X10 Tutorial

December 2006
IBM Research

This work has been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004

X10 Team

- **X10 Core Team**
  - **Rajkishore Barik**
  - **Vincent Cave**
  - **Chris Donawa**
  - **Allan Kielstra**
  - **Igor Peshansky**
  - **Christoph von Praun**
  - **Vijay Saraswat**
  - **Vivek Sarkar**
  - **Tong Wen**
- **X10 Tools**
  - **Philippe Charles**
  - **Julian Dolby**
  - **Robert Fuhrer**
  - **Frank Tip**
  - **Mandana Vaziri**
- **Emeritus**
  - **Kemal Ebcioglu**
  - **Christian Grothoff**
- **Research colleagues**
  - **R. Bodik, G. Gao, R. Jagadeesan, J. Palsberg, R. Rabbah, J. Vitek**
  - **Several others at IBM**

Recent publications

1. "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing", P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, V. Sarkar. OOPSLA conference, October 2005.
2. "Concurrent Clustered Programming", V. Saraswat, R. Jagadeesan. CONCUR conference, August 2005.
3. “Experiences with an SMP Implementation for X10 based on the Java Concurrency Utilities Rajkishore Barik, Vincent Cave, Christopher Donawa, Allan Kielstra,Igor Peshansky, Vivek Sarkar. Workshop on Programming Models for Ubiquitous Parallelism (PMUP), September 2006.
4. "An Experiment in Measuring the Productivity of Three Parallel Programming Languages”, K. Ebcioglu, V. Sarkar, T. El-Ghazawi, J. Urbanic. P-PHEC workshop, February 2006.
5. "X10: an Experimental Language for High Productivity Programming of Scalable Systems", K. Ebcioglu, V. Sarkar, V. Saraswat. P-PHEC workshop, February 2005.

Tutorial outline

1) X10 in a nutshell

2) Sequential X10
- Type system
- Standard library

3) Concurrency in X10
- Activities
- Atomic blocks
- Clocks, clocked variables

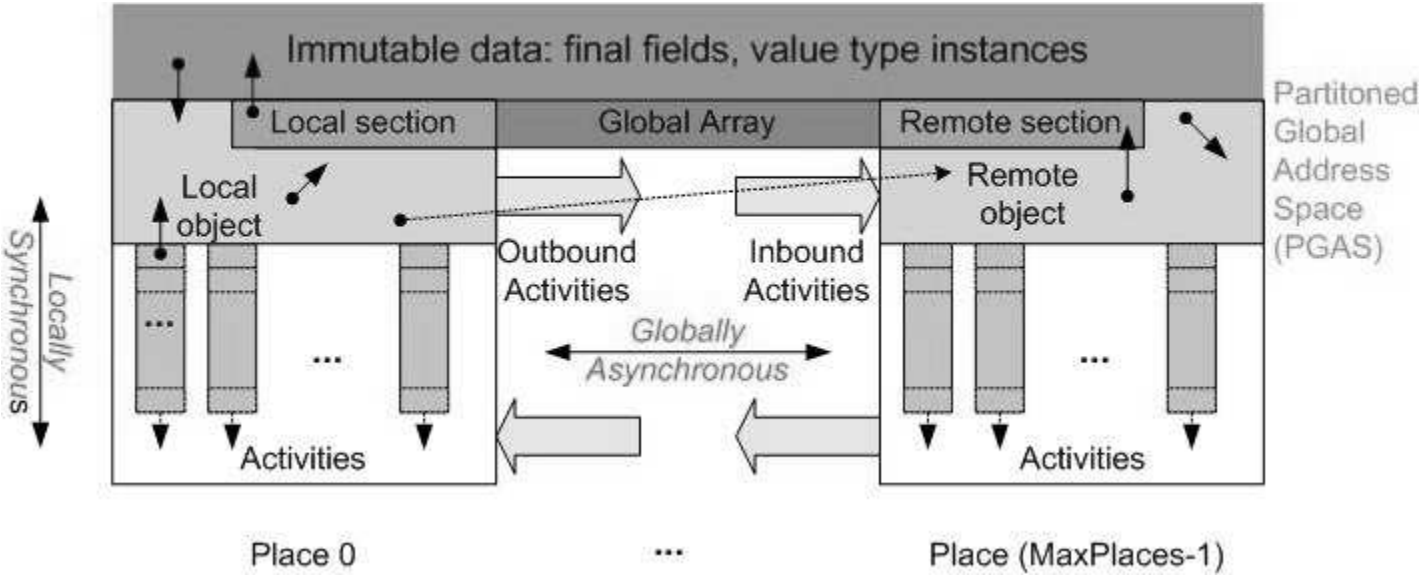
4) X10 arrays
- Points
- Regions

5) Distributed X10
- Places
- Distributions
- Distributed arrays

6) Further examples

X10 in a Nutshell

X10 Programming Model



Storage classes:

- **Activity-local**
- **Place-local**
- **Partitioned global**
- **Immutable**

- Dynamic parallelism with a *Partitioned Global Address Space*
- *Places* encapsulate binding of activities and globally addressable data
  - Number of places currently fixed at launch time
- All concurrency is expressed as *asynchronous activities* – subsumes threads, structured parallelism, messaging, DMA transfers, etc.
- *Atomic sections* enforce mutual exclusion of co-located data
  - No place-remote accesses permitted in atomic section
- *Immutable* data offers opportunity for single-assignment parallelism

X10 v0.41 Cheat sheet

Stm:

async ***[*** (***Place***) ***] [***clocked ***ClockList] Stm***

when (***SimpleExpr***) ***Stm***

finish ***Stm***

next; ***c.resume()*** ***c.drop()***

for(***i : Region***) ***Stm***

foreach (***i : Region***) ***Stm***

ateach (***I : Distribution***) ***Stm***

Expr:

ArrayExpr

ClassModifier : Kind

MethodModifier: atomic

DataType:

ClassName | InterfaceName | ArrayType

nullable ***DataType***

future ***DataType***

Kind :

value | reference

x10.lang has the following classes (among others)

point, range, region, distribution, clock, array

Some of these are supported by special syntax.

Forthcoming support: closures, generics, dependent types, place types, implicit syntax, array literals.

Programming Technologies

X10 v0.41 Cheat sheet: Array support

ArrayExpr:

- new **ArrayType (Formal) { Stm }**
- ***Distribution Expr*** -- Lifting
- ***ArrayExpr*** [***Region***] -- Section
- ***ArrayExpr*** | ***Distribution*** -- Restriction
- ***ArrayExpr*** || ***ArrayExpr*** -- Union
- ***ArrayExpr***.overlay(***ArrayExpr***) -- Update
- ***ArrayExpr***. scan(***[fun [, ArgList]***)
- ***ArrayExpr***. reduce(***[fun [, ArgList]***)
- ***ArrayExpr***.lift(***[fun [, ArgList]***)

ArrayType:

- ***Type [Kind]*** [ ]
- ***Type [Kind]*** [region(N)]
- ***Type [Kind]*** [***Region***]
- ***Type [Kind]*** [***Distribution***]

Region:

- ***Expr : Expr*** -- 1-D region
- [***Range, …, Range***] -- Multidimensional Region
- ***Region*** && ***Region*** -- Intersection
- ***Region*** || ***Region*** -- Union
- ***Region*** – ***Region*** -- Set difference
- ***BuiltinRegion***

Dist:

- ***Region -> Place*** -- Constant distribution
- ***Distribution*** | ***Place*** -- Restriction
- ***Distribution*** | ***Region*** -- Restriction
- ***Distribution*** || ***Distribution*** -- Union
- ***Distribution*** – ***Distribution*** -- Set difference
- ***Distribution.***overlay (***Distribution***)
- ***BuiltinDistribution***

Language supports type safety, memory safety, place safety, clock safety.

Programming Technologies

Comparison with Java

X10 language builds on the Java language

Shared underlying philosophy: shared syntactic and semantic tradition, simple, small, easy to use, efficient to implement, machine independent

X10 does not have:

- Dynamic class loading
- Java's concurrency features
  - thread library, volatile, synchronized, wait, notify

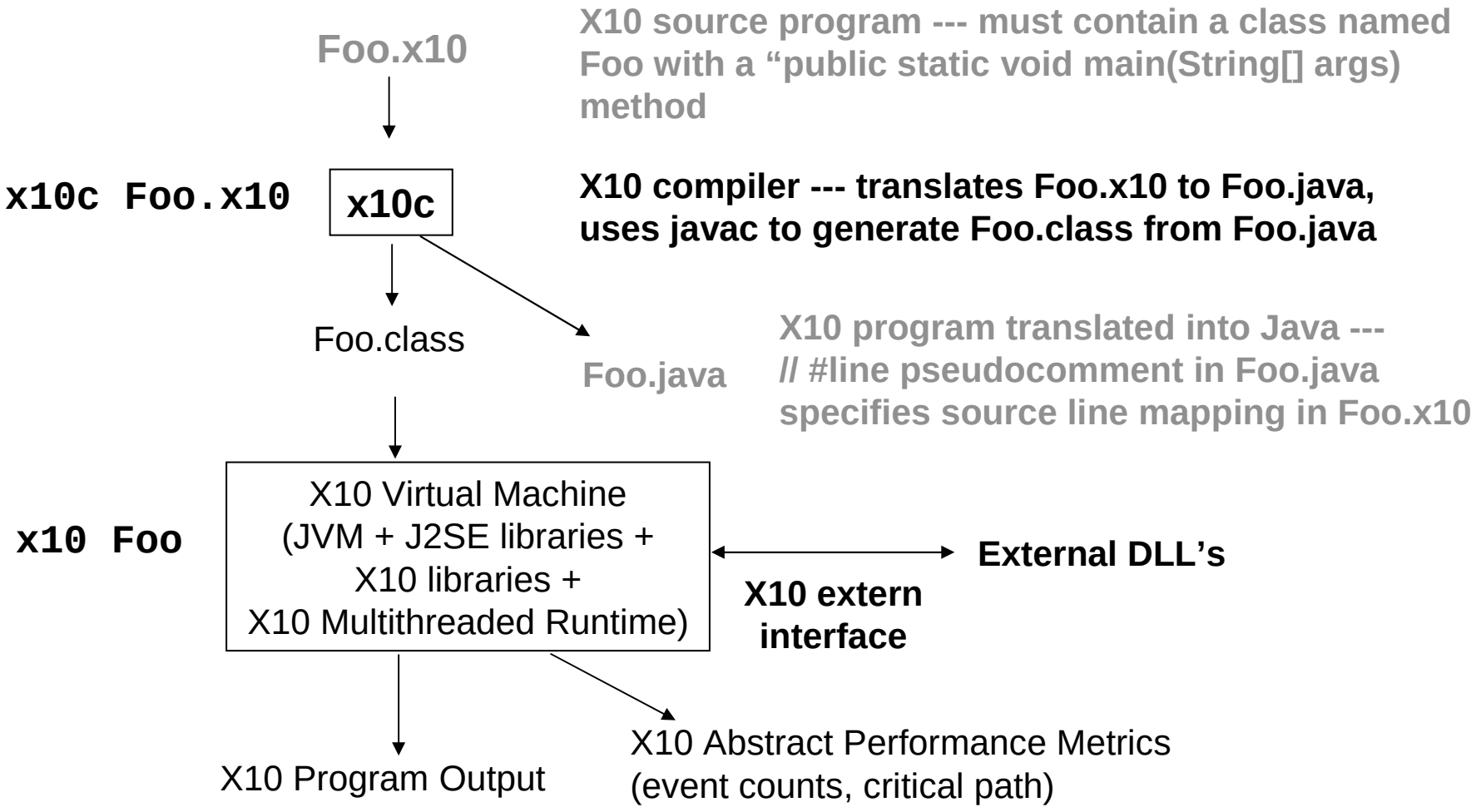
X10 restricts:

- Class variables and static initialization

X10 adds to Java:

- **value types, nullable**
- **Array language**
  - Multi-dimensional arrays, aggregate operations
- **New concurrency features**
  - activities (async, future), atomic blocks, clocks
- **Distribution**
  - places
  - distributed arrays

X10 prototype implementation



Examples of X10 compiler error messages

```
1) x10c TutError1.x10
TutError1.x10:8: Could not find field or local variable "evenSum".
            for (int i = 2 ; i <= n ; i += 2 ) evenSum += i;
                                                ^----^

2) x10c TutError2.x10
x10c: TutError2.x10:4:27:4:27: unexpected token(s) ignored

3) x10c TutError3.x10
x10c: C:\vivek\eclipse\workspace\x10\examples\Tutorial\TutError3.java:49:
    local variable n is accessed from within inner class; needs to be declared
    final
```

Case 1: Error message identifies source file and line number

Case 1: Carats indicate column range

Case 2: Error message identifies source file, line number, and column range

Case 3: Error message reported by Java compiler – look for #line comment in .java file to identify X10 source location

Tutorial outline

1) X10 in a nutshell

2) Sequential X10
- Type system
- Standard library

3) Concurrency in X10
- Activities
- Atomic blocks
- Clocks, clocked variables

4) X10 arrays
- Points
- Regions

5) Distributed X10
- Places
- Distributions
- Distributed arrays

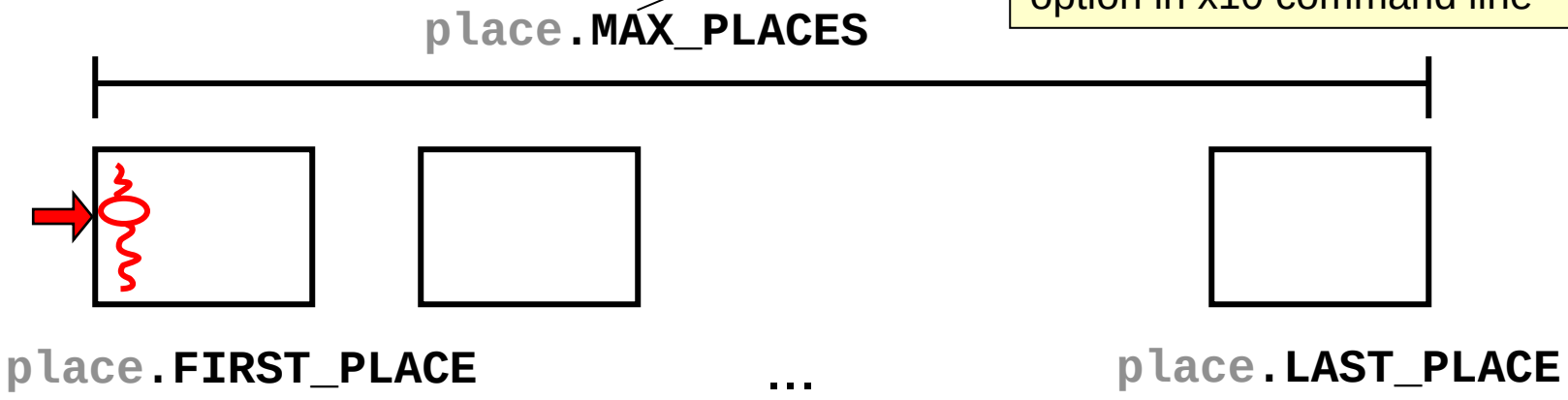
6) Further examples

Sequential X10

- **Overview**
- **value types**
- **nullable types**
- **Safety properties**

Sequential X10

Runtime constant.
Can be changed by using the NUMBER_OF_LOCAL_PLACES option in x10 command line



root activity

```
public class HelloWorld {
  public static void main(String[] args) {
      System.out.println("Hello, world!");
  }
}
```

Value types : immutable instances

value class

- Can only extend value class or x10.lang.Object.
- All fields are implicitly **final**
- Can only be extended by value classes.
- May contain fields with reference type.
- May be implemented by reference or copy.

Values are equal (==) if their fields are equal, recursively.

```
public value complex {
  double im, re;
  public complex(double im,
                 double re) {
    this.im = im;
    this.re = re;
  }
  public complex add(complex a)
  {
   return new complex(im+a.im,
                      re+a.re);
 } …
}
```

Memory safety

Runtime invariants

- **An object may only access memory within its representation, and other objects it has a reference to.**
  - X10 supports no pointer arithmetic.
  - Array access is bounds-checked dynamically (if necessary).
- **No "ill mem ref"**
  - No object can have a reference to an object who's memory has been freed.
  - X10 uses garbage collection.
- **Every value read from a location has been previously written into the location.**
  - No uninitialized variables.

Pointer safety

X10 supports the nullable type constructor.

- For any datatype T, the datatype nullable<T> contains all the value of T and null.
- If a method is invoked or a field is accessed on the value null, a NullPointerException (NPE) is thrown.

Runtime invariant

No operation on a value of type T, which is not of the form nullable S, can throw an NPE.

```
public interface Table {
     void put(Object o);
     nullable<Object> get(Object o);
}
public class Foo {
  boolean check (Table h) {
     return h.get(this) != null;
  }
}
```

May return null

Cannot throw NPE.

x10.lang standard library

Java package with “built in” classes that provide support for selected X10 constructs

- Standard types
  - **boolean, byte, char, double, float, int, long, short, String**
- **x10.lang.Object** -- root class for all instances of X10 objects
- **x10.lang.clock** --- clock instances & clock operations
- **x10.lang.dist** --- distribution instances & distribution operations
- **x10.lang.place** --- place instances & place operations
- **x10.lang.point** --- point instances & point operations
- **x10.lang.region** --- region instances & region operations

All X10 programs implicitly import the x10.lang.* package, so the x10.lang prefix can be omitted when referring to members of x10.lang.* classes

- e.g., place.MAX_PLACES, dist.factory.block([0:100,0:100]), …

Similarly, all X10 programs also implicitly import the java.lang.* package

- e.g., X10 programs can use Math.min() and Math.max() from java.lang

Tutorial outline

1) X10 in a nutshell

2) Sequential X10
- Type system
- Standard library

3) Concurrency in X10
- Activities
- Atomic blocks
- Clocks, clocked variables

4) X10 arrays
- Points
- Regions

5) Distributed X10
- Places
- Distributions
- Distributed arrays

6) Further examples

Concurrency in X10

- **async, finish**
- **future, force**
- **foreach**
- **Global vs. local termination**
- **Exception handling**
- **Behavioral annotations**
- **atomic**
- **Memory model**
- **clocks**

async

Stmt ::= async ***PlaceExpSingleListopt Stmt***

cf Cilk's spawn

async (P) S

- Creates a new child activity at place P, that executes statement S
- Returns immediately
- S may reference **final** variables in enclosing blocks
- Activities cannot be named
- Activity cannot be aborted or cancelled

```
// global dist. array
final double a[D] =  …;
final int k = …;

async ( a.distribution[99] ) {
    // executed at A[99]'s
    // place
    atomic a[99] = k;
}
```

- **Memory model: hb edge between stm before async and start of async.**

finish

finish S

- Execute S, but wait until all (transitively) spawned asyncs have terminated.

Rooted exception model

- Trap all exceptions thrown by spawned activities.
- Throw an (aggregate) exception if any spawned async terminates abruptly.
- implicit **finish** at main activity

finish is useful for expressing "synchronous" operations on (local or) remote data.

Stmt ::= **finish** ***Stmt***

cf Cilk's sync

```
finish ateach(point [i]:A)
    A[i] = i;

finish async
    (A.distribution [j])
    A[j] = 2;

// all A[i]=i will complete
// before A[j]=2;
```

- **Memory model: hb edge between last stm of each async and stm after finish S.**

Termination

Local termination:

Statement *s* terminates locally when activity has completed all its computation with respect to *s*.

Global termination:

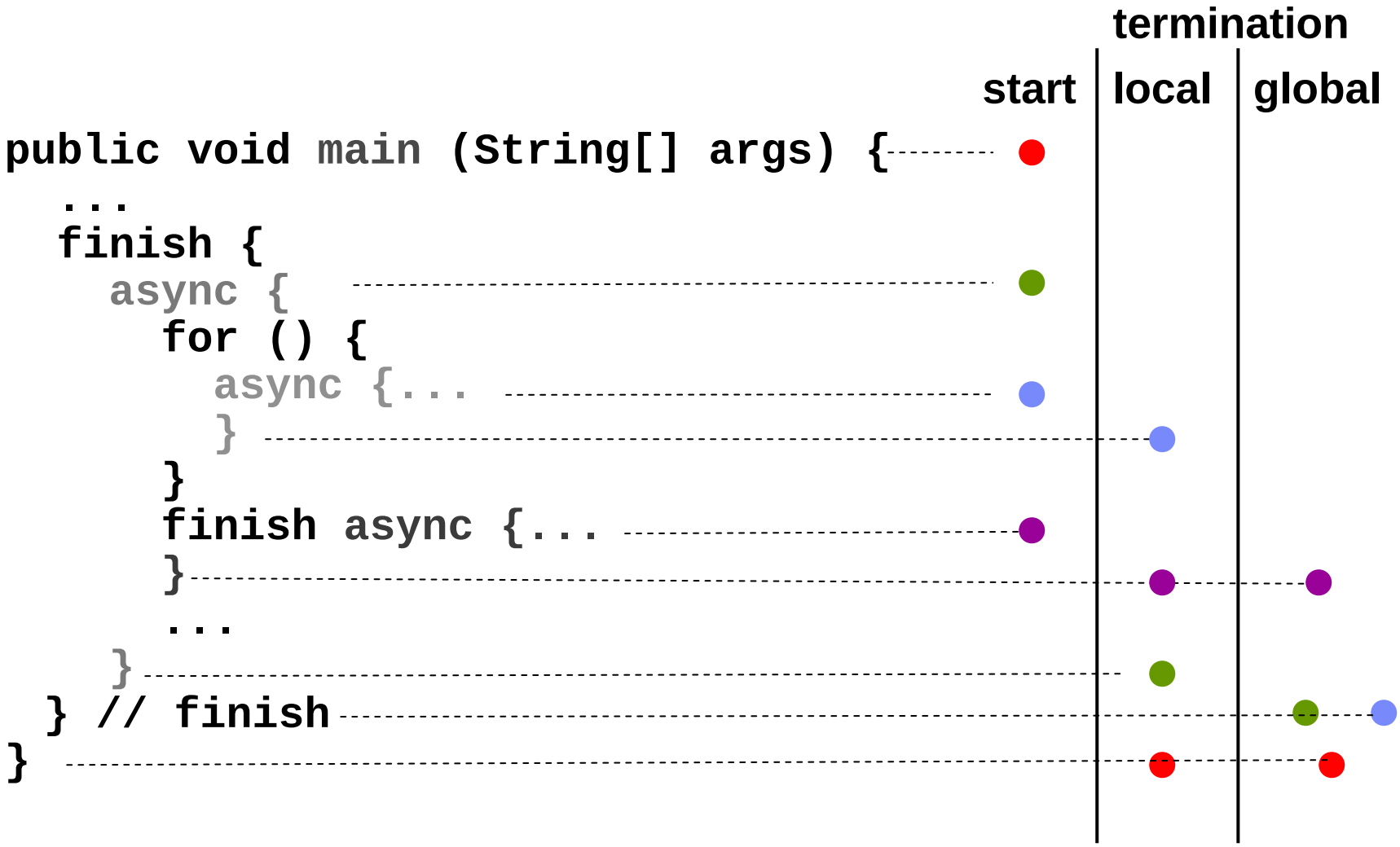
Local termination + activities that have been spawned by *s* terminated globally (recursive definition)

→ main function is **root activity**

→ program terminates iff root activity terminates. (implicit finish at root activity)

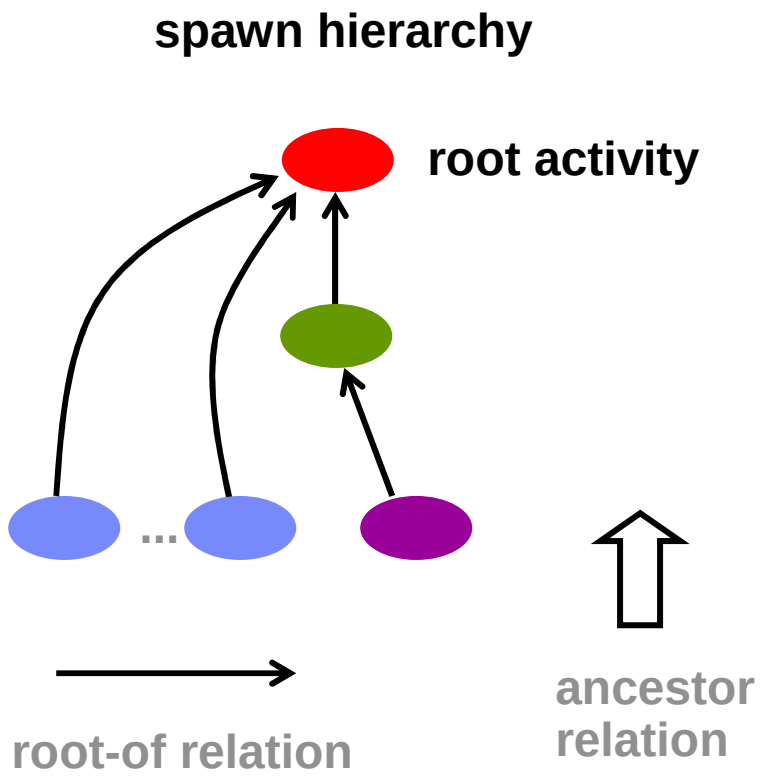
→ 'daemon threads' (child outlives root activity) not allowed in X10

Termination (example)



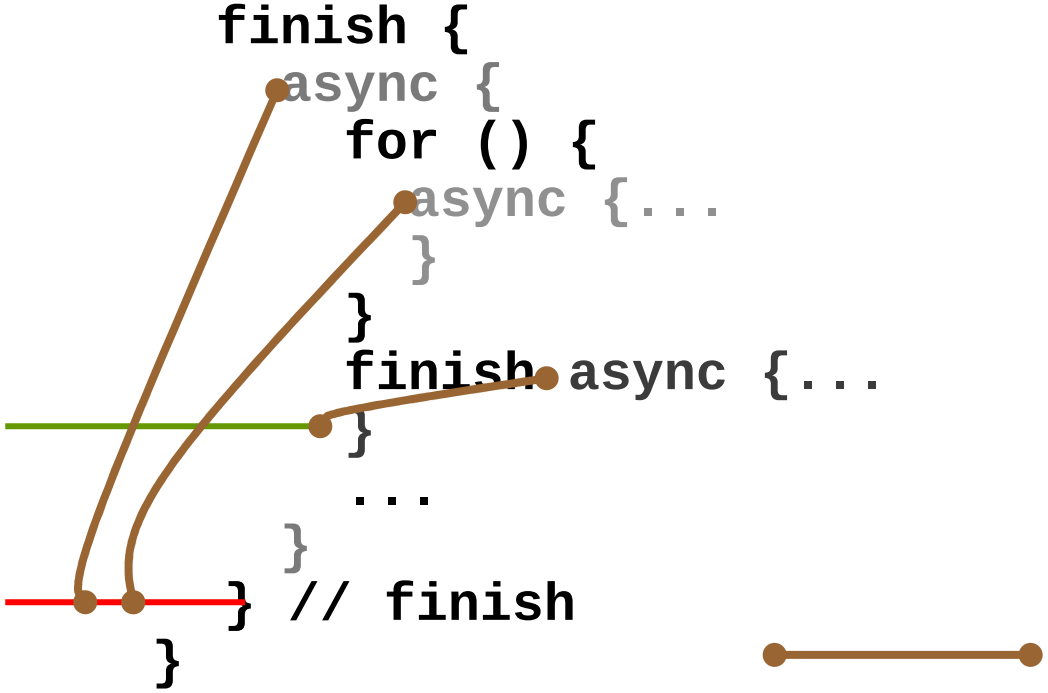
Rooted computation X10

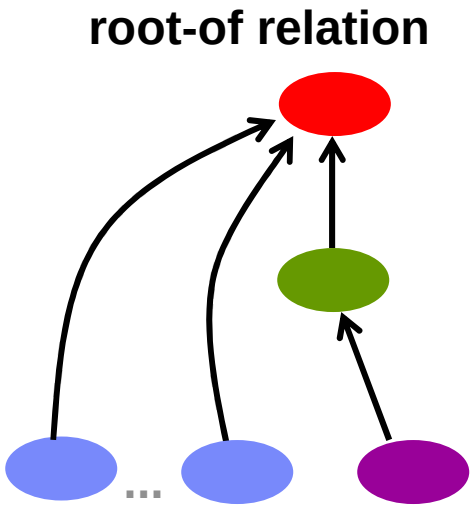
```
public void main (String[] args) {
  ...
  finish {
    async {
      for () {
        async {...
        }
      }
      finish async {...
      }
      ...
    }
  } // finish
}
```



Rooted exception model

```
public void main (String[] args) {
  ...
  finish {
    async {
      for () {
        async {...
        }
      }
      finish async {...
      }
      ...
    }
  } // finish
}
```





exception flow along root-of relation

Propagation along the **lexical scoping**:

Exceptions that are not caught inside an activity are propagated to the nearest suspended ancestor in the root-of relation.

Example: rooted exception model (async)

```
int result = 0;
try {
   finish {
     ateach (point [i]:dist.factory.unique()) {
       throw new Exception (“Exception from “+here.id)
     }
     result = 42;
   } // finish
} catch (x10.lang.MultipleExceptions me) {
   System.out.print(me);
}
assert (result == 42); // always true
```

- no exceptions are ‘thrown on the floor’
- exceptions are propagated across activity and place boundaries

Behavioral annotations

nonblocking

On *any* input store, a nonblocking method can continue execution or terminate. (dual: **blocking**, default: **nonblocking**)

recursively nonblocking

Nonblocking, and every spawned activity is recursively nonblocking.

local

A local method guarantees that its execution will only access variables that are local to the place of the current activity.
(dual: **remote**, default: **local**)

sequential

Method does not create concurrent activities.
In other words, method does not use **async**, **foreach**, **ateach.**
(dual: **parallel**, default: **parallel**)

Sequential and nonblocking imply recursively nonblocking.

Static semantics

- Behavioral annotations are checked with a conservative intra-procedural data-flow analysis.
- Inheritance rule: Annotations must be preserved or strengthened by overriding methods.
- Multiple behavioral annotations must be mutually consistent.

foreach

foreach (*FormalParam*: *Expr*) *Stmt*

foreach (point p: R) S

- Creates |R| async statements in parallel at current place.

```
foreach (point p:R) S
```

```
for (point p: R)
  async { S }
```

- Termination of all (recursively created) activities can be ensured with **finish**.
- **finish foreach** is a convenient way to achieve master-slave fork/join parallelism (OpenMP programming model)

atomic

Stmt ::= atomic ***Statement***
MethodModifier ::= atomic

- Atomic blocks are conceptually executed in a single step while other activities are suspended: isolation and atomicity.
- An atomic block ...
  - must be **nonblocking**
  - must not create concurrent activities (**sequential**)
  - must not access remote data (**local**)
  - **Memory model: end of tx hb start of next tx in the same place.**

```
// target defined in lexically
// enclosing scope.
atomic boolean CAS(Object old,
                   Object new) {
   if (target.equals(old)) {
     target = new;
     return true;
   }
   return false;
}
```

```
// push data onto concurrent
// list-stack
Node node = new Node(data);
atomic {
     node.next = head;
     head = node;
}
```

Static semantics of atomic blocks

An **atomic** block must...be **local, sequential, nonblocking**:

- ...not include blocking operations
  - no **await**, no **when**, no calls to **blocking** methods
- ... not include access to data at remote places
  - no **ateach**, no **future**, only calls to **local** methods
- ... not spawn other activities
  - no **async**, no **foreach**, only calls to **sequential** methods

Using X10 method annotations

A method declaration, foo(), can be annotated with:

- **nonblocking** ➔ no static occurrence in foo() of when, force(), next(); any method that foo() invokes must also be annotated as **nonblocking**
- **local** ➔ all data accessed in foo() is statically guaranteed to be *place-local*; any method that foo() invokes must also be annotated as **local**

To check if an activity (async, foreach, ateach, future) is **local nonblocking**

- Check local body of activity to ensure that it satisfies the conditions
- Check that all methods called in activity are also annotated (and checked) as **local nonblocking**
- NOTE: this also works in the presence of recursion

Exceptions in atomic blocks

- Atomicity guarantee only for successful execution.
  - Exceptions should be caught inside atomic block
  - Explicit undo in the catch handler

```
boolean move(Collection s, Collection d, Object o) {
   atomic {
     if (!s.remove(o)) {
       return false; // object not found
     } else {
       try {
         d.add(o);
       } catch (RuntimeException e) {
         s.add(o); // explicit undo
         throw e;  // exception
       }
       return true; // move succeeded
     }
   }
}
```

cf. [Harris CSJP'04]

- (Uncaught) exceptions propagate across the atomic block boundary; atomic terminates on normal or abrupt termination of its block.

Data races with async / foreach

```
final double arr[R] =  …; // global array

class ReduceOp {
   double accu = 0.0;
   double sum ( double[.] arr ) {
finish foreach (point p: arr) {
          atomic accu += arr[p];
       }
       return accu;
}
```

concurrent conflicting access to shared variable: data race

X10 guideline for avoiding data races:

- access shared variables inside an atomic block
- combine **ateach** and **foreach** with **finish**
- declare data to be read-only where possible (final or value type)

Memory Model

Please see: http://www.saraswat.org/rao.html

- **X10 v 0.41 specifies sequential consistency per place.**
  - Not workable.
- **We are considering a weaker memory model.**
- **Built on the notion of atomic: identify a step as the basic building block.**
  - A step is a partial write function.
- **Use links for non hb-reads.**

- **A process is a pomset of steps closed under certain transformations:**
  - Composition
  - Decomposition
  - Augmentation
  - Linking
  - Propagation
- **There may be opportunity for a weak notion of atomic: decouple *atomicity* from *ordering*.**

Correctly synchronized programs behave as SC.

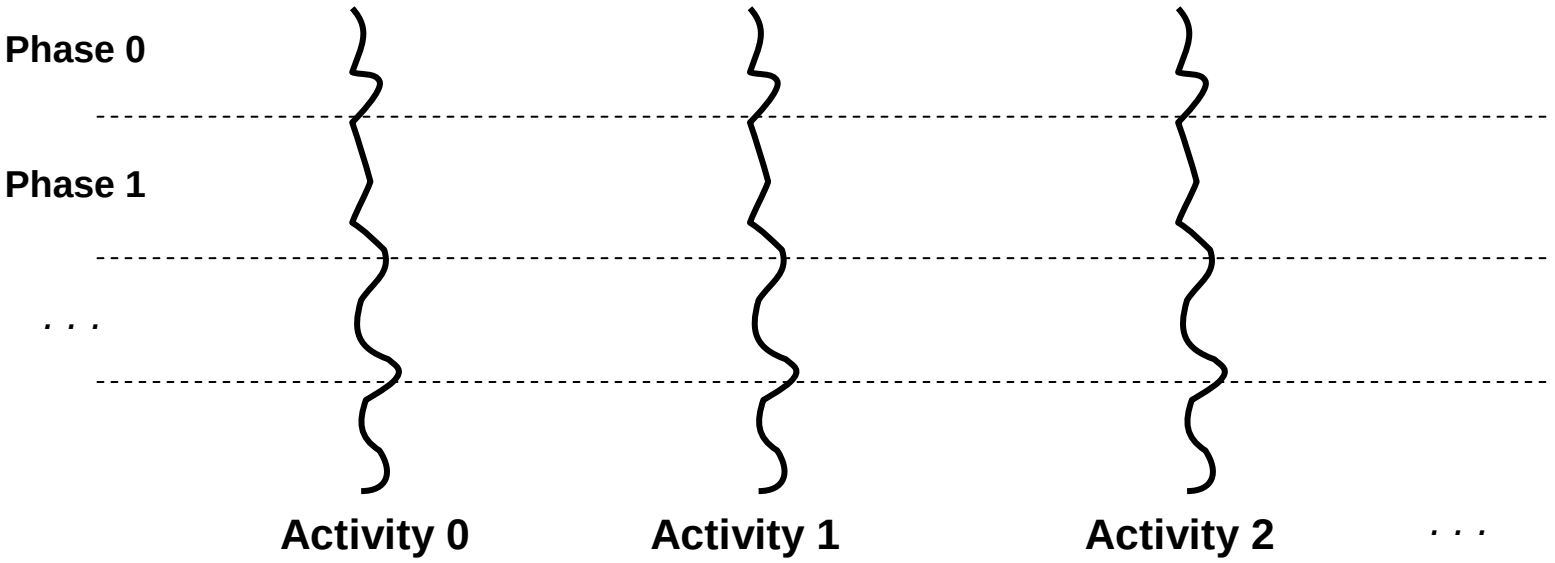
Correctly synchronized programs = programs whose SC executions have no races.

Concurrency Control: Clocks

- **clock**
- **Clocks safety**
- **Clocked variables**

Clocks: Motivation

- Activity coordination using **finish** and **force()** is accomplished by checking for activity termination
- However, there are many cases in which a producer-consumer relationship exists among the activities, and a “barrier”-like coordination is needed without waiting for activity termination
  - The activities involved may be in the same place or in different places



Clocks (1/2)

clock c = clock.factory.clock();

- Allocate a clock, register current activity with it. Phase 0 of c starts.

async(…) clocked (c1,c2,…) S
ateach(…) clocked (c1,c2,…) S
foreach(…) clocked (c1,c2,…) S

- Create async activities registered on clocks c1, c2, …

c.resume();

- Nonblocking operation that signals completion of work by current activity for this phase of clock c

next;

- Barrier --- suspend until all clocks that the current activity is registered with can advance. **c.resume()** is first performed for each such clock, if needed.
- Next can be viewed like a “finish” of all computations under way in the current phase of the clock

Clocks (2/2)

c.drop();

- Unregister with c. A terminating activity will implicitly drop all clocks that it is registered on.

c.registered()

- Return true iff current activity is registered on clock c
- **c.dropped()** returns the opposite of **c.registered()**

ClockUseException

- Thrown if an activity attempts to transmit or operate on a clock that it is not registered on
- Or if an activity attempts to transmit a clock in the scope of a finish

Semantics

Static semantics

- An activity may operate only on those clocks it is registered with.
- In **finish S,S** may not contain any (top-level) clocked asyncs.

Dynamic semantics

- A clock **c** can advance only when all its registered activities have executed **c.resume()**.
- An activity may not pass-on clocks on which it is not live to sub-activities.
- An activity is deregistered from a clock when it terminates

- **Memory model: hb edge between next stm of all registered activities on c, and their subsequent stm**

Supports over-sampling, hierarchical nesting.
No explicit operation to register a clock.

Behavioral annotations for clocks

clocked (c0,..., ck).

- A method *m* that spawns an **async clocked(c0,...,ck)** must declare {c0,...,ck} (or a superset) in its annotation: **clocked (c0,..., ck).**
- {c0,...,ck} are fields of type clock declared in the calss that declares *m.*

Example (TutClock1.x10)

```
finish async {
  final clock c = clock.factory.clock();
  foreach (point[i]: [1:N]) clocked (c) {
     while ( true ) {
       int old_A_i = A[i];
       int new_A_i = Math.min(A[i],B[i]);
       if ( i > 1 )
          new_A_i = Math.min(new_A_i,B[i-1]);
       if ( i < N )
          new_A_i = Math.min(new_A_i,B[i+1]);
       A[i] = new_A_i;
       next;
       int old_B_i = B[i];
       int new_B_i = Math.min(B[i],A[i]);
       if ( i > 1 )
          new_B_i = Math.min(new_B_i,A[i-1]);
       if ( i < N )
          new_B_i = Math.min(new_B_i,A[i+1]);
       B[i] = new_B_i;
       next;
       if ( old_A_i == new_A_i && old_B_i == new_B_i )
          break;
     } // while
   } // foreach
} // finish async
```

parent transmits clock to child

exiting from while loop terminates activity for iteration i, and automatically deregisters activity from clock

Programming Technologies

Clock safety

- An activity may be registered on one or more clocks
- Clock c can advance only when all activities registered with the clock have executed c.resume() and all posted activities have terminated globally.

Runtime invariant: Clock operations are guaranteed to be deadlock-free.

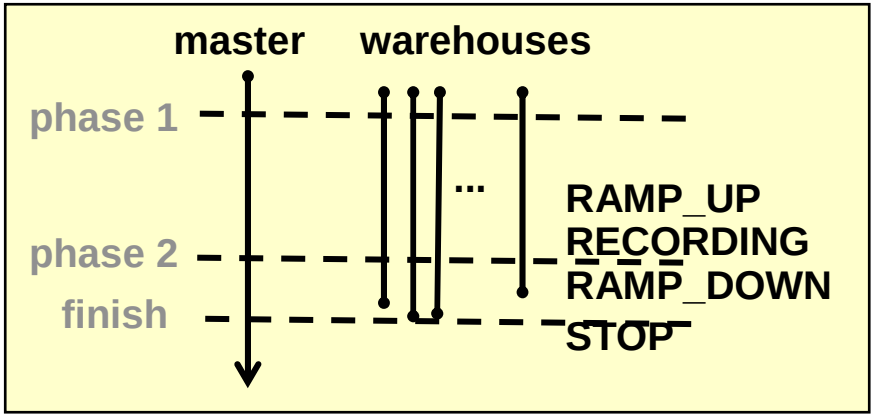
Deadlock freedom

- **Central theorem of X10:**
  - Arbitrary programs with async, atomic, finish (and clocks) are deadlock-free.
- **Key intuition:**
  - atomic is deadlock-free.
  - finish has a tree-like structure.
  - clocks are made to satisfy conditions which ensure tree-like structure.
  - Hence no cycles in wait-for graph.
- **Where is this useful?**
  - Whenever synchronization pattern of a program is independent of the data read by the program
  - True for a large majority of HPC codes.
  - (Usually not true of reactive programs.)

Clock example: SPECjbb

```
finish async {
  final clock c = new clock();
  final Company company =
  createCompany(...);
  for (int w : [0:wh_num]) {
     async clocked(c) { // a warehouse
        int mode;
        atomic { mode = company.mode; };
        initialize;
        next; // 1.
        while (mode != STOP) {
          select a transaction;
          think;
          process the transaction;
          if (mode == RECORDING)
            record data;
          if (mode == RAMP_DOWN)
            next; // 2.
          atomic { mode = company.mode; };
        } // while
     } // a warehouse
  } // for
 // ------ continued next column -->
```

```
  // master activity
  next; // 1.
  atomic { company.mode = RAMP_UP; };
  sleep rampuptime;
  atomic { company.mode = RECORDING; };
  sleep recordingtime;
  atomic { company.mode = RAMP_DOWN; };
  next; // 2.
  // all clients in RAMP_DOWN
  company.mode = STOP;
} // finish async
// simulation completed.
print results.
```



Programming Technologies

Futures

future

Expr ::= **future** ***PlaceExpSingleListopt {Expr }***

future (P) S

- Creates a new child activity at place P, that executes statement S;
- Returns immediately.
- S may reference **final** variables in enclosing blocks.

future vs. **async**

- Return result from asynchronous computation
- Tolerate latency of remote access.

```
// global dist. array
final double a[D] =  …;
final int idx = …;

future<double> fd =
  future (a.distribution[idx])
  {
    // executed at a[idx]'s
    // place
      a[idx];
  };
```

future type

- no subtype relation between T and future<T>

future example

```
public class TutFuture1 {
    static int fib (final int n) {
        if ( n <= 0 ) return 0;
        if ( n == 1 ) return 1;
        future<int> x = future { fib(n-1) };
        future<int> y = future { fib(n-2) };
        return x.force() + y.force();
    }

    public static void main(String[] args) {
        System.out.println("fib(10) = " + fib(10));
    }
}
```

Divide and conquer: recursive calls execute concurrently.

Example: rooted exception model (future)

```
double div (final double divisor)
   future<double> f = future { return 42.0 / divisor; }
   double result;
   try {
     result = f.force();
   } catch (ArithmeticException e) {
     result = 0.0;
   }
   return result;
}
```

Exception is propagated when the future is forced.

Futures can deadlock

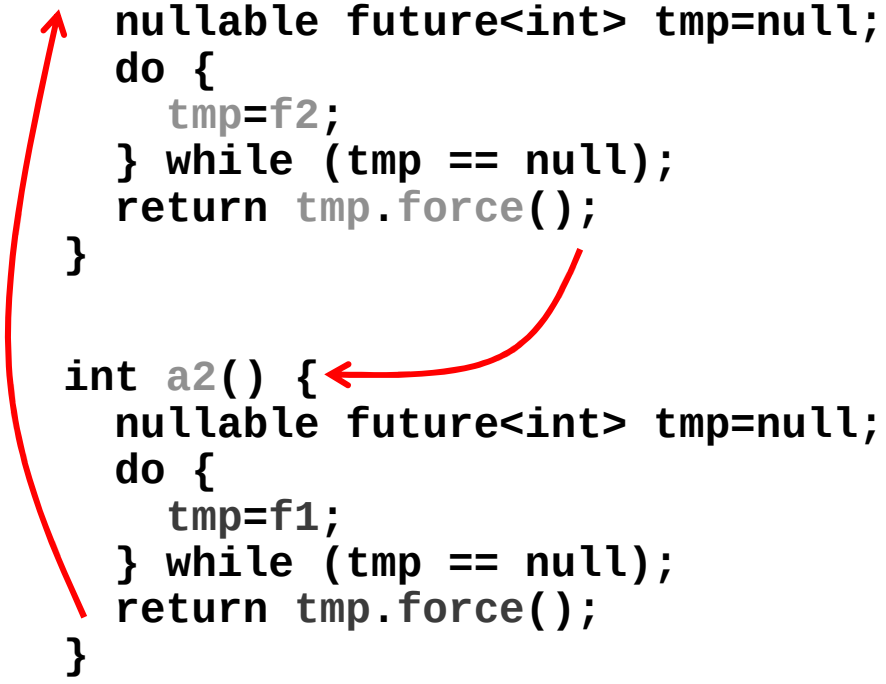
```
nullable future<int> f1=null;
nullable future<int> f2=null;

void main(String[] args) {
  f1 = future(here){a1()};
  f2 = future(here){a2()};
  f1.force();
}
```

cyclic wait condition

```
int a1() {
  nullable future<int> tmp=null;
  do {
    tmp=f2;
  } while (tmp == null);
  return tmp.force();
}

int a2() {
  nullable future<int> tmp=null;
  do {
    tmp=f1;
  } while (tmp == null);
  return tmp.force();
}
```



X10 guidelines to avoid deadlock:

- avoid futures as shared variables
- force called by same activity that created body of future, or a descendent.

Concurrency Control: Conditional atomic blocks, when, await

when

Stmt ::= WhenStmt
WhenStmt ::= when **(** ***Expr*** **)** ***Stmt*** **|**
WhenStmt or ***(Expr) Stmt***

- **when (E) S**
  - Activity suspends until a state in which the guard **E** is true.
  - In that state, **S** is executed atomically and in isolation.
- **Guard E**
  - boolean expression
  - must be **nonblocking**
  - must not create concurrent activities (**sequential**)
  - must not access remote data (**local**)
  - must not have side-effects (**const**)
- **await (E)**
  - syntactic shortcut for **when (E) ;**

```
class OneBuffer {
   nullable<Object> datum = null;
   boolean filled = false;

   void send(Object v) {
       when ( ! filled ) {
         datum = v;
         filled = true;
       }
   }

   Object receive() {
       when ( filled ) {
         Object v  = datum;
         datum = null;
         filled = false;
         return v;
       }
   }
}
```

Static semantics of guard for when / await

- boolean field
- boolean expression with field access or constant values

```
class BufferBuffer {
  ..
  void send(Object v) {
       when (size() < MAX_SIZE)
       {
         datum = v;
         filled = true;
       }
   }
   ...
}
```

compile-time error

Semaphores

```
class Semaphore {
   private boolean taken;

   void p() {                          acquire semantics
       when (!taken)
          taken = true;
   }

   atomic void v() {                   release semantics
       taken = false;
   }
}
```

Atomic blocks: Simplifying barrier synchronization

Original Java code

```
// Main thread (see spec.jbb.Company): …
// Wait for all threads to start.
synchronized (company.initThreadsStateChange) {
  while (initThreadsCount != threadCount) {
    try {
      initThreadsStateChange.wait();
    } catch (InterruptedException e) {…}
  }
} ...
// Tell everybody it's time for warmups.
mode = RAMP_UP;
synchronized (initThreadsCountMonitor) {
    initThreadsCountMonitor.notifyAll();
} ....
```

```
// Worker thread
// (see spec.jbb.TransactionManager): …
synchronized (company.initThreadsCountMonitor) {
  synchronized (company.initThreadsStateChange) {
    company.initThreadsCount++;
      company.initThreadsStateChange.notify();
    }
    try {
      company.initThreadsCountMonitor.wait();
    } catch (InterruptedException e) {…}
} ...
```

X10 atomic sections

```
// Main thread: …
// Wait for all threads to start.
when(company.initThreadsCount==
                threadCount) {
    mode = RAMP_UP;
    initThreadsCountReached = true;
} …
```

```
// Worker thread: …
atomic {
    company.initThreadsCount++;
}

await ( initThreadsCountReached );
    //barrier synch.
…
```

Programming Technologies

Tutorial outline

1) X10 in a nutshell

2) Sequential X10
- Type system
- Standard library

3) Concurrency in X10
- Activities
- Atomic blocks
- Clocks, clocked variables

4) X10 arrays
- Points
- Regions

5) Distributed X10
- Places
- Distributions
- Distributed arrays

6) Further examples

X10 Array Language

- **point, region, distribution**
- **Syntax extensions**
- **Initialization**
- **Multi-dimensional arrays**
- **Aggregate operations**

point

A **point** is an element of an n-dimensional Cartesian space (n>=1) with integer-valued coordinates e.g., [5], [1, 2], …

- Dimensions are numbered from 0 to n-1
- n is also referred to as the **rank** of the point

A point variable can hold values of different ranks e.g.,

- point p; p = [1]; … p = [2,3]; …

Operations

- p1.rank
  - returns rank of point p1
- p1.get(i)
  - returns element (i mod p1.rank) if i < 0 or i >= p1.rank
- p1.lt(p2), p1.le(p2), p1.gt(p2), p1.ge(p2)
  - returns true iff p1 is **lexicographically** <, <=, >, or >= p2
  - only defined when p1.rank and p1.rank are equal

Syntax extensions for points

- **Implicit syntax for points:**

```
point p = [1,2]  →  point p = point.factory(1,2)
```

- **Exploded variable declarations for points:**

```
point p [i,j]    // final int i,j
```

- **Typical uses :**

  - `for (point p [i, j] : r) { ... }`
  - `for (point [i, j] : r) { ... }`
  - `int sum (point [i,j], point [k, l])`
    `{ return [i+k, j+l]; }`
  - `int [] iarr = new int [2] (point [i,j]) { return i; }`

Example: point (TutPoint1)

```
public class TutPoint {
    public static void main(String[] args) {
        point p1 = [1,2,3,4,5];
        point p2 = [1,2];
        point p3 = [2,1];
        System.out.println("p1 = " + p1 +
               " ; p1.rank = " + p1.rank +
               " ; p1.get(2) = " + p1.get(2));
        System.out.println("p2 = " + p2 +
               " ; p3 = " + p3 + " ; p2.lt(p3) = " +
               p2.lt(p3));
    }
}
```

Console output:

```
p1 = [1,2,3,4,5] ; p1.rank = 5 ; p1.get(2) = 3
p2 = [1,2] ; p3 = [2,1] ; p2.lt(p3) = true
```

Rectangular regions

A **rectangular region** is the set of points contained in a rectangular subspace

A region variable can hold values of different ranks e.g.,

- **region R; R = [0:10]; … R = [-100:100, -100:100]; … R = [0:-1]; …**

Operations

- **R.rank** ::= # dimensions in region;
- **R.size()** ::= # points in region
- **R.contains(P)** ::= predicate if region R contains point P
- **R.contains(S)** ::= predicate if region R contains region S
- **R.equal(S)** ::= true if region R equals region S
- **R.rank(i)** ::= projection of region R on dimension i (a one-dimensional region)
- **R.rank(i).low()** ::= lower bound of i^{th} dimension of region R
- **R.rank(i).high()** ::= upper bound of i^{th} dimension of region R
- **R.ordinal(P)** ::= ordinal value of point P in region R
- **R.coord(N)** ::= point in region R with ordinal value = N
- **R1 && R2** ::= region intersection (will be rectangular if R1 and R2 are rectangular)
- **R1 || R2** ::= union of regions R1 and R2 (may not be rectangular)
- **R1 – R2** ::= region difference (may not be rectangular)

Example: region (TutRegion1)

```
public class TutRegion {
    public static void main(String[] args) {
        region R1 = [1:10, -100:100];
        System.out.println("R1 = " + R1 + " ; R1.rank = " +
   R1.rank + " ; R1.size() = " + R1.size() + " ;
   R1.ordinal([10,100]) = " + R1.ordinal([10,100]));
        region R2 = [1:10,90:100];
        System.out.println("R2 = " + R2 + " ; R1.contains(R2) =
   " + R1.contains(R2) + " ; R2.rank(1).low() = " +
   R2.rank(1).low() + " ; R2.coord(0) = " + R2.coord(0));
    }
}
```

Console output:

```
R1 = {1:10,-100:100} ; R1.rank = 2 ; R1.size() = 2010 ;
  R1.ordinal([10,100]) = 2009
R2 = {1:10,90:100} ; R1.contains(R2) = true ;
  R2.rank(1).low() = 90 ; R2.coord(0) = [1,90]
```

Syntax extensions for regions

Region constructors

```
int hi, lo;
region r = hi;
     → region r = region.factory.region(0, hi)
region r = [low:hi]
     → region r = region.factory.region(lo, hi)

region r1, r2;  // 1-dim regions
region r = [r1, r2]
     → region r = region.factory.region(r1, r2);
       // 2-dim region
```

X10 arrays

- Java arrays are one-dimensional and local
  - e.g., array args in main(String[] args)
  - Multi-dimensional arrays are represented as “arrays of arrays” in Java
- X10 has true multi-dimensional arrays (as Fortran) that can be distributed (as in UPC, Co-Array Fortran, ZPL, Chapel, etc.)

Array declaration

- **T [.] A** declares an X10 array with element type T
- An array variable can refer to arrays with different rank

Array allocation

- **new T [R]** creates a local rectangular X10 array with rectangular region R as the index domain and T as the element (range) type
- e.g., int[.] A = new int[[0:N+1, 0:N+1]];

Array initialization

- elaborate on a slide that follows...

Array declaration syntax: [] vs [.]

General arrays: <Type>[.]

- one or multidimensional arrays
- can be distributed
- arbitrary region

Special case (“rail”): <Type>[]

- 1 dimensional
- 0-based, rectangular array
- not distributed
- can be used in place of general arrays
- supports compile-time optimization

Array of arrays (“jagged array”): <Type>[.][.]

Simple array operations

- **A.rank** ::= # dimensions in array
- **A.region** ::= index region (domain) of array
- **A.distribution** ::= distribution of array A
- **A[P]** ::= element at point P, where P belongs to A.region
- **A | R** ::= restriction of array onto region R
  - Useful for extracting subarrays

Aggregate array operations

- **A.sum(), A.max()** ::= sum/max of elements in array
- **A1 <*op*> A2**
  - returns result of applying a pointwise op on array elements, when A1.region = A2. region
  - <*op*> can include +, -, *, and /
- **A1 || A2** ::= disjoint union of arrays A1 and A2 (A1.region and A2.region must be disjoint)
- **A1.overlay(A2)**
  - returns an array with region, A1.region || A2.region, with element value A2[P] for all points P in A2.region and A1[P] otherwise.

Future work: framework for array operators

Example: arrays (TutArray1)

```
public class TutArray1 {
    public static void main(String[] args) {
        int[.] A = new int[ [1:10,1:10] ]
                   (point [i,j]) { return i+j;} ;
        System.out.println("A.rank = " + A.rank +
                           " ; A.region = " + A.region);
        int[.] B = A | [1:5,1:5];
        System.out.println("B.max() = " + B.max());
    }
}
```

array copy

Console output:

```
A.rank = 2 ; A.region = {1:10,1:10}
B.max() = 10
```

Initialization of mutable arrays

Mutable array with nullable references to mutable' objects:

```
RefType nullable [] farr = new RefType[N];  // init with null value
```

Mutable array with references to mutable objects:

```
RefType [] farr = new RefType [N];  // compile-time error, init required

dist d = dist.factory.block(N);
RefType [.] farr = new RefType [d] (point[i]) { return RefType(here, i); }
```

Execution of initializer is implicitly parallel / distributed (pointwise operation):

That hold 'reference to value objects' (value object can be inlined)

```
int [] iarr = new int[N] ;  // init with default value, 0
int [] iarr = new int[] {1, 2, 3, 4};  // Java style
int [] iarr = new int[N] (point[i])
                  {return i}; // explicit init
```

Initialization of value arrays

Initialization of value arrays requires an initializer.

Value array of reference to mutable objects:

```
RefType value [] farr = new value RefType [N];
                    // compile-time error, init required
RefType value [] farr = new value RefType [N] (point[i])
                                  { return new Foo(); }
```

Value array of ‘reference to value objects’ (value object can be inlined)

```
int value [] iarr = new value int[] {1, 2, 3, 4};
                                  // Java style init

int value [] iarr = new value int[N] (point[i])
                                     { return i };
                                     // explicit init
```

Tutorial outline

1) X10 in a nutshell

2) Sequential X10
- Type system
- Standard library

3) Concurrency in X10
- Activities
- Atomic blocks
- Clocks, clocked variables

4) X10 arrays
- Points
- Regions

5) Distributed X10
- Places
- Distributions
- Distributed arrays

6) Further examples

Distributed X10

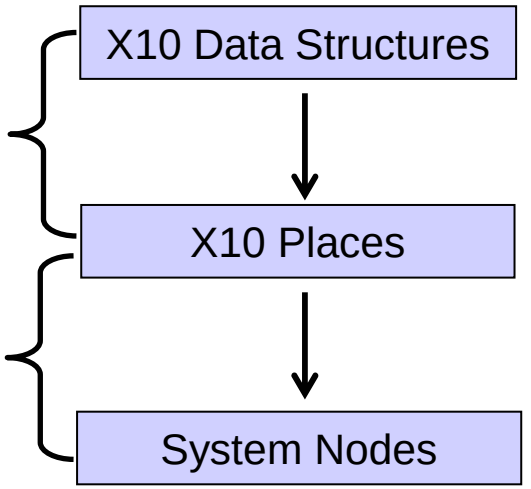
- **Places**
- **Locality rule**
- **Distributions**
- **async, futures**
- **ateach**
- **Distributed arrays**

Places in X10

- **place.MAX_PLACES** = total number of places (runtime constant)
- **place.places** = value array of all places in an X10
- **place.factory.place(i)** = place corresponding to index i
- **here** = place in which current activity is executing
- **<place-expr>.toString()** returns a string of the form “place(id=99)”
- **<place-expr>.id** returns the id of the place

X10 language defines mapping from X10 objects to X10 places, and abstract performance metrics on places

Future X10 deployment system will define mapping from X10 places to system nodes; not supported in current implementation



Locality rule

Any access to a mutable (shared heap) datum must be performed by an activity located at the place as the datum.

→ direct access via a remote heap reference is not permitted.

→Inter-place data accesses can only be performed by creating remote activities (with weaker ordering guarantees than intra-place data accesses)

→**BadPlaceException** is thrown if the locality rule is violated.

async and future with explicit place specifier

async (P) S

- Creates new activity to execute statement S at place P
- **async S** is equivalent to **async (here) S**

future (P) { E }

- Create new activity to evaluate expression E at place P
- **future { E }** is equivalent to **future (here) { E }**

Note that **here** in a child activity for an async/future computation will refer to the place P at which the child activity is executing, not the place where the parent activity is executing

Specify the destination place for async/future activities so as to obey the Locality rule e.g.,

```
async (O.location) O.x = 1;
future<int> F = future (A.distribution[i]) { A[i] } ;
```

Implicit syntax

- **Use conventional syntax for operations on values of remote type:**

- `x.f = e //write x.f of type T`

  ➔
```
final T v = e;
finish async(x.loc) {
   x.f=v;
}
```

- `… = …x.f …//read x.f of type T`

  ➔

  `future<T>(x.loc){x.f}.force()`

- **Similarly for array reads and writes.**

- **Invoke a method synchronously on values of remote type**

- `e.m(e1,…,en);`

  ➔
```
final T  v  = e;
final T1 v1 = e1;
…
final Tn vn = en;
finish async (v.loc) {
  v.m(v1,…,vn);
}
```

- **Similarly for methods returning values.**

Inter-place communication using async and future

Question: how to assign A[i] = B[j], when A[i] and B[j] may be in different places?

Answer #1: Use nested async:

```
finish async ( B.distribution[j] ) {
  final int bb = B[j];
  async ( A.distribution[i] ) A[i] = bb;
}
```

Answer #2: Use future-force and an async:

```
final int b = future (B.distribution[j])
        { B[j] }.force();
finish async ( A.distribution[i] ) A[i] = b;
```

ateach (distributed parallel iteration)

ateach (*FormalParam*: *Expr*) *Stmt*

ateach (point p:D) S

- Creates |D| async statements in parallel at place specified by distribution.

```
ateach (point p:D) S
```

```
for (point p:D.region)
  async (D[p]) { S }
```

- Termination of all (recursively created) activities with **finish**.
- **ateach** is a convenient construct for writing parallel matrix code that is independent of the underlying distribution, e.g.,

```
ateach ( point p : A.distribution )
  A[p] = f(B[p], C[p], D[p]) ;
```

- SPMD computation:

```
finish ateach( point[i] : dist.factory.unique() ) S
```

Example: ateach (TutAteach1)

```
public class TutAteach1 {
    public static void main(String args[]) {
        finish ateach (point p: dist.factory.unique()) {
            System.out.println("Hello from " + here.id);
        }
    } // main()
}
```

unique distribution: maps point i in region [0 : place.MAX_PLACES-1] to place place.factory.place(i).

Console output:

```
Hello from 1
Hello from 0
Hello from 3
Hello from 4
```

Distributions in X10

A **distribution** maps every point in a region to a place.

Creating distributions (x10.lang.dist):

- **dist D1 = dist.factory.constant(R, here); //** local distribution – maps region R to here
- **dist D2 = dist.factory.block(R);** // blocked distribution
- **dist D3 = dist.factory.cyclic(R);** // cyclic distribution
- **dist D4 = dist.factory.unique();** // identity map on [0:MAX_PLACES-1]

Using distributions

D[P] = place to which point P is mapped by distribution D

- if point p is in D.region
- otherwise **ArrayOutOfBoundException**

Allocate a distributed array e.g., T[.] A = new T[D];

- Allocates an array with index set = D.region, such that element A[P] is located at place D[P] for each point P in D.region
- NOTE: “new T[R]” for region R is equivalent to “new T[R->here]”

Iterating over a distribution – generalization of **foreach** to **ateach**

Operations on distributions

- **D.region** ::= source region of distribution
- **D.rank** ::= rank of D.region
- **D | R** ::= region restriction for distribution D and region R (returns a restricted distribution)
- **D | P** ::= place restriction for distribution D and place P (returns region mapped by D to place P)
- **D1 || D2** ::= union of distributions D1 and D2 (assumes that D1.region and D2.region are disjoint)
- **D1.overlay(D2)** ::= asymmetric union of D2 over D1
- **D.contains(p)** ::= true iff D.region contains point p
- **D1 – D2** ::= distribution difference: D1 | (D1.region – D2.region)

Syntax extensions for distributions

Constant distributions

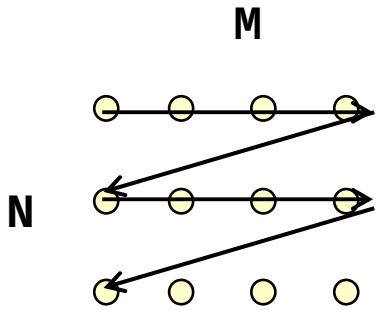
```
region r = [0:N];
dist d = r->here
    → dist d = dist.factory.constant(r, here);
dist d = 1000->here
    → dist d = dist.factory.constant([0,1000],
 here);
```

Distributions are implicitly converted to regions

```
for (point [i,j]: d) {...}
      → for (point [i,j]: d.region) {...}
```

Multidimensional arrays

```
double[.] darr = new double[[0:N, 0:M]->here];
for (point [i,j]: darr.region)
    darr[i,j] = ..;
```

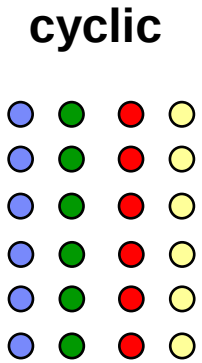


- **initial values in darr are `0.0`**
- **Iteration schema**
  - ‘lexicographical order’ (standard, fix)
  - **`[0,0], [0,1], [0,2], ...`**
- **Storage layout**
  - row major (fix)
  - spatial access locality with standard iteration schema

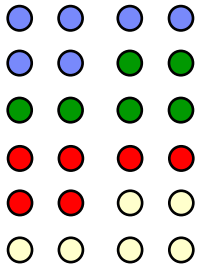
Distributed multidimensional arrays

```
dist cyclic = dist.factory.cyclic([0:4, 0:6])
dist blockcyclic = dist.factory.blockCyclic([0:4, 0:6], 6)
double[.] darr = new double[XXX];
```

assuming 4 places

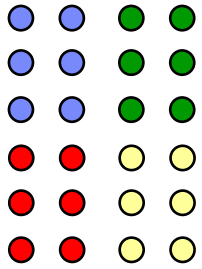


cyclic



block cyclic

for 1D arrays: cf. UPC



tiled

Future work: hierarchically tiled regions

Example: RandomAccess (1/2)

```
   dist D = dist.factory.block(TABLE_SIZE);
(1) final long[.] table = new long[D] (point [i]) { return i; }
(2) final long[.] RanStarts = new  long[dist.factory.unique()]
       (point [i]) { return starts(i);};
(3) final long value [.] SmallTable =  new long value[TABLE_SIZE]
       (point [i]) { return i*S_TABLE_INIT; };

(4) finish ateach (point [i] : RanStarts ) {
       long ran = nextRandom(RanStarts[i]);
       for (int count: 1:N_UPDATES_PER_PLACE) {
          int J = f(ran);
          long K = SmallTable[g(ran)];
          async (table.distribution[J]) atomic table[J] ^= K;
          ran = nextRandom(ran);
       }
    }
    assert(table.sum() == EXPECTED_RESULT);
```

Example: RandomAccess (2/2)

(1) Allocate and initialize table as a block-distributed array.

(2) Allocate and initialize **`RanStarts`** with one random number seed for each place.

(3) Allocate a small immutable table that can be copied to all places.

(4) Everywhere in parallel, repeatedly generate random table indices and atomically read/modify/write table element.

JGF Monte Carlo benchmark -- Sequential

```
double[] expectedReturnRate =
    new double[nRunsMC];
...
final ToInitAllTasks t =
   (ToInitAllTasks) initAllTasks;
for
   (point [i]: expectedReturnRate) {
   PriceStock ps = new PriceStock();
   ps.setInitAllTasks(t);
   ps.setTask(tasks[i]);
   ps.run();
   ToResult r =
         (ToResult) ps.getResult();
   expectedReturnRate[i] =
         r.get_expectedReturnRate();
   volatility[i] =
         r.get_volatility();
}
```

A task array (of size **nRunsMC**) is initialized with **ToTask** instances at each index.

Task:

- Simulate stock trajectory,
- Compute expected rate of return and volatility,
- Report average expected rate of return and volatility.

JGF Monte Carlo benchmark -- Parallel

```
double[] expectedReturnRate =
    new double[nRunsMC];
...
final ToInitAllTasks t =
   (ToInitAllTasks) initAllTasks;
finish foreach
   (point [i]:expectedReturnRate) {
   PriceStock ps = new PriceStock();
   ps.setInitAllTasks(t);
   ps.setTask(tasks[i]);
   ps.run();
   ToResult r =
         (ToResult) ps.getResult();
   expectedReturnRate[i] =
         r.get_expectedReturnRate();
   volatility[i] =
         r.get_volatility();
}
```

JGF Monte Carlo benchmark -- Distributed

```
dist D = dist.factory.block([0:(nRunsMC-1)]);
double[.] expectedReturnRate = new double[D];...

final ToInitAllTasks t =
   (ToInitAllTasks) initAllTasks;
finish ateach
   (point [i]:expectedReturnRate) {
   PriceStock ps = new PriceStock();
   ps.setInitAllTasks(t);
   ps.setTask(tasks[i]);
   ps.run();
   ToResult r =
         (ToResult) ps.getResult();
   expectedReturnRate[i] =
         r.get_expectedReturnRate();
   volatility[i] =
         r.get_volatility();
}
```

Tutorial outline

1) X10 in a nutshell

2) Sequential X10
- Type system
- Standard library

3) Concurrency in X10
- Activities
- Atomic blocks
- Clocks, clocked variables

4) X10 arrays
- Points
- Regions

5) Distributed X10
- Places
- Distributions
- Distributed arrays

6) Further examples

Cellular Automata Simulation: Game of Life

Acknowledgment:

“Barriers”, Chapter 5.5.4, Java Concurrency in Practice, Brian Goetz et al

Game of Life – Java version (1 of 2)

java.util.concurrent version (Listing 5.15, p102, JCiP)

```
public class CellularAutomata {
  private final Board mainBoard;
  private final CyclicBarrier barrier;
  private final Worker[] workers;

  public CellularAutomata(Board board) {
    this.mainBoard = board;
    int count = Runtime.getRuntime().availableProcessors();
    this.barrier = new CyclicBarrier(count,
                         new Runnable() { // barrier action
                           public void run(){mainBoard.commitNewValues();}});
    this.workers = new Worker[count];
    for (int i = 0; i < count; i++)
      workers[i] = new Worker(mainBoard.getSubBoard(count, i));
  } // constructor

  public void start() {
    for (int i = 0; i < workers.length; i++) new Thread(workers[i]).start();
    mainBoard.waitForConvergence();
  } // start()
} // CellularAutomata
```

Game of Life – Java version (2 of 2)

```
private class Worker implements Runnable {
  private final Board board;
  public Worker(Board board) { this.board = board; }

  public void run() {
    while (!board.hasConverged()) {
      for (int x = 0; x < board.getMaxX(); x++)
        for (int y = 0; y < board.getMaxY(); y++)
          board.setNewValue(x, y, computeValue(x, y));
      try { barrier.await(); }
      catch (InterruptedException ex) { return; }
      catch (BrokenBarrierException ex) { return; }
    } // while
  } // run()

  private int computeValue(int x, int y) {
    // Compute the new value that goes in (x,y)
    . . .
  }
} // Worker
```

Game of Life – X10 version

```
public class CellularAutomata {
  private final Cell[.] mainBoard1, mainBoard2;
  public CellularAutomata(Cell[.] board) {
    mainBoard1 = board; mainBoard2 = null;
  } // constructor

  public void start() {
    finish async {
      final clock barrier = clock.factory.clock();
      ateach ( point[i] : dist.unique() ) clocked(barrier) {
        boolean red = true;
        while ( !subBoardHasConverged(mainBoard1,mainBoard2,red) ) {
          for ( point[x,y] : mainBoard1 | here )
            if ( red ) mainBoard2[x,y] = computeValue(mainBoard1, x, y);
            else mainBoard1[x,y] = computeValue(mainBoard2, x, y);
          next;
          red = ! red;
        } // while
      } // foreach
      if (! red) mainBoard1 = mainBoard2; // answer is now in mainBoard1
    } // finish async
    // All boards have now converged
  } // start()
} // CellularAutomata
```

Game of Life – X10 version

```
public class CellularAutomata {
  private final Cell[.] mainBoard1, mainBoard2;
  public CellularAutomata(Cell[.] board) {
    mainBoard1 = board; mainBoard2 = null;
  } // constructor

  public void start() {
    finish async {
      final clock barrier = clock.factory.clock();
      ateach ( point[i] : dist.unique() ) clocked(barrier) {
        boolean red = true;
        while ( !subBoardHasConverged(mainBoard1,mainBoard2,red) ) {
          for ( point[x,y] : mainBoard1 | here )
            if ( red ) mainBoard2[x,y] = computeValue(mainBoard1, x, y);
            else mainBoard1[x,y] = comput
          next;
          red = ! red;
        } // while
      } // foreach
      if (! red) mainBoard1 = mainBoard2; // answer is now in mainBoard1
    } // finish async
    // All boards have now converged
  } // start()
} // CellularAutomata
```

Example of transmitting clock from parent to child

NOTE: exiting from while loop terminates activity for iteration i, and automatically deregisters activity from clock

Memoization

Acknowledgment:

“Memoization”, Chapter 5.6, Java Concurrency in Practice, Brian Goetz et al

Memoization in Java

```
public class Memoizer<A,V> implements Computable<A,V> {
 private final ConcurrentMap<A,Future<V>> cache
     = new ConcurrentHashMap<A, Future<V>>();
 private final Computable<A,V> c;
 public Memorizer(Computable<A,V> c) { this.c = c;}
 public V compute(final A arg) throws InterruptedException {
   while (true) {
      Future<V> f = cache.get(arg);
      if (f==null) {
        Callable<V> eval = new Callable<V>() {
            public V call() throws InterruptedException {
               return c.compute(arg);
            }
        };
        FutureTask<V> ft = new FutureTask<V>(eval);
        f = cache.putIfAbsent(arg, ft);
        if (f == null) { f = ft; ft.run();}
      }
      try {
        return f.get();
      } catch (CancellationException e) {
         cache.remove(arg,f);
      } catch (ExecutionException e) {
         throw launderThrowable(e.getCause());
      }}}}
```

Memoization

```
public class Memoizer implements Computable  {
  private final ConcurrentMap cache =
      new ConcurrentHashMap ();
  private final Computable c;

  public Memoizer(Computable c) { this.c = c;}

  public Object compute (final Object arg) throws Exception
  {
    nullable<Future> f = (Future) cache.get(arg);
    if (f == null) {
      Future g = new Latch(c, arg);
      f = cache.putIfAbsent(arg, g);
      if (f==null) { f=g; f.run();}
    }
     return f.force();
  }

}
```

Memoization (with proposed generics)

```
public class Memoizer<V,A> implements Computable<V,A> {
  private final ConcurrentMap<future<V>,A> cache =
      new ConcurrentHashMap<future<V>,A>();
  private final Computable<V,A> c;

  public Memoizer(Computable<V,A> c) { this.c = c;}

  public V compute (final A arg) throws Exception {
    nullable<future<V>> f = cache.get(arg);
    if (f == null) {
      future<V> g = new Latch(c, arg);
      f = cache.putIfAbsent(arg, g);
      if (f==null) { f=g; f.run();}
    }
     return f.force();
  }
}
```

Event Handling and Concurrency: GUI Applications as an Exemplar

Acknowledgment:

“GUI Applications”, Chapter 9, Java Concurrency in Practice, Brian Goetz et al

Scenario: Thread Hopping in a GUI Application (Java)

java.util.concurrent version (Listing 9.5, p196, JCiP)

```
private void longRunningTaskWithFeedback() {
  button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      button.setEnabled(false); label.setText("busy"); // 1) Dim button
      exec.execute( // 2) Submit long-running task for execution
        new Runnable() {
          public void run() {
            try {
              /* Do big computation */
            } finally {
               // 3) Submit task to run in GUI even thread executor
              GuiExecutor.instance().execute(new Runnable() {
                public void run() {
                  button.setEnabled(true); label.setText("idle");
                }
              });
            }
          } // run()
        });
      } // run()
    });
  }
```

Scenario: Thread Hopping in a GUI Application (X10)

```
private void longRunningTaskWithFeedback() {
  button.setEnabled(false); label.setText("busy"); // 1) Dim button
  async (ExecPlace) {// 2) Create long-running task at ExecPlace
     /* Do big computation */
     // 3) When done, create task at GuiExecutorPlace
     async (GuiExecutorPlace) {
       button.setEnabled(true);
       label.setText("idle");
     }
  }
}
```

Swing utility	X10 idiom
SwingUtilities.isEventDispatchThread()	here == GuiExecutorPlace
SwingUtilities.invokeLater()	async (GuiExecutorPlace)
SwingUtilities.invokeAndWait()	finish async (GuiExecutorPlace)

Single-threaded vs. Multi-threaded GUI frameworks

1) **Java approach** -- *Single-threaded* GUI framework
   - GUI objects are kept consistent by thread confinement
   - Pro: Programmer does not have to worry about deadlock in GUI thread
   - Cons:
     - Cannot exploit parallelism to speed up GUI framework
     - Reasoning about data accesses across task boundaries can still be tricky due to nondeterminism of task scheduling

2) **X10 approach** – *Single-place Multi-threaded* GUI framework a
   - All GUI tasks are scheduled at GuiExecutorPlace -- GUI objects are accessed only by activities in GuiExecutorPlace
   - Pro: Can easily exploit parallelism within GuiExecutorPlace
   - Con: atomic blocks necessary to ensure mutual exclusion among tasks (but making atomicity explicit should also make the code more maintainable?)
   - See next slide on how to address overhead of atomic blocks in a Single-place Multi-threaded GUI framework

Performance Implications (Discussion)

- Use of atomic blocks can introduce additional overhead in X10 implementation, compared to single-threaded Java version
  - For multi-core architectures, this additional overhead should be more than compensated for by performance improvements due to concurrency …

- … but if there is a real need for improving the performance of GuiExecutorPlace for execution on a *single thread* …
  - Restrict GuiExecutorPlace to be a *local nonblocking place*
    - only local nonblocking activities are permitted to run at such a place
  - *nonblocking* ➔ no static occurrence of when, force(), next() permitted (but finish is permitted)
  - *local* ➔ all data accessed is statically guaranteed to be *place-local*
  - X10 runtime can use a *single active worker thread* for GuiExecutorPlace and guarantee absence of interleaving among tasks at GuiExecutorPlace

    ➔ atomic-enter and atomic-exit can then be replaced by no-ops

Distributed Containers

- **DistributedHashMap**

Adaptation of ConcurrentHashMap by Doug Lea for X10.

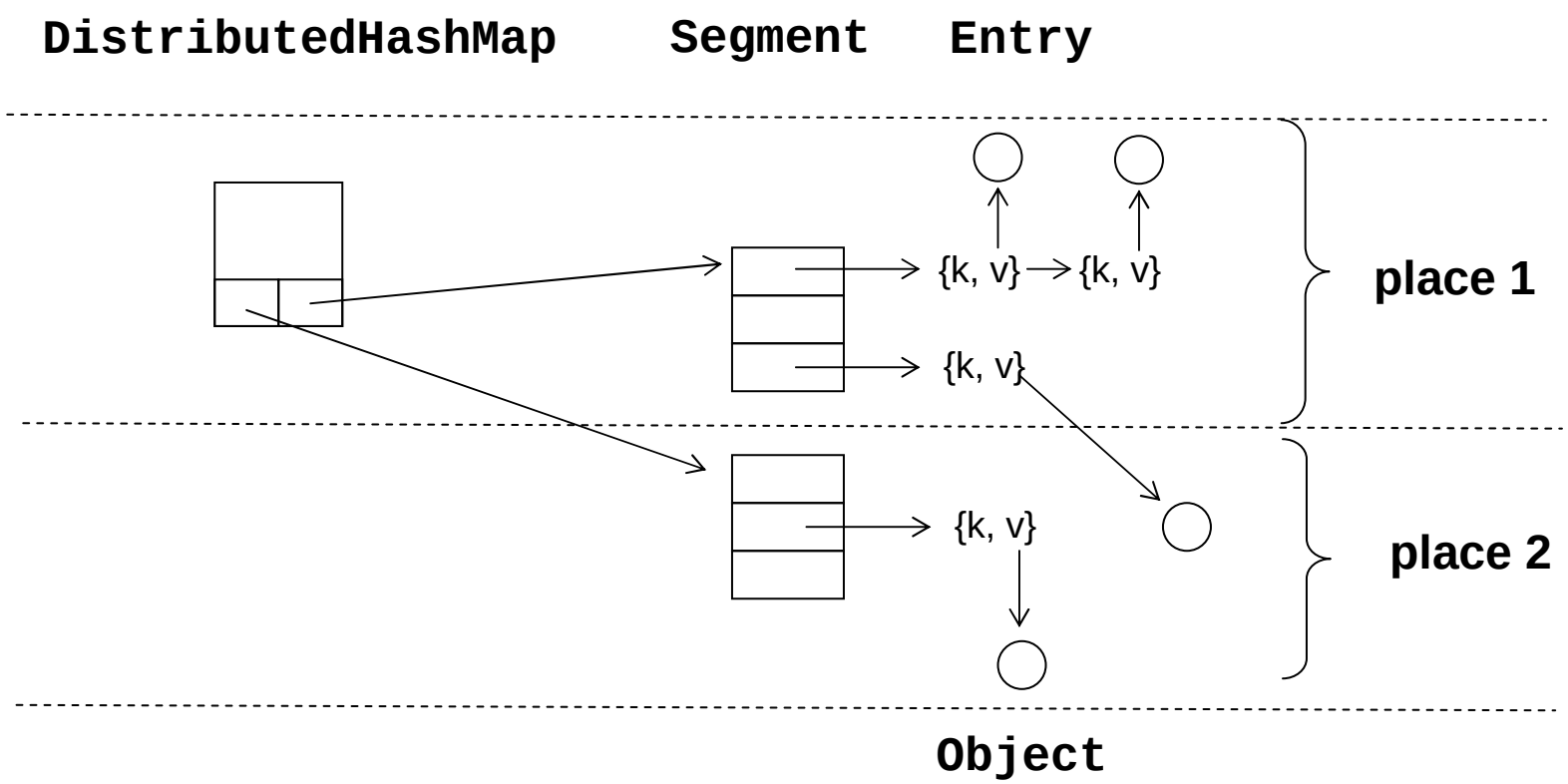
DistributedHashMap

- **Keys**
  - immutable objects (instances of value classes)
  - hashing of entries according to keys across places
- **Values**
  - references to mutable objects

Design goals

- Distribution of Key-Value pairs
- Thread-safety
- Operations are linearizable
- Internal concurrency for optimization

DistributedHashMap - design



DistributedHashMap - data structures

```
class DistributedHashMap {
   Segment[] segments;

   Segment segmentFor(final int hash) { ... }
   int hash(final Object x) { ... }
}
class Segment {
   final int index;
   int count;
   int modCount;
   Entry[] table;
   public final Semaphore sem;
...
}
class Entry {
   final value key;
   final int hash;
   Object value;
   final nullable<Entry> next;
}
```

references to segments in different places (unique distribution)

index in Segments[]

for consistency among concurrent readers and writers

to detect ABA violation

mutual exclusion among writers / fallback for global operations

key is an instance of a value type

DistributedHashMap - operations

Selected operations

- **`boolean containsValue(final value key)`**
  - must not suffer from aba problem
  - optimization: internal concurrency across places
  - reader concurrency
- **`nullable<Object> put(final value key, final Object value)`**
  - concurrent across places, sequential in each place
- **`nullable<Object> get(final value key)`**
  - concurrent intra and inter-place read access
- others that we do not discuss here

DistributedHashMap – aba problem

Linearizability requires that ABA problem cannot occur:

```
// initially {k1, v} is in the table

// thread 1                  // thread2
table.put(k2, v);            r = table.containsValue(v);
table.remove(k1, v);
```

- ABA problem: thread 2 must not observe r == false;
  (could happen if k1, k2 target different segments and operations in both thread occur concurrently)
- Problem can occur whenever Hashtable is traversed
  (operations isEmpty, size, containsValue)

→ Prevention of ABA complicates implementation significantly
→ Modification counters

DistributedHashMap – get

```
class DistributedHashMap ...

  nullable<Object> get(final Object key) {
     final int hash = hash(key); // throws NullPointerException if key null
     final Segment segmentfor = segmentFor(hash);
     return segmentfor.get(key, hash);
  }
class Segment ...

  nullable<Object> get(final Object key, final int hash) {
      atomic if (count==0) return
      int hashIndex = indexFor(hash, index);
      nullable<Entry> first = table[hashIndex];
      nullable<Entry> e = first;
      for (e = first; e !=null; e =e.next)
          if (e.hash == hash && e.key == key) {
             Object value = e.value;
             if (value !=null) return value;
             break;
          }
      // Recheck under synch if key apparently not there or interference
      Segment seg = segments[hash & SEGMENT_MASK];
      sem.p();
      try{
        Entry newFirst = table[index];
        if (e != null || first != newFirst) {
          for (e = newFirst; e != null; e = e.next) {
            if (e.hash == hash && eq(key, e.key))
              return e.value;
          }
        }
        return null;
      } finally { sem.v();}
  }
```

atomic, to reliably communicate with put.

DistributedHashMap – put

```
class DistributedHashMap ...
   nullable<Object> put(final Object key, final Object value) {
      int hash = hash(key);
      Segment segmentfor = segmentFor(hash);
      return segmentfor.put(key, hash, value);
   }
class Segment ...
   nullable<Object> put(final Object key, final int hash, final Object value) {
      nullable<Object> oldval = null;
      sem.p();
      try {
        nullable<Entry> first = table[indexFor(hash,index)];
        nullable<Entry> e = first;
        while (e != null) {
           if (e.hash == hash && key == e.key)
              break;
           e = e.next;
        }

        if (e != null) {
           oldval = e.value;
           atomic { e.value = value; }
        } else {
           modCount ++;
           table[index] = new Entry(key, hash, value, first);
           atomic { count ++; }
        }
      } finally { sem.v(); }
      return oldval;
   }
```

acquire lock – exclusive put per segment, sync with concurrent put.

comparison of values with operator ==

atomic write means release (sync with concurrent get)

atomic read + write means acquire-release sync with concurrent get

release lock, sync with concurrent put.

DistributedHashMap – containsValue (1/2)

```
class Segment ...
  boolean containsValue(final Object value) {

    final int[.] mc = new int[segments.distribution];
    final boolean[.] vals = new boolean[segments.distribution];
    ---------------------------------------------------------
    // try without locking
    finish ateach (point p:segments) {
       atomic {
           mc[p] = segments[p].modCount;
           vals[p] = segments[p].containsValue(value);
       }
    }
    if (vals.or())
       return true;
    finish ateach (point p:segments) {
       mc[p] -= segments[p].modCount;
    }
    if (mc.sum() == 0)
       return false;
    ---------------------------------------------------------

    // resort to locking all segments ....
    for (point p:segments)
        finish async (segments.distribution[p]) { segments[p].sem.p(); }

    finish ateach (point p:segments) {
       vals[p] = segments[p].containsValue(value);
       segments[p].sem.v();
    }
    return vals.or();
  }
```

temporary distributed arrays

reduction

non-blocking

reduction

acquire all locks in order

blocking

search in parallel across segments

release locks in any order

reduction

Programming Technologies

DistributedHashMap – containsValue (1/2)

```
class DistributedHashMap ...
  boolean containsValue(final Object value) {

    final int[.] mc = new int[segments.distribution];
    final boolean[.] vals = new boolean[segments.distribution];
  ---------------------------------------------------------------
    // try without locking
    finish ateach (point p:segments) {
       atomic {
           mc[p] = segments[p].modCount;
           vals[p] = segments[p].containsValue(value);
       }
    }
    if (vals.or())
       return true;
    finish ateach (point p:segments) {
       mc[p] -= segments[p].modCount;
    }
    if (mc.sum() == 0)
       return false;
  ---------------------------------------------------------------

    // resort to locking all segments ....
    for (point p:segments)
        finish async (segments.distribution[p]) { segments[p].sem.p(); }

    finish ateach (point p:segments) {
       vals[p] = segments[p].containsValue(value);
       segments[p].sem.v();
    }
    return vals.or();
  }
```

temporary distributed arrays

reduction

non-blocking

reduction

acquire all locks in order

blocking

search in parallel across segments

release locks in any order

reduction

Programming Technologies

DistributedHashMap – containsValue (2/2)

```
class Segment ...

   boolean containsValue(final Object value) {
      atomic if (count == 0) return;
      for (point [p]: table) {
         nullable<Entry> e = table[p];
         while (e != null) {
            if (e.value.equals(value))
               return true;
            e = e.next;
         }
      }
      return false;
   }
```

atomic read means acquire sync with concurrent put.

Examples of Array Kernels

- **Jacobi**
- **Edminston**
- **NAS CG**

Jacobi 1d

```
class Jacobi  {
  public static final int N=100;
  public static final double epsilon=0.002;

  public static void main(String args[]) {
    region R = [0..N+1];
    distribution D = distribution.blocked(R);
```

Single threaded main loop, performing aggregate operations.

Built-in distribution

Subsequent code does not assume D is blocked.

```
    region R_inner = [1..N];
    distribution D_inner = D | R_inner;
    distribution D_boundary = D-D_inner;
    int iters = 0;
    double[D] a = (D_boundary 0.0) || new double[D_inner]
                { return Math.Random(); };
    while ( true ) {
      final double[D_inner] temp = new double[D_inner] (i) {
                future<double>low = future (a[i-1]) { a[i-1] };
                future<double>low = future (a[i+1]) { a[i+1] };
                return (low.force()] + high.force())/2.0;};
      double error = (reduce (Math.abs((a | D_inner)-
                              temp)).operator_'+'()),
      if ( error < epsilon )
         break;
      a = a.overlay( temp );
      iters++;
    }
    System.out.println("Number of iterations="+iters);
  }
}
```

Restriction to a region
Distribution difference

Lifting of <op> on base type to array type

Array initializer

Reduction operation

Restriction of array to a subdistribution

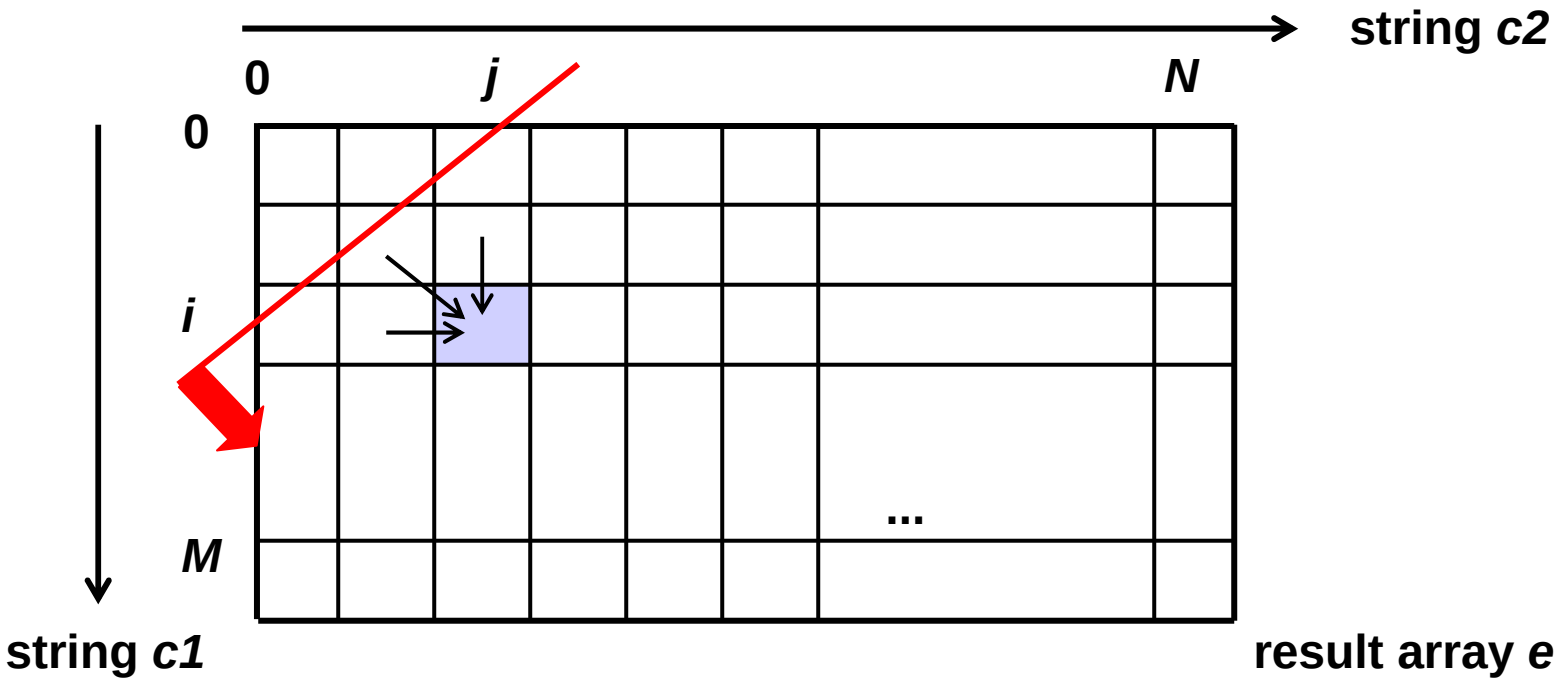
Updating one array with another.

Programming Technologies

Edmiston

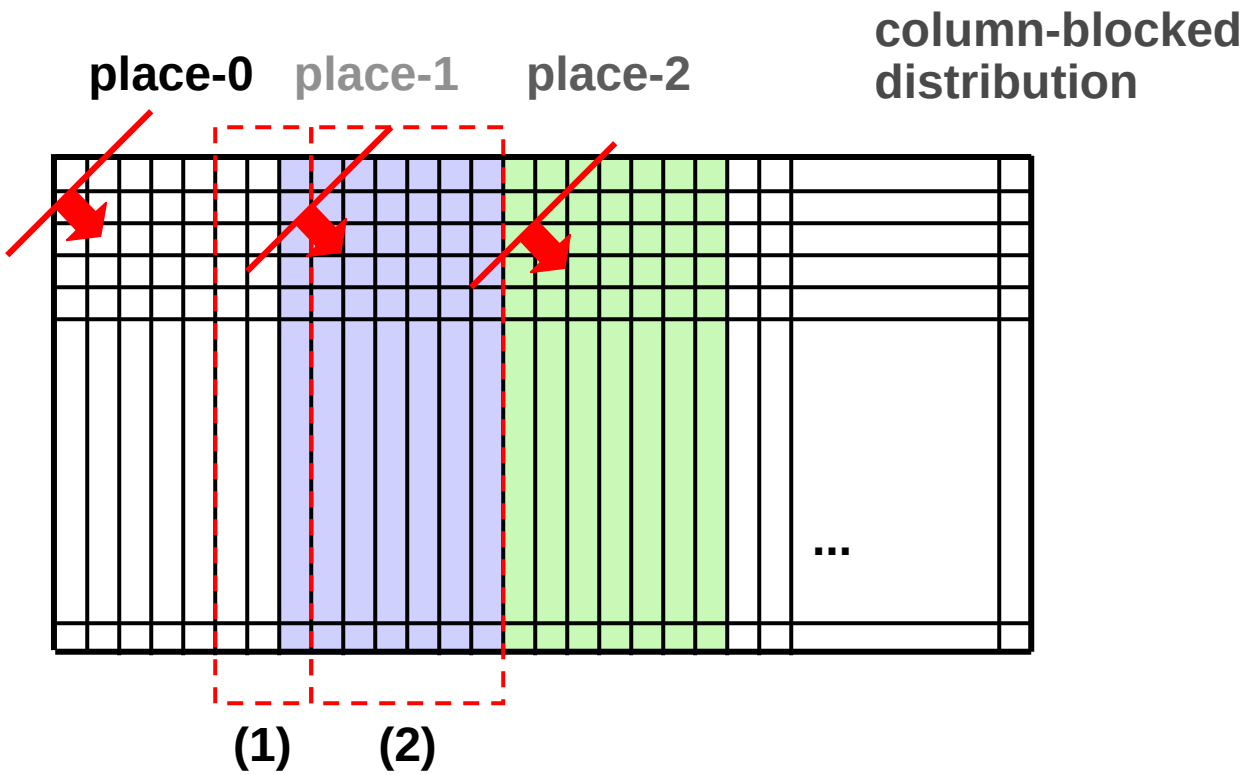
wavefront computation

Algorithm for gene sequence comparison




```
e[i, j] =  min (e[i-1,j]   + iGapPen,
                e[i,j-1]    + iGapPen,
                e[i-1,j-1] + (c1[i] == c2[j] ? iMatch : iMisMatch));
```

Edmiston - Parallelization



Computation in every place:

step (1): compute “warmup” in a place-local result array

step (2): compute results based on initial condition for step1 in result array

Edmiston

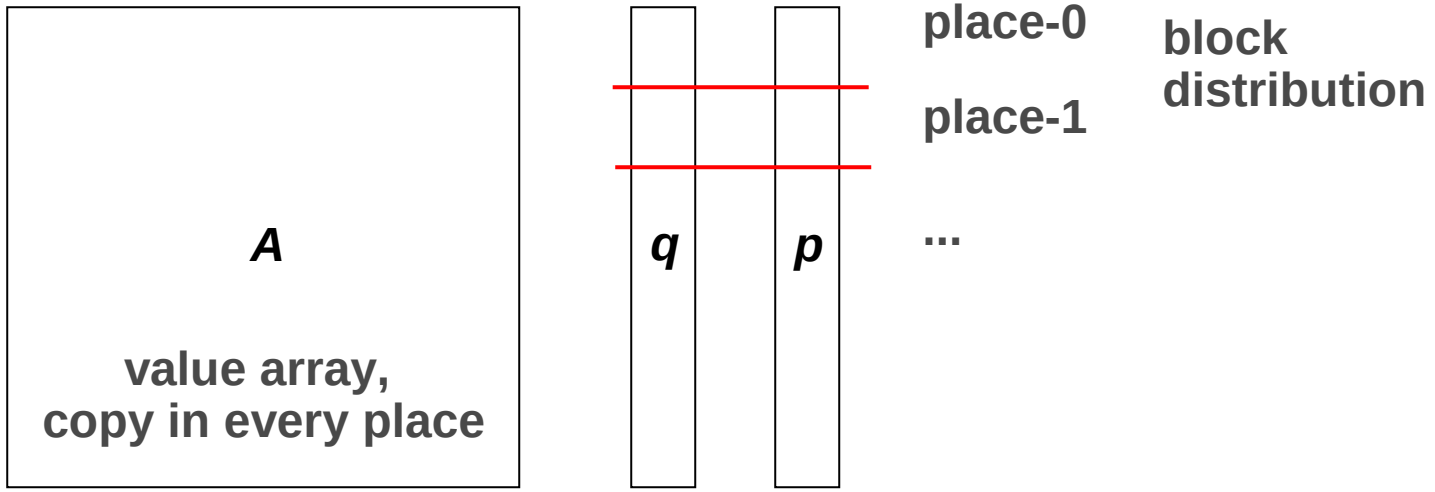
```
final RandCharStr c1, c2;
final int N = c1.s.length-1, int M = c2.s.length-1;
final dist D = columnBlocked([0:N],[0:M]);
final int[.] e = new int[D];

// SPMD computation at each place
finish ateach (point [p]:dist.factory.unique(D.places())) {
   // get sub-distribution for this place
   final dist myD = D|here;
   final int myLow = myD.region.rank(1).low();
   final int myHigh = myD.region.rank(1).high();
   final int overlapStart = Math.max(0,myLow-overlap);
   final dist warmupD = [0:N, overlapStart:myLow]->here;
   // create a local warmup array
   final int [.] W = new int[warmupD];
   // compute columns overlapStart+1 .. myLow using column overlapStart
   computeMatrix(W, c1, c2, overlapStart+1, myLow);                    (1)
   // copy column, e[0:N,myLow] = W[0:N,myLow];
   finish foreach (point [i] : [0:N]) e[i,myLow] = W[i,myLow];
   computeMatrix(e, c1, c2, myLow+1, myHigh);                          (2)
}

void computeMatrix(int[.] a, final RandCharStr c1,
                   final RandCharStr c2, int firstCol, int lastCol) {
   for (point[i,j] : [1:N,firstCol:lastCol] )
        a[i,j] = min4(0, a[i-1,j]+iGapPen, a[i,j-1]+iGapPen,
                  a[i-1,j-1] + (c1.s[i]==c2.s[j] ? iMatch : iMisMatch));
}
```

NPB – CG in X10

Sparse matrix-vector multiplication: *q* = *Ap*



- **square matrix: na x na**
- **non-zero elements: *nz***
- **sparse representation in column compressed format**
  - ***A [nz]***
  - ***A_colidx [nz]***
  - ***A_rowstr [na]***

NPB – CG in X10

```
dist THREADS = dist.factory.block([0:np-1]);
dist D = dist.factory.block([1:na]);
double[.] p = new double[D];
double[.] q = new double[D];
double[.] r = new double[D];
double[.] x = new double[D] (point [p]) { return 1.0; };
double[.] z = new double[D];

final double value [.] A_val = new value double[nz+1] {...};
final int value [.] A_colidx_val = new value int [nz+1] {...};
final int value [.] A_rowstr_val = new value int [na+2] {...};

for (point iter: [1:niter]) {
  finish ateach (point[p]: THREADS)
    { zero q, z, r and p, update rhomaster with square sum of x }
  double rho = rhomaster.sum();
  for (point it: [0:cgitmax]){
    // q = Ap  submatrix vector multiply
    finish ateach (point [it]: THREADS) {
      mvmult (q, p);
      dmaster[here.id]=(p[D|here]).mul(q[D|here]).sum();
    }
    final double rho0 = rho;
    final double alpha = rho / dmaster.sum();
    finish ateach (point [it]: THREADS)
    { z += alpha *p  r -= alpha*q; update rhomaster with square sum of x }
    rho = rhomaster.sum();
    final double beta = rho/rho0;
    finish ateach (point [it]:THREADS) { p = r+beta*p }
  }
```

sparse matrix

continues on next slide →

NPB – CG in X10

← continuation from previous slide

```
    // r = Az  submatrix vector multiply
    finish ateach (point [it]:THREADS) {
       mvmult (r, z);
       rnormmaster[here.id]=(x[D|here]).sub(r[D|here]).pow(2).sum();
    }
    // compute residual norm  ||r|| = ||x-Az||
    rnorm = Math.sqrt( rnormmaster.sum() );
    tnorm1 = x.mul(z).sum();
    tnorm2 = z.mul(z).sum();
    tnorm2 = 1.0 / Math.sqrt(tnorm2);
    zeta = shift + 1.0 / tnorm1;
    final double tnorm2ff = tnorm2;
    finish ateach (point[jj]: D) x[jj] = tnorm2ff*z[jj];
  }

  // q = Ap  submatrix vector multiply
  void mvmult(double[.] q, double[.] p) {
     region Dlocal = (D | here).region;
     for (point [j] : Dlocal) {
        double sum = 0.0;
        for (point [k] : [A_rowstr_val[j]:A_rowstr_val[j+1]-1]){
           int idx = A_colidx_val[k];
           future<double> tmp = future (p.distribution(p[idx]) {p[idx]};
           sum += A_val[k] * tmp.force();
        }
        q[j] = sum;
     }
  }
```

sparse matrix access