,\{\,\overline{\texttt{at}}(p)\ \texttt{finish}\ \overline{\texttt{at}}(q)\ \overline{\texttt{async}}\ s_1 \quad s_2\}$

p fails while s1 is running at q

s2 runs at 0 after s1

same behaviour!

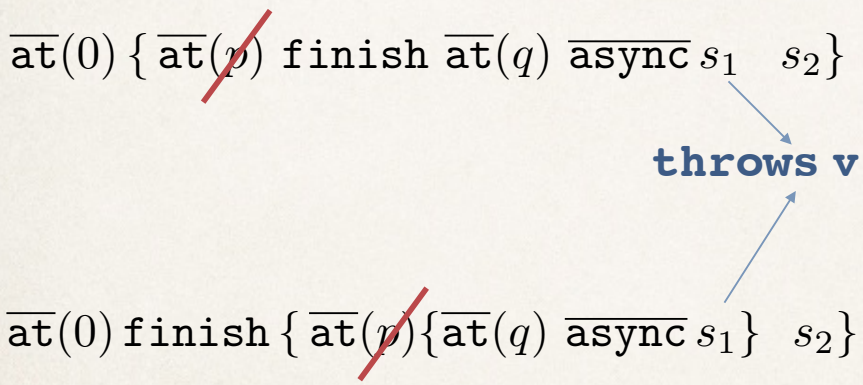
$\overline{\texttt{at}}(0)\ \texttt{finish}\,\{\,\overline{\texttt{at}}(p)\{\overline{\texttt{at}}(q)\ \overline{\texttt{async}}\ s_1\} \quad s_2\}$

s2 runs at 0 in parallel with s1

Semantics of Resilient X10

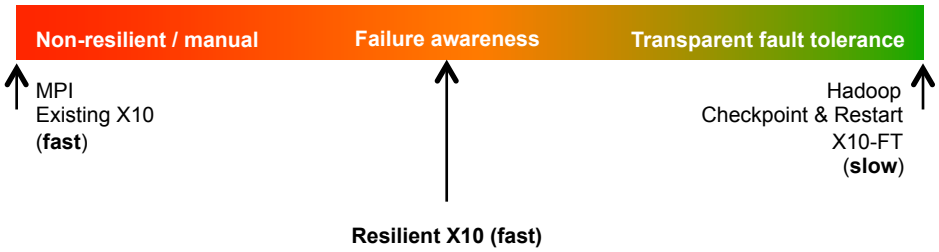
- **Happens Before Invariance**
  - failure of place q does not alter the happens before relationship between statement instances at places other than q

DPE⊗ ***flows at place 0 discarding s1***



$v\times$ ***flows at place 0 while s2 is running***

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

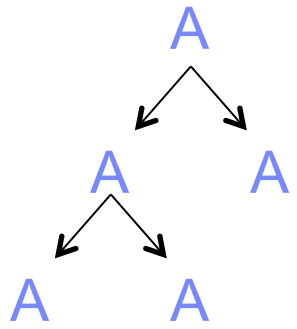
Implemented, tested at scale, released (X10 2.4.1)

- 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?

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



Papers

- PPoPP 2014 “Resilient X10: Efficient failure-Aware Programming”
- ECOOP 2014 “Semantics of (Resilient) X10”

Future Work

- More applications!
- “Elastic” X10
  - Expand into new hardware
  - Allow new hardware to replace failed hardware
- Tolerate failure of place 0
  - Checkpoint the heap at place 0? Slow place 0, use only for orchestration
  - Or, just don’t have a rooted activity model

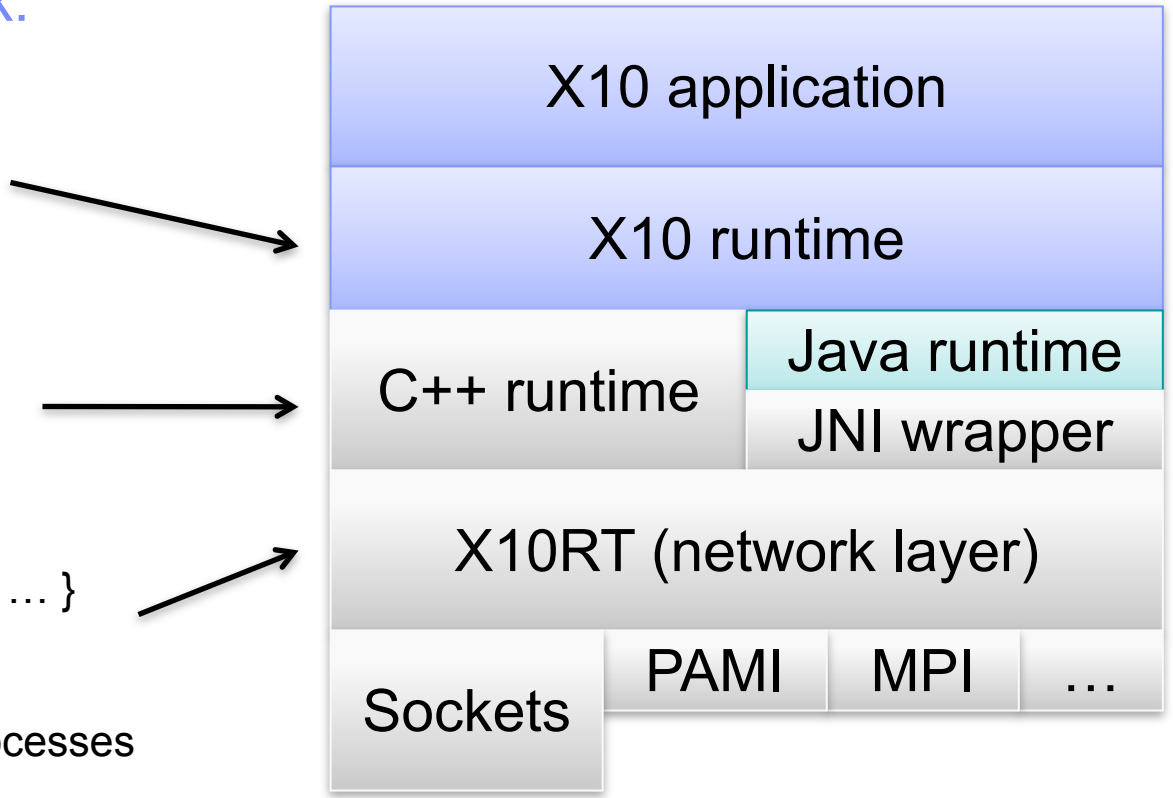
Implementation: X10 Architectural Overview

Runtime stack:

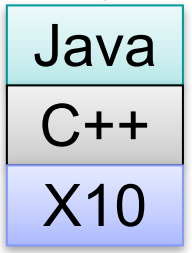
- async { … }
- finish { … }
- at (p) { … }

- OS threads
- Serialization

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



Key:



Implementing Resilient X10 (X10RT)

Focus on sockets backend

- We have complete control
- Handle TCP timeouts / connection resets gracefully
- Communicate failures up the stack
- Abort on timeout during start-up phase

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();
}
```


```
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 Counters Abstraction)

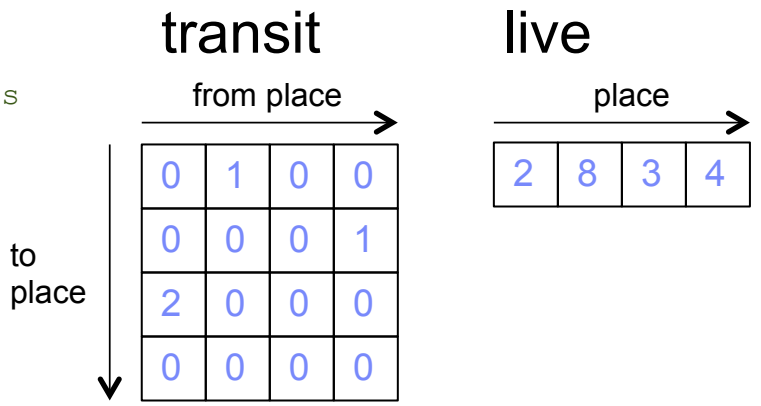
The implementation reduces ‘at’ to a special case of ‘finish’.

Abstractly, finish is a set of counters

Simplified illustration:

```
finish {        val v = new FinishCounters();
    ...             ...
}               f.wait(); // may throw MultipleExceptions
```

```
async {         f.begin(...); (); // may communicate
    ...             ...
}               f.end(...) // may communicate
```



Counters are used to

- **Wait for termination**
- **Throw DeadPlaceException**

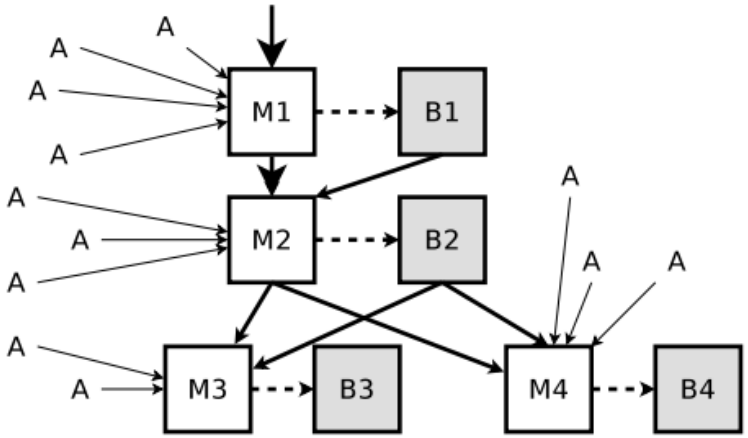
3 Possible Finish Implementations

Finish counters need to survive failure of place holding FinishCounters object…

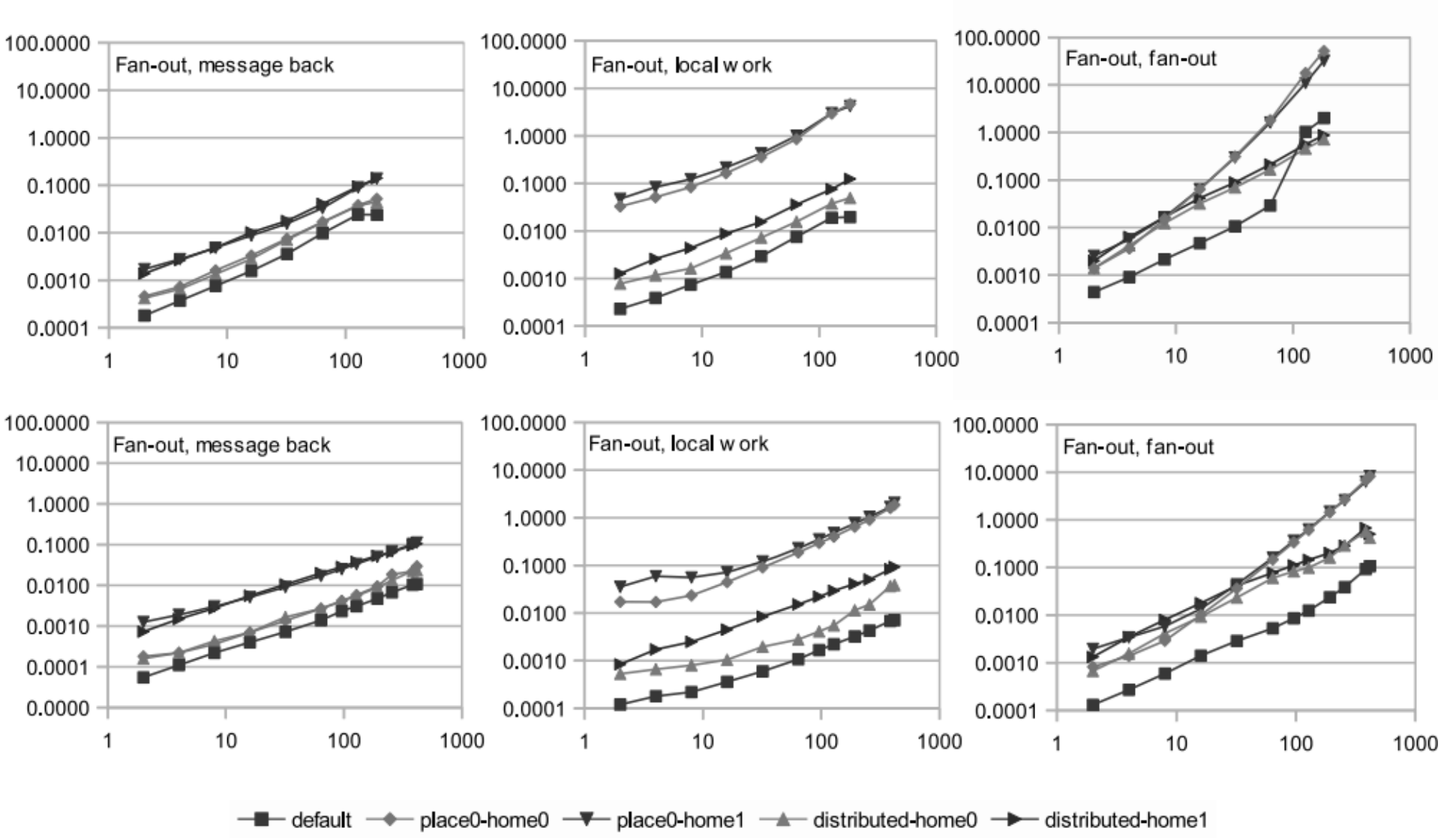
- **Store all finish state at place zero.**
  - Simple
  - Makes use of ‘immortal’ place zero.
  - No problem: If finishes are logically at place zero in the code.
  - Otherwise: Bottle neck at place zero.

- **Store all finish state in ZooKeeper**
  - From Hadoop project
  - External paxos group of processes
  - Lightweight resilient store
  - Still too much overhead (details in paper)

- **Distributed resilient finish.**
  - Finish state is replicated at one other node.
  - Execution aborted if both nodes die.
  - Best all round performance
  - No bottle neck at place zero

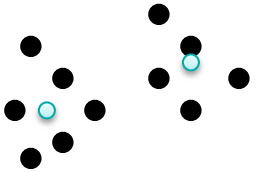


Finish Micro-benchmark results



Application – K-Means (Lloyd's algorithm)

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



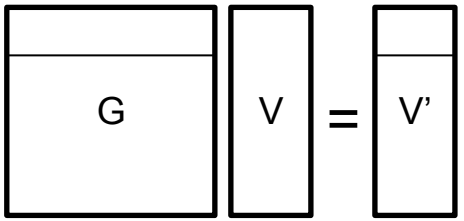
^ N=11, k=2

•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

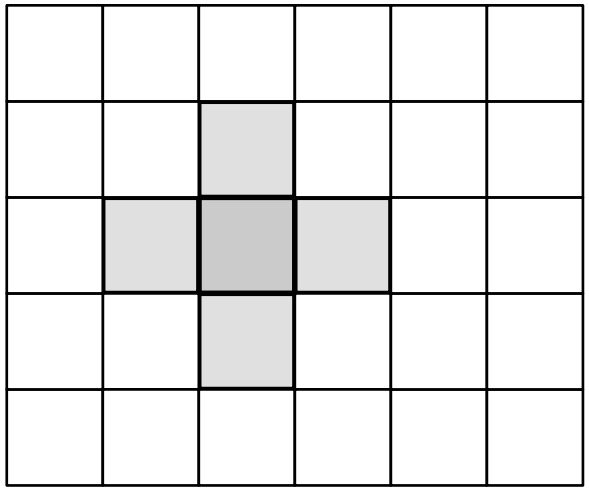
G distributed into row blocks. Every place starts with entire V, computes fragment of V'.
Every place communicates fragments of V to place 0 to be aggregated.
New V broadcast from place 0 for next iteration (G is never modified).

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

Equational theory for (Resilient) X10

$\langle s, g\rangle \cong \langle t, g\rangle$ **equivalent configurations** when

- transition steps are weakly bi-simulated
- ***under any modification of the shared heap by current activities***
  (object field update, object creation, place failure)

$\langle s, g\rangle \mathcal{R} \langle t, g\rangle$ *whenever*

1. $\vdash \mathsf{isSync}\, s$ *iff* $\vdash \mathsf{isSync}\, t$
2. $\forall p, \forall \Phi$ *environment move*

   if $\langle s, \Phi(g)\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle$ *then* $\exists t'.\ \langle t, \Phi(g)\rangle \overset{\lambda}{\Longrightarrow}_p \langle t', g'\rangle$

   with $\langle s', g'\rangle \mathcal{R} \langle t', g'\rangle$ *and viceversa*

models the update of g:

$dom(\Phi(g)) = dom(g)$ *and*
$\forall p{\in}dom(g)\ dom(g(p)) \subseteq dom(\Phi(g)(p))$

Bisimulation whose Bisimilarity is a congruence

Equational theory for (Resilient) X10

$$\{\{s\ \ t\}\ \ u\} \cong \{s\ \ \{t\ \ u\}\}$$

$$\vdash \mathsf{isAsync}\ s \quad \texttt{try}\,\{s\ t\}\,\texttt{catch}\,u \cong \{\texttt{try}\,s\,\texttt{catch}\,u\ \ \texttt{try}\,t\,\texttt{catch}\,u\}$$

$$\texttt{at}(p)\{s\ t\} \not\cong_{\mathrm{R}} \{\texttt{at}(p)s\ \ \texttt{at}(p)t\}$$

$$\texttt{at}(p)\texttt{at}(q)s \not\cong_{\mathrm{R}} \texttt{at}(q)s$$

$$\texttt{async}\ \texttt{at}(p)s \not\cong_{\mathrm{R}} \texttt{at}(p)\,\texttt{async}\,s$$

$$\texttt{finish}\,\{s\ \ t\} \cong \texttt{finish}\,s\ \ \texttt{finish}\,t$$

$$\texttt{finish}\,\{s\ \ \texttt{async}\,t\} \cong \texttt{finish}\,\{s\ \ t\}$$

$$\texttt{finish}\ \texttt{at}(p)\,s \not\cong_{\mathrm{R}} \texttt{at}(p)\,\texttt{finish}\,s$$

if s throws a sync exc. and home is failed, then l.h.s. throws a masked DPEx while r.h.s. re-throws vx since synch exc are not masked by DPE

(Par Left)

$$\frac{\langle s, g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle \mid g'}{\begin{array}{ll} \lambda{=}\epsilon, v\times & \langle\{s\ \ t\}, g\rangle \xrightarrow{\lambda}_p \langle\{s'\ \ t\}, g'\rangle \mid \langle t, g'\rangle \\ \lambda = v\otimes & \langle\{s\ \ t\}, g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle \mid g' \end{array}}$$

(Par Right)

$$\frac{\vdash \mathsf{isAsync}\, t \quad \langle s, g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle \mid g'}{\langle\{t\ \ s\}, g\rangle \xrightarrow{\lambda}_p \langle\{t\ \ s'\}, g'\rangle \mid \langle t, g'\rangle}$$

(Place Shift)

$$\frac{(v', g') = \mathsf{copy}(v, q, g)}{\langle \mathtt{at}(q)\mathtt{val}\, x = v\, \mathtt{in}\, s,\ g\rangle \longrightarrow_p \langle \overline{\mathtt{at}}(q)\{s[{}^{v'}/_{x}]\ \mathtt{skip}\}, g'\rangle}$$

(At)

$$\frac{\langle s, g\rangle \xrightarrow{\lambda}_q \langle s', g'\rangle \mid g'}{\langle \overline{\mathtt{at}}(q)\, s, g\rangle \xrightarrow{\lambda}_p \langle \overline{\mathtt{at}}(q)\, s', g'\rangle \mid g'}$$

(Spawn)

$$\frac{}{\langle \texttt{async}\, s, g\rangle \longrightarrow_p \langle \overline{\texttt{async}}\, s, g\rangle}$$

(Async)

$$\frac{\langle s, g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle \mid g'}{\begin{array}{rl} \lambda = \epsilon & \langle \overline{\texttt{async}}\, s, g\rangle \xrightarrow{\lambda}_p \langle \overline{\texttt{async}}\, s', g'\rangle \mid g' \\ \lambda = v\times, v\otimes & \langle \overline{\texttt{async}}\, s, g\rangle \xrightarrow{v\times}_p \langle \overline{\texttt{async}}\, s', g'\rangle \mid g' \end{array}}$$

(Finish)

$$\frac{\langle s, g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle}{\langle \texttt{finish}_\mu\, s, g\rangle \longrightarrow_p \langle \texttt{finish}_{\mu\cup\lambda}\, s', g'\rangle}$$

(End Finish)

$$\frac{\langle s, g\rangle \xrightarrow{\lambda}_p g' \quad \lambda' = \mathsf{E}\otimes \;\; if\, \lambda\cup\mu\neq\emptyset\; else\; \epsilon}{\langle \texttt{finish}_\mu\, s, g\rangle \xrightarrow{\lambda'}_p g'}$$

(Exception)

$$\frac{}{\langle \texttt{throw}\, v, g\rangle \xrightarrow{v\otimes}_p g}$$

(Try)

$$\frac{\langle s, g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle \mid g'}{\begin{array}{rl} \lambda{=}\epsilon, v\times & \langle \texttt{try}\, s\, \texttt{catch}\, t, g\rangle \xrightarrow{\lambda}_p \langle \texttt{try}\, s'\, \texttt{catch}\, t, g'\rangle \mid g' \\ \lambda{=}v\otimes & \langle \texttt{try}\, s\, \texttt{catch}\, t, g\rangle \longrightarrow_p \langle \{s'\ \ t\}, g'\rangle \mid \langle t, g'\rangle \end{array}}$$

(Skip)

$$\frac{}{\langle \texttt{skip}, g\rangle \longrightarrow_p g}$$

Plus rules for expression evaluation