COMP 422, Lecture 12: Single-Place Programming in X10

Vivek Sarkar

Department of Computer Science
Rice University

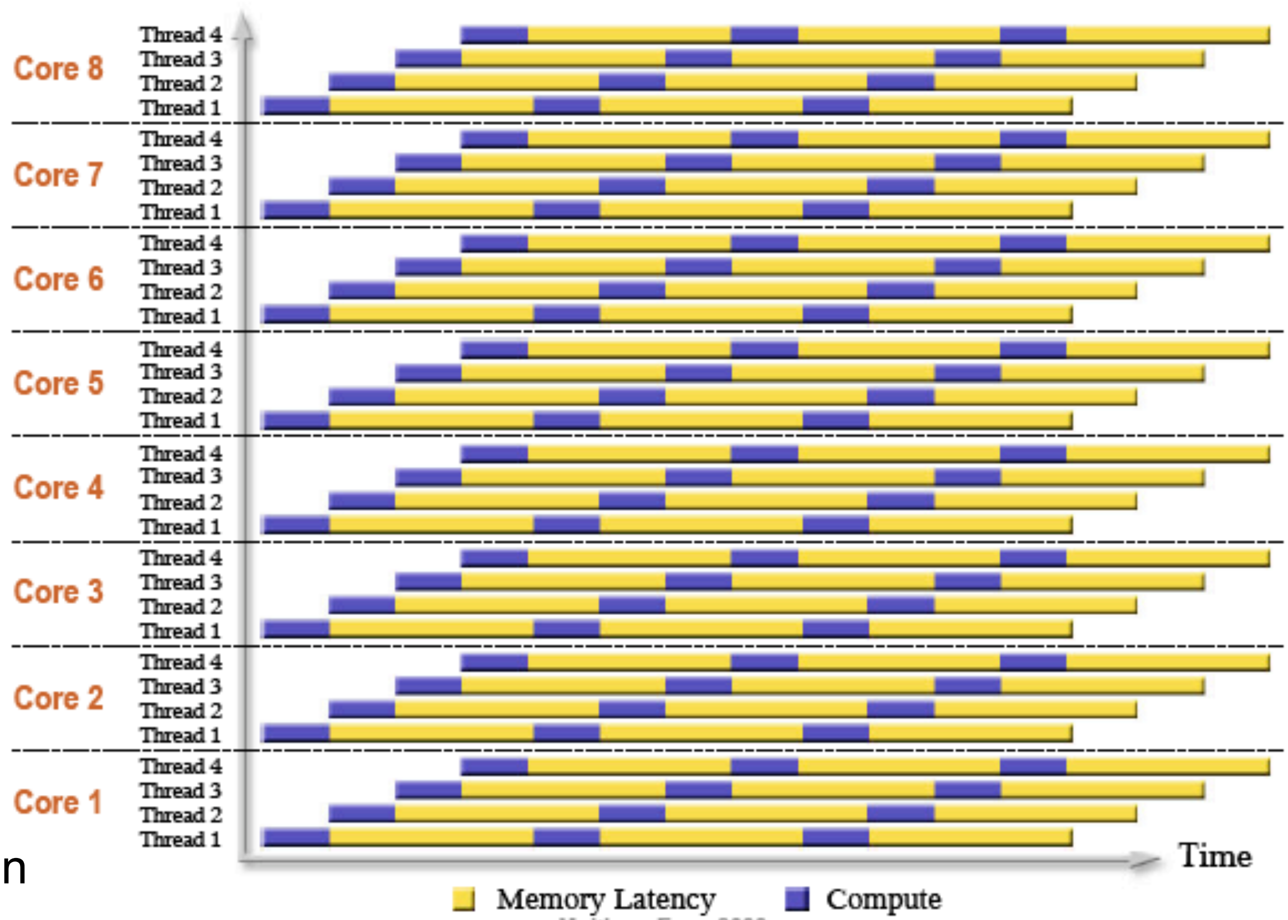
vsarkar@rice.edu

In-class Midterm Exam on 2/28/08

- Duration: 1 hour
- Weightage: 20%
- Three written questions to cover the following areas:
  - Performance models for parallel algorithms and machines
    - Sections 2.2, 2.3, 2.4, 2.5, 3.1, 3.2
  - Cilk
    - Cilk reference manual
  - OpenMP
    - Section 7.10, OpenMP 2.5 specification

Programmer's view of T2000 (yellowstone.rcsg.rice.edu)

- Chip with 8 cores, 4 threads per core
- Fine Grained Multi-threading – can read whole register file without penalty in context switch
- An 8 KB data cache and 16 KB instruction cache per core give an L1-hitrate of 90% or less. To support this a large 3MB, 12 way associative L2 cache shared among every core.



X10 Background

- Developed since 2004 as part of DARPA High Productivity Computing Systems (HPCS) program
  - — DARPA's goal: increase development productivity by 10x from 2002 to 2010
- Unified abstractions of asynchrony and concurrency for use in
  - — Multi-core SMP Parallelism (single place)
  - — Messaging and Cluster Parallelism (multiple places)
- Performance transparency – don't lock out the performance expert!
  - – Expert programmer should have controls to tune optimizations and tailor distributions & communications to actual deployment
- X10 programming model can be used to extend any sequential language --- we chose to build the X10 language on a sequential subset of Java
  - — Retain core values of Java --- productivity, portability, safety
  - — Target adoption by mainstream developers with Java/C/C++ skills
  - — Efficient foreign function interfaces for libraries written in Fortran and C/C++
- Reference: "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing", P.Charles et al, OOPSLA 2005 Onward! track.
- Open source project on SourceForge: x10.sf.net
- Acknowledgment: material for this lecture was taken from PLDI 2007 tutorial on X10 by V.Saraswat, V.Sarkar, N.Nystrom

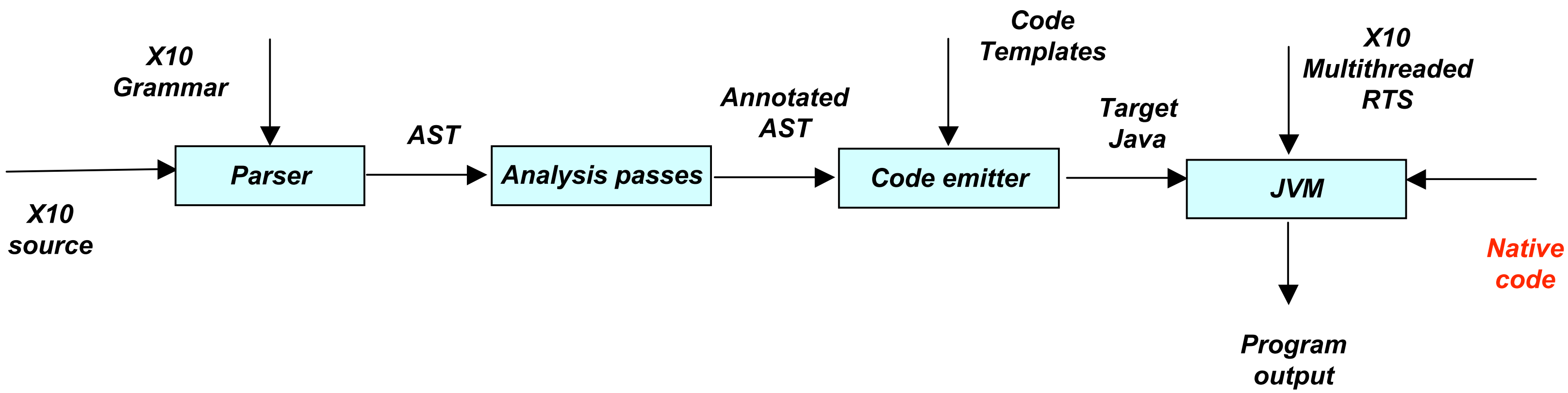
X10 availability

- X10 is an open source project (Eclipse Public License).
- Website: http://x10.sf.net
- Reference implementation in Java, runs on any Java 5 VM.
  - Windows/Intel, Linux/Intel
  - AIX/PPC, Linux/PPC
  - Runs on multiprocessors
- Website contains
  - Tutorial material
  - Presentations
  - Download instructions
  - Copies of some papers
  - Pointers to mailing list

X10 Compiler (06/2007)

09/03
PERCS Kickoff
02/04
X10 Kickoff
07/04
X10 0.32 Spec Draft
02/05
X10 Prototype #1
07/05
X10 Productivity Study
12/05
X10 Prototype #2
12/06
Open Source Release
6/07
Annotations, X10lib v1

Operational X10 implementation (since 02/2005)



Structure

- Translator based on Polyglot (Java compiler framework)
- X10 extensions are modular.
- Uses Jikes parser generator.

Code metrics

- Parser: ~45/14K*
- Translator: ~112/9K
- RTS: ~190/10K – revised for JUC
- Polyglot base: ~517/80K
- Approx 280 test cases.

(* classes+interfaces/LOC)

New features 6/07

- Annotations
- X10lib v 1

X10 Runtime (06/2007)

- **Implemented in Java**
- **Full implementation of X10 v1**
- **Concurrent implementation**
  - —Based on Java concurrency utils
  - —Thread pool used to manage activities in each place
  - —Asyncs implemented using shared work queue per place
  - —Atomics implemented with per place lock
- **Support for native code integration**
  - —Enables development of hybrid X10 applications: manage concurrency in X10, use existing single-threaded C/Fortran kernels.

Single-place X10: Java extensions

Stm:

async *[*clocked *ClockList] Stm*

atomic *Stm*

finish *Stm*

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

for (*i : Region*) *Stm*

foreach (*i : Region*) *Stm*

ateach (*I : Distribution*) *Stm*

MethodModifier:

atomic nonblocking sequential

Type:

nullable<*Type*>

future <*Type* >

x10.lang has the following classes (among others)

point, range, region, array, clock

Some of these are supported by special syntax.

Regions and Arrays

ArrayExpr:

- new ArrayType (Formal) { Stm }
- *ArrayExpr* [*Region*] -- Section
- *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*]

Kind :

value | reference

Region:

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

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

Comparison with Java ™

X10 language builds on the Java language

Shared underlying philosophy: shared syntactic and semantic tradition, simple, small, easy to use, efficiently implementable, 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**

async

async S

Stmt ::= async *Stmt*

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

cf Cilk's spawn

```
final int k = …;

async {
    … = f(k);
    p.x = … ;
}
```

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

Stmt ::=* finish *Stmt

cf Cilk's sync

```
try {
  finish {
    async foo();
    bar();
  }
}
catch ( … ) { … }
```

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)

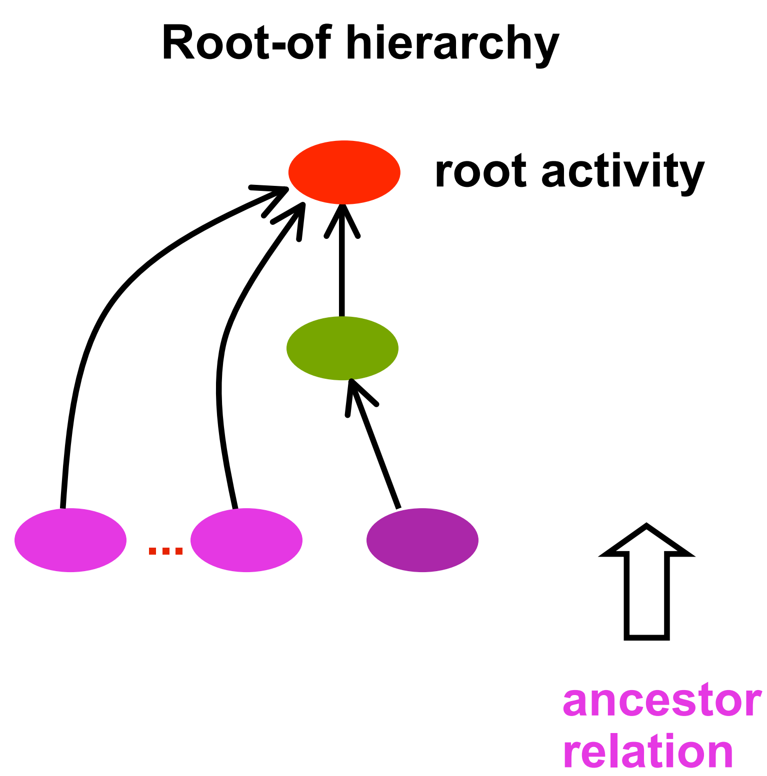
→ main function is **root activity**

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

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

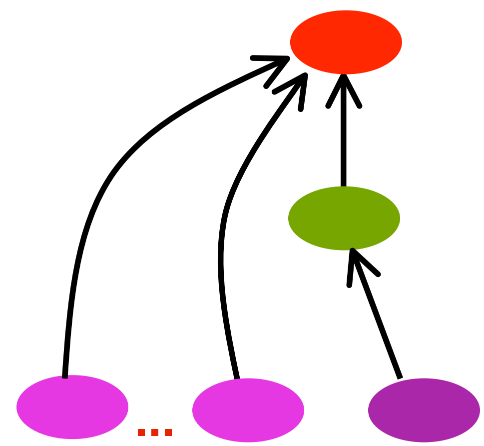
Rooted computation X10

```
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
}
```



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 {
     async {
       throw new Exception ("Hello world exception")
     }
     result = 42;
   } // finish
} catch (x10.lang.MultipleExceptions me) {
   System.out.print(me);
}
assert (result == 42); // always true
```

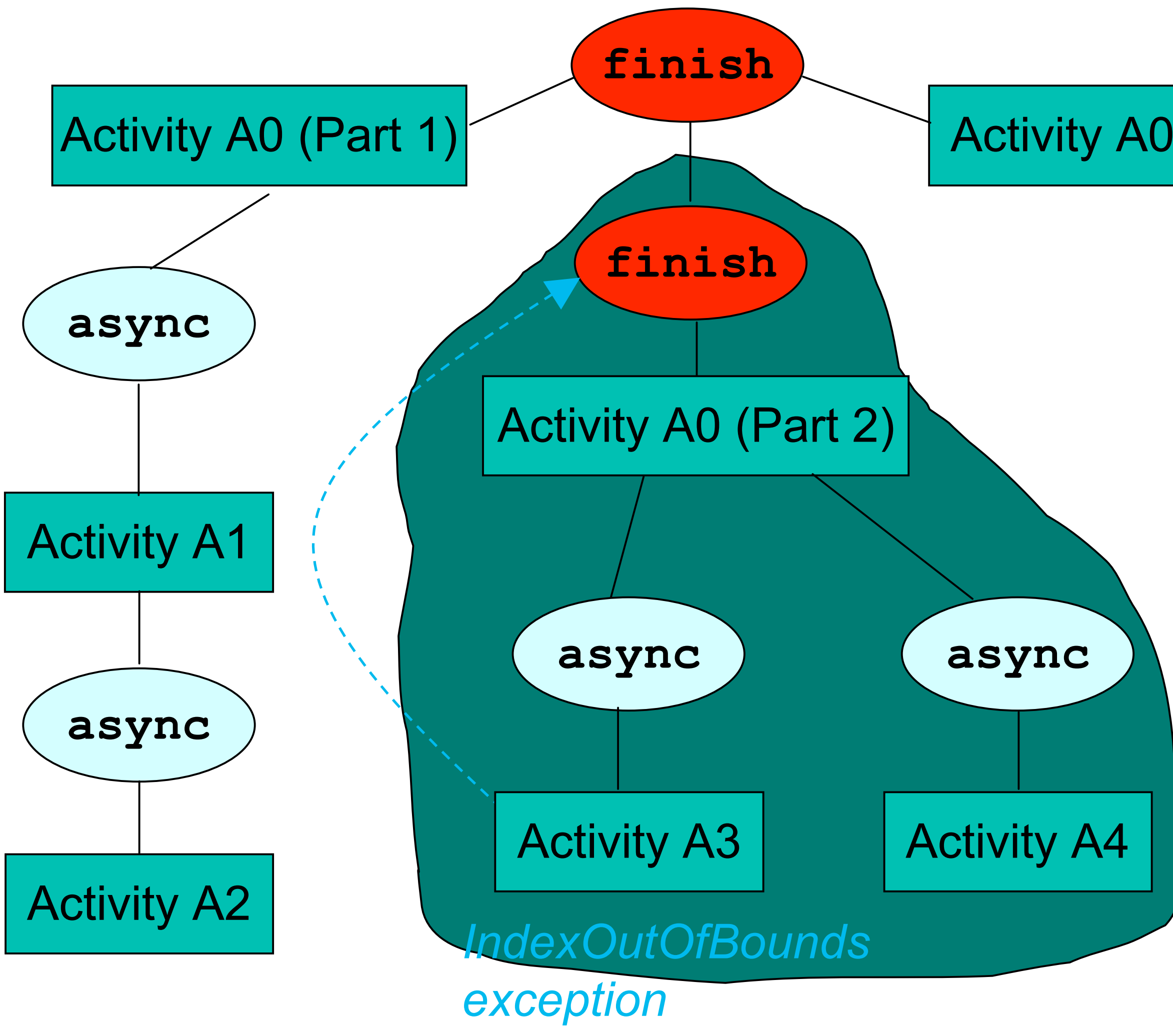
- no exceptions are ‘thrown on the floor’
- exceptions are propagated across activity and place boundaries

Spanning tree Example

```
public class V  {
   final int index;
   V parent;
   int degree;
   V [] neighbors;
   Color color;
   V (int i) {index=i;}

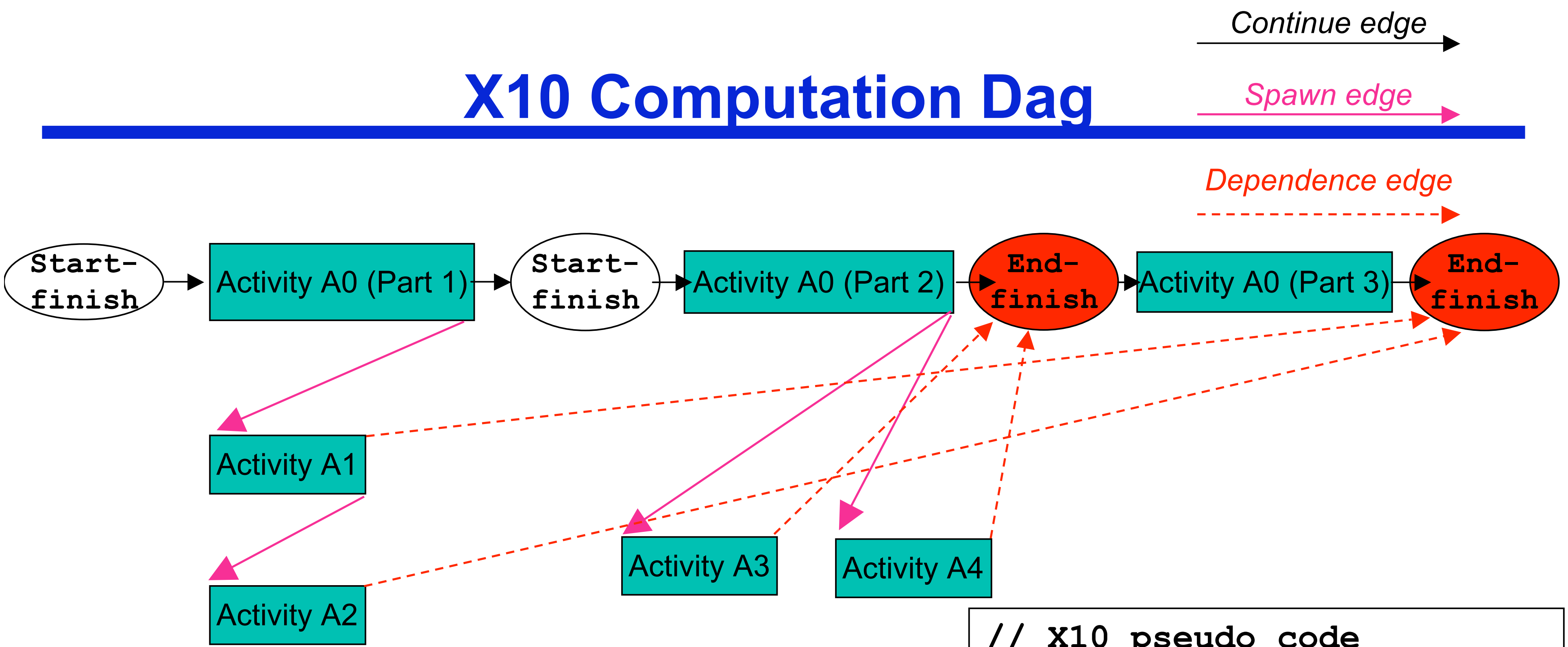
      public void compute()  {
            V node = this;
            for (int k=0; k < node.degree; k++) {
              final V v = node.neighbors[k];
              if (v.color.color==0
                 && UPDATER.compareAndSet(v.color,0,1)) {
                 // Use CompareAndSet from JUC
                v.parent=node;
                async v.compute();
              }
         }
      . . .
      finish root.compute();
      . . .
}
```

Asynchronous Activities in X10




```
// X10 pseudo code
main(){ // implicit finish
  Activity A0 (Part 1);
  async {A1; async A2;}
  try {
    finish {
      Activity A0 (Part 2);
      async A3;
      async A4;
    }
  catch (…) { … }
  Activity A0 (Part 3);
}
```

X10 Computation Dag



"Deadlock-Free Scheduling of X10 Computations with Bounded Resources", S.Agarwal et al, SPAA 2007.

Theorem 2.6: A work-stealing execution of a (terminally strict) X10 multithreaded computation with finish & async constructs on P processor uses at most S_1*P space in its dequeue's, where S_1 is the maximum stack depth in a sequential execution of the program.

```
// X10 pseudo code
main(){ // implicit finish
  Activity A0 (Part 1);
  async {A1; async A2;}
  try {
    finish {
      Activity A0 (Part 2);
      async A3;
      async A4;
    }
  catch (…) { … }
  Activity A0 (Part 3);
}
```

Behavioral annotations

nonblocking

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

sequential

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

- 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 (point p: R) S

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

- 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-worker 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**)**

```
// 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;
}
```

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 **foreach** with **finish**
- declare data to be read-only where possible (final or value type)

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 :**
  - `region R = [0:M-1,0:N-1];`
  - `for (point p [i, j] : R) { ... }`
  - `for (point [i, j] : R) { ... }`
  - ```
    point sum (point [i,j], point [k, l])
                              { return [i+k, j+l]; }
    ```
  - `int [.] iarr = new int [R] (point [i,j]) { return i; }`
    ```

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)**

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...**

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:

```
nullable<RefType> [.] farr = new RefType[R];  // init with null value
```

Mutable array with references to mutable objects:

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

RefType [.] farr = new RefType [R] (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
ValType [.] V = new ValType[N] (point[i])
                              { return ValType(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(); }
```

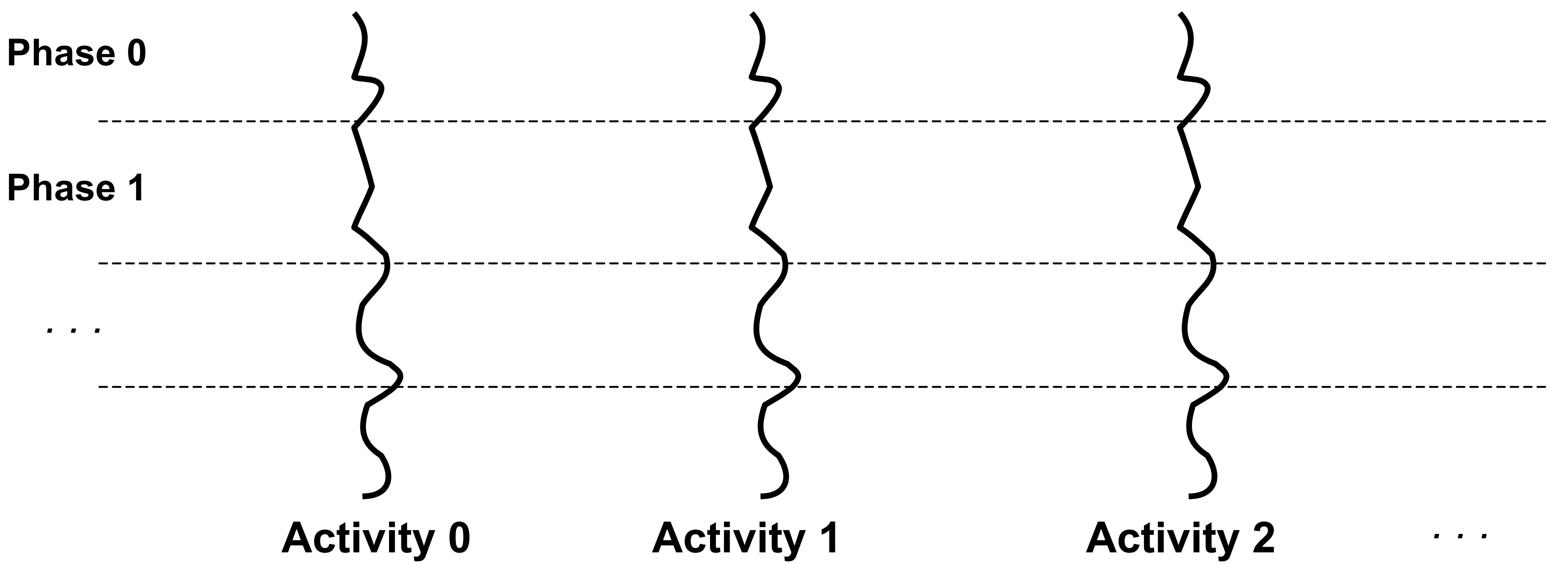
Value array of ‘reference to value objects’ (value object can be inlined)

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

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

Clocks: Motivation

- Activity coordination using **finish** and **force()** is accomplished by checking for activity termination
- But in many cases activities have a producer-consumer relationship 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**
- *Design clocks to offer deadlock-free coordination among a dynamically varying number of activities.*



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

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

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.)