X10: Concurrent Object-Oriented Programming for Modern Architectures

Vijay Saraswat, Christoph von Praun

September, 2006

IBM Research

Tutorial outline

1) X10 Project

2) X10 Introduction
- cheat sheets
- Hello world
- comparison to Java

3) Sequential X10

4) Concurrency in X10
- activities
- atomic blocks
- clocks, clocked variables

5) Distributed X10
- places
- distributions and distributed arrays

6) X10 Array Language

7) Current Status and Future Work

X10 Project

Acknowledgments

- **X10 Core Team**
  - **Rajkishore Barik**
  - **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. "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.
4. "X10: an Experimental Language for High Productivity Programming of Scalable Systems", K. Ebcioglu, V. Sarkar, V. Saraswat. P-PHEC workshop, February 2005.

Upcoming tutorials

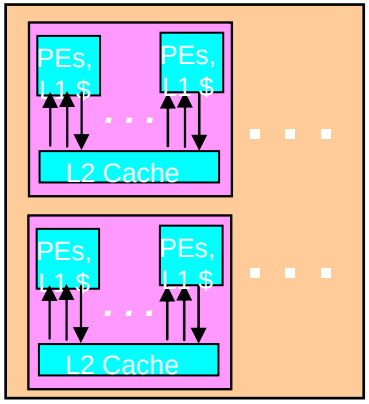
- OOPSLA 2006

A new era of mainstream parallel processing

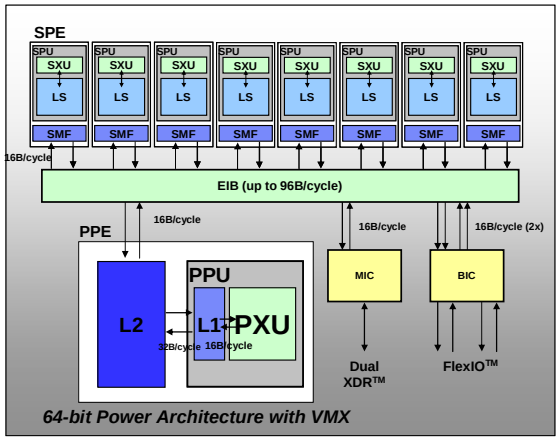
The Challenge

Parallelism scaling replaces frequency scaling as foundation for increased performance ➔ Profound impact on future software

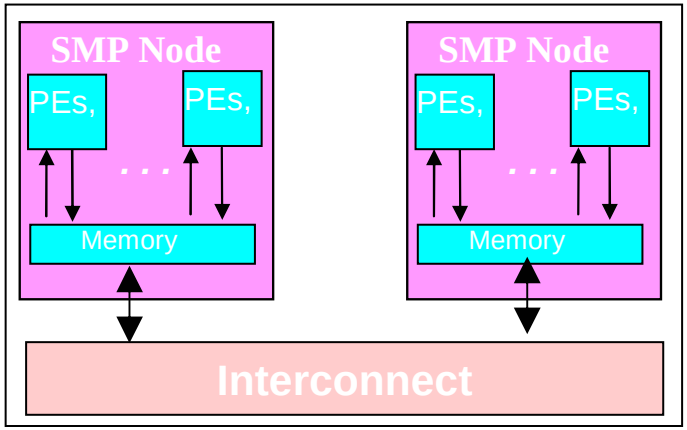
Multi-core chips



Heterogeneous Parallelism



Cluster Parallelism



Our response:

Use X10 as a new language for parallel hardware that builds on existing tools, compilers, runtimes, virtual machines and libraries

Programming Technologies

The X10 programming model

Support for productivity

- Axiom: Exploit proven OO benefits (productivity, maintenance, portability benefits).
- Axiom: Rule out large classes of errors by design (Type safe, Memory safe, Pointer safe, Lock safe, Clock safe …)
- Axiom: Support incremental introduction of explicit place types/remote operations.
- Axiom: Integrate with static tools (Eclipse) -- flag performance problems, refactor code, detect races.
- Axiom: Support automatic static and dynamic optimization (CPO).

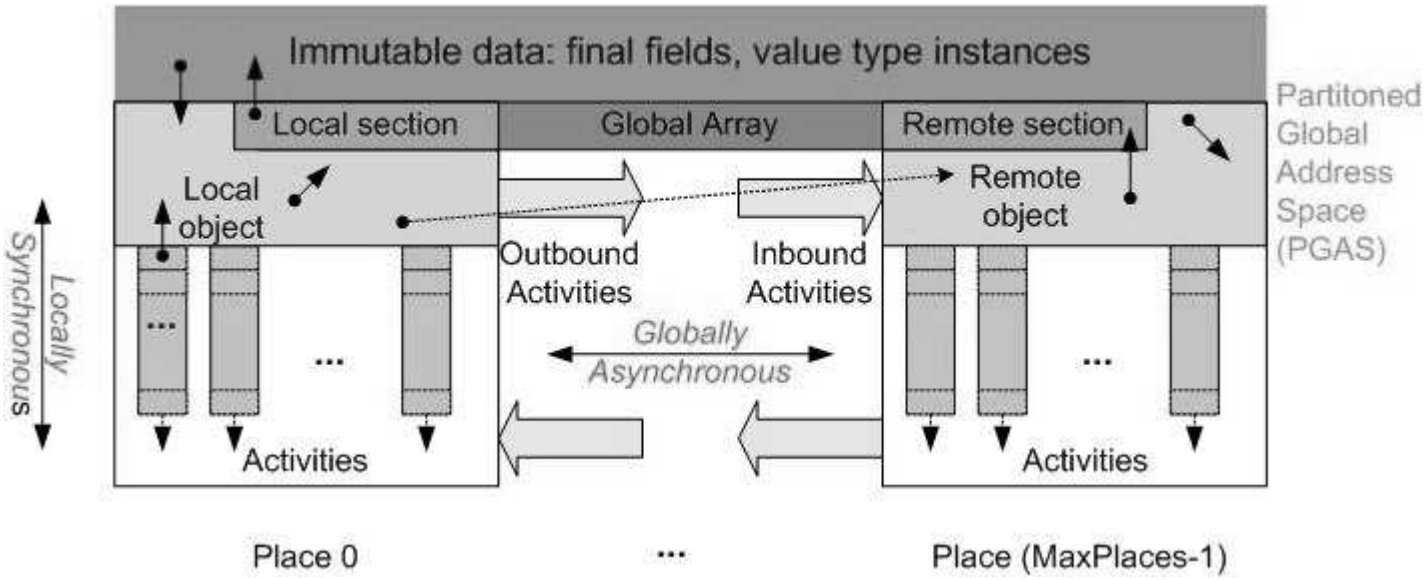
Support for scalability

- Axiom: Provide constructs to deal with non-uniformity of access.
- Axiom: Build on asynchrony. (To support efficient overlap of computation and communication.)
- Axiom: Use scalable synchronization constructs.
- Axiom: Permit programmer to specify aggregate operations.

Our philosophy

- **Be conservative strategically, aggressive tactically.**
- **Build on sound foundations, but design for the programmer.**
  - Not the theoretician, not the language designer.
- **Use Occam's Razor.**
  - Avoid a variety of linguistic mechanisms for the same programming idiom.
- **Steal.**
- **Focus on a few things, do them well.**
- **Keep the language small.**
- **Keep the language orthogonal.**
- **Ensure the language "grows on you."**
- **Exploit structure in concurrency.**
- **Make easy things easy, hard things possible.**

The X10 programming model



Place = collection of resident activities & objects

Storage classes

- **Immutable Data**
- **PGAS**
  - Local Heap
  - Remote Heap
- **Activity Local**

Locality Rule

Any access to a mutable datum must be performed by a local activity ➔ remote data accesses can be performed by creating remote activities

Ordering Constraints (Memory Model)

Locally Synchronous:

Guaranteed coherence for local heap ➔ Sequential consistency

Globally Asynchronous:

No ordering of inter-place activities ➔ use explicit synchronization for coherence

X10 project landscape

- Core concurrency and distribution design.
- Array language design
  V1 → V2
- Extern interface
- JVM implementation
- Place types
- Applications
- X10DT
- Dependent types
- Implicit syntax
- Memory model
- Generics
- Tiled regions
- X10lib
- XVM spec
- FP semantics
- Relaxed exceptions
- Annotations

02/04 | 07/04 | 02/05 | 07/05 | 02/06 | 07/06 →

Programming Technologies

Tutorial outline

1) X10 Project

2) X10 Introduction
- cheat sheets
- Hello world
- comparison to Java

3) Sequential X10

4) Concurrency in X10
- activities
- atomic blocks
- clocks, clocked variables

5) Distributed X10
- places
- distributions and distributed arrays

6) X10 Array Language

7) Current Status and Future Work

X10 Cheat Sheet

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, 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***

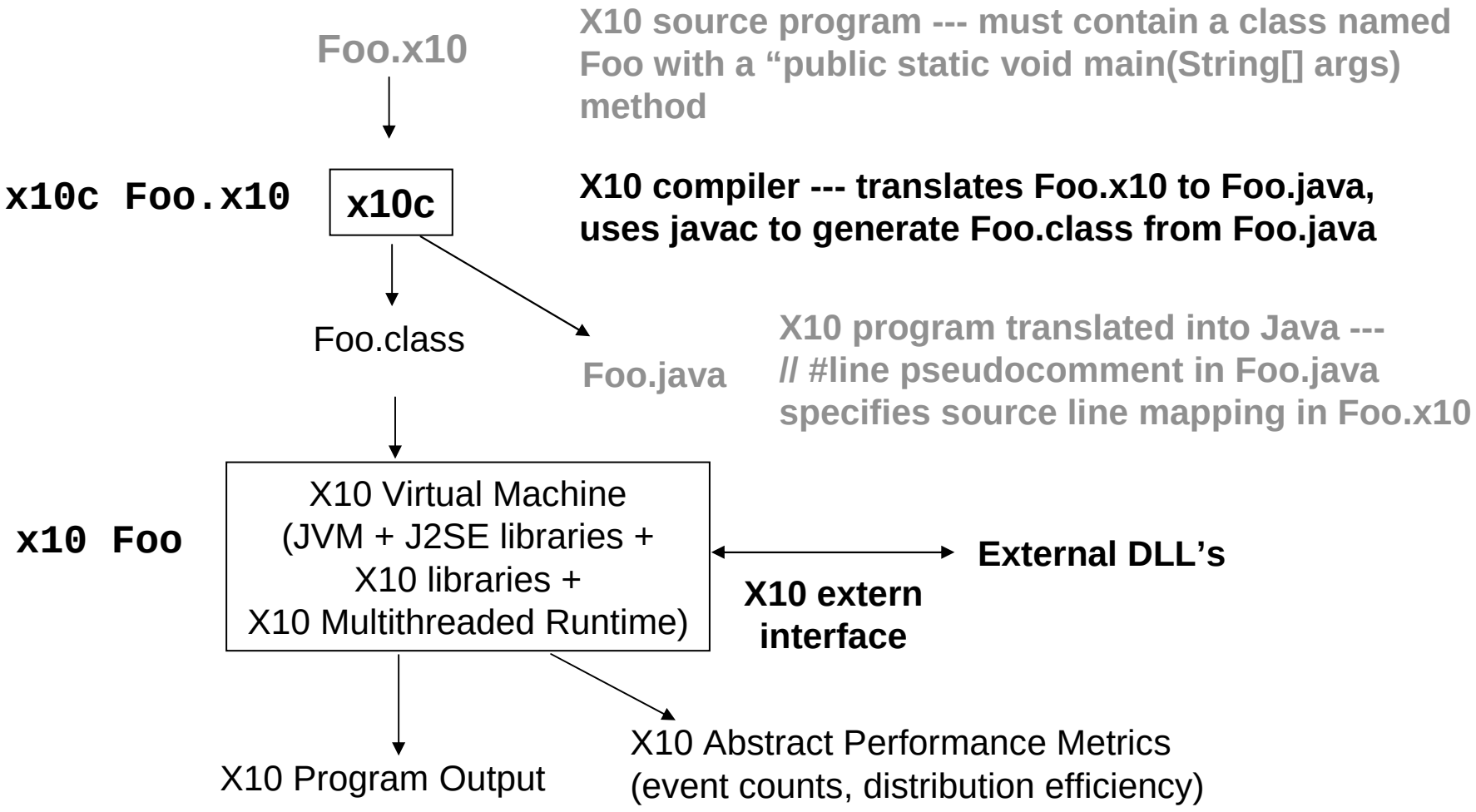
Language supports type safety, memory safety, place safety, clock safety.

Programming Technologies

X10 Startup

- **Translation**
- **Machine model**
- **Startup**
- **Hello World**

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

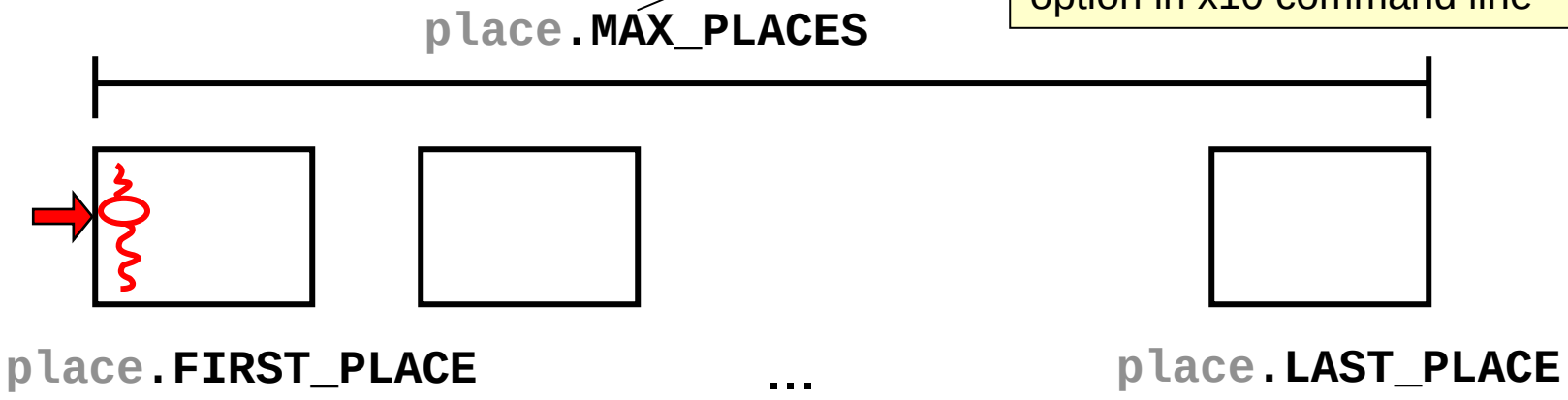
Case 1: Carats indicate column range

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

Sequential X10

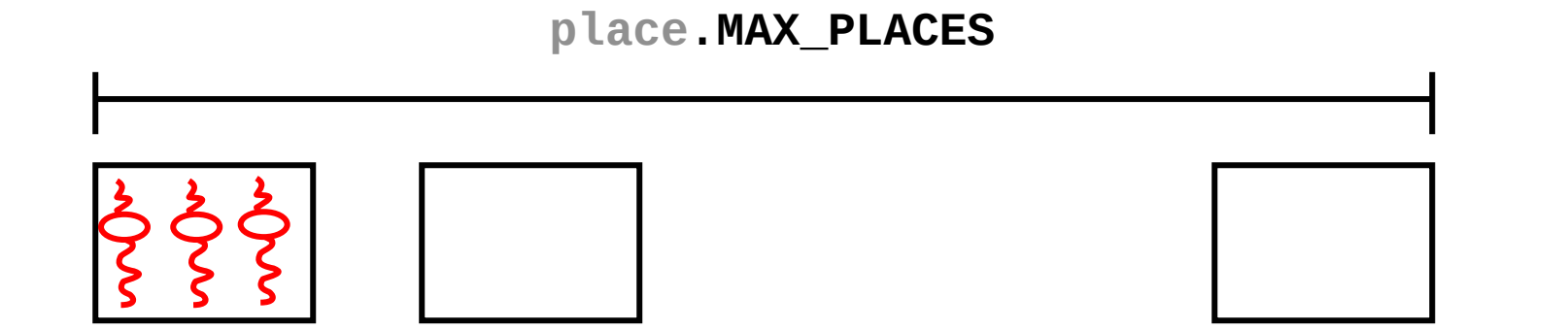
Runtime constant.
Can be changed by using the NUMBER_OF_LOCAL_PLACES option in x10 command line



root activity

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

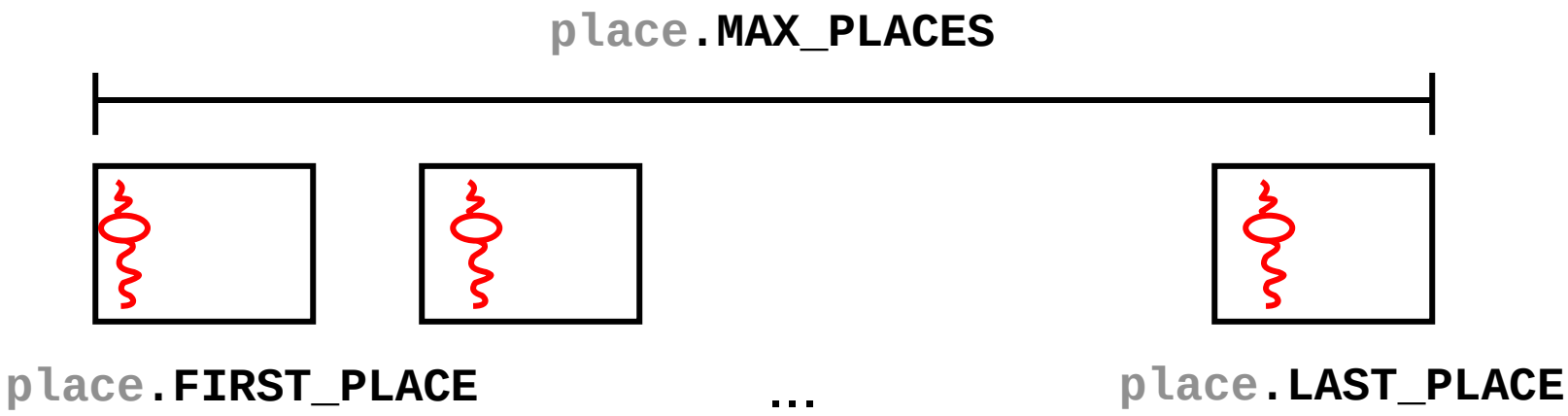
Parallel X10



root activity

```
public class HelloWorld2 {
  public static void main(String[] args) {
    foreach (point [p] : [1:2])
      System.out.println("Hello from activity " + p + "!");
  }
}
```

Distributed X10




```
public class HelloWorld2 {
  public static void main(String[] args) {
    ateach (place p: dist.factory.unique(place.MAX_PLACES))
      System.out.println("Hello from place " + p + "!");
  }
}
```

root activity

Current prototype simulates places within one Java virtual machine.
Distributed X10 implementation being developed at Purdue University.

Comparison with Java

Comparison with Java (1/2)

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

Comparison with Java (2/2)

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
- **A formal memory model**
- **FP support**

Future work

Tutorial outline

1) X10 Project

2) X10 Introduction
- cheat sheets
- Hello world
- comparison to Java

3) Sequential X10

4) Concurrency in X10
- activities
- atomic blocks
- clocks, clocked variables

5) Distributed X10
- places
- distributions and distributed arrays

6) X10 Array Language

7) Current Status and Future Work

Sequential X10

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

Sequential X10

- ✓ **Classes and interfaces**
  - ✓ **Fields, methods, Constructors**
  - ✓ **Encapsulated state**
  - ✓ **Single inheritance**
  - ✓ **Multiple interfaces**
  - ✓ **Nested/Inner/Anon classes**
- ✓ **Static typing**
- ✓ **Objects, GC**
- ✓ **Statements**
  - ✓ **Conditionals, assignment,…**
  - ✓ **Exceptions (but relaxed)**

- ? **Not included**
  - ? **Dynamic linking**
  - ? **User-definable class loaders**
- x **Changes**
  - x **Value types**
  - x **Aggregate data/operations**
  - x **Space: Distribution**
  - x **Time: Concurrency**
- x **Changes planned**
  - x **Generics**
  - x **FP support**

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);
 } …
}
```

X10 safety properties

Type safety

- **Every location has a static type**
- **Runtime invariant**

  A location contains only those values whose dynamic type satisfies the constraints imposed by the location's static type.

- **Every value has a dynamic type**
- **Runtime invariant**

  Every runtime operation performed on the value in a location is permitted by the static type of the location.

Based on type safety:

→ Memory safety
→ Pointer safety
→ Clock safety
→ Place safety

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.

Safety: Static vs. dynamic checking

X10 virtual machine maintains a set of invariants (type safety).

- Some guarantees through static type check.
- Complementary "local" dynamic checks.
- Semantic annotations and static analysis / program transformations reduce the frequency of dynamic checks.

Dynamic checks

BadPlaceException

- Local access to remote object

```
void m (Object o) {
  if (o.location == here)
    // local method invocation
    o.foo();
  else
    // remote method invocation
    finish async (o.location) o.foo();
}
```

ClockUseException

- Access to clock on which current activity is not registered.
- Pass-on of clocks on which the current activity is not live.

ArrayIndexOutOfBoundsException

... <and others like Java>

X10 Standard Library

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

X10 Native Interface

Interface to C / FORTRAN (1/2)

Key issues

- No memory safety in C and FORTAN
- X10 domain should be protected
- Efficient transition from X10 $\leftrightarrow$ C/FORTAN

Calling conventions

- Value types are passed by value
- Instances of reference types and arrays have to be allocated in unsafe memory to allow access from C/FORTAN code.

Interface to C / FORTRAN (2/2)

X10 side:

- Keyword extern for method declaration.
- Compiler generates X10 + C stub code

C/FORTRAN side:

- Stub implements interface generated by x10c and calls native code
- Native code attached as shared library to VM

Example: native code (1/2)

```
extern static void daxpy(int n, double da, double[] dx,
       int incx, double[] dy,int incy);

public static void daxpy(int n, double da, double[] dx,
       int incx, double[] dy,int incy)   {
   // Call C routine passing memory address
   daxpy_C(n, da, dx.address(),incx, dy.address(),incy);
}
```

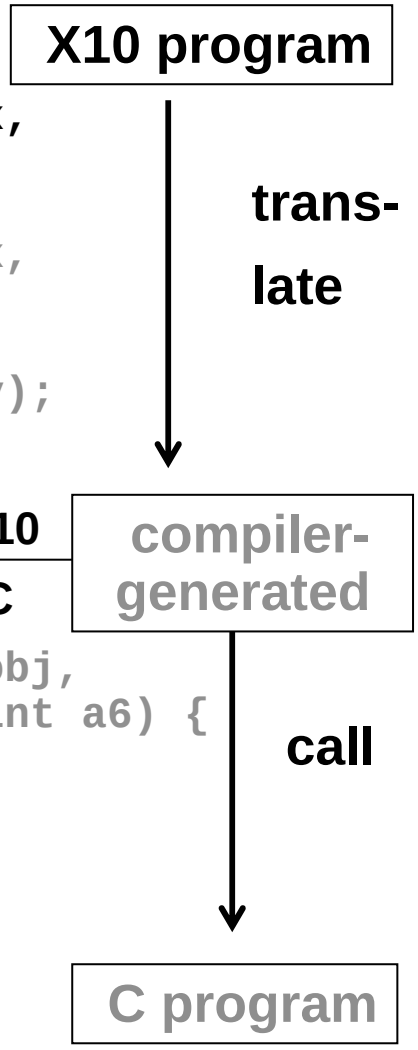
X10

C

```
JNIEXPORT void JNICALL daxpy_C (X10Env* env, jobject obj,
  xint a1, xdouble a2, xlong a3, xint a4, xlong a5, xint a6) {
     daxpy_C (...);
}

extern static void daxpy_C(int n, double da, long dx,
       int incx, long dy, int incy);

<daxpy C-code> in libblas.so
```



Example: native code (2/2)

```
class Daxpy {

   static { System.loadLibrary("blas"); }
   extern static void daxpy(int n, double da, double[] dx,
                            int incx, double[] dy, int incy);

   public static void main(String args[]) {
      final int N = 10;
      double da = 2.0;
      double[] dx = new unsafe double [N];
      double[] dy = new unsafe double [N];
      int incx = 1, incy = 1;

      for (int i = 0; i < N; i++) {
          dx[i] = 4.0;
          dy[i] = 3.0;
      }
      daxpy (n, da, dx, incx, dy, incy);
   }
}
```

declaration

call

Tutorial outline

1) X10 Project

2) X10 Introduction
- cheat sheets
- Hello world
- comparison to Java

3) Sequential X10

4) Concurrency in X10
- activities
- atomic blocks
- clocks, clocked variables

5) Distributed X10
- places
- distributions and distributed arrays

6) X10 Array Language

7) Current Status and Future Work

Concurrency in X10

- **async, finish**
- **future, force**
- **foreach**
- **Global vs. local termination**
- **Exception handling**
- **Behavioral annotations**
- **Possible fallacies and synchronization defects**
- **Compilation aspects**

async

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

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;
}
```

cf Cilk's spawn

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***

```
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;
```

cf Cilk's sync

Termination

Local termination:

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

Global termination:

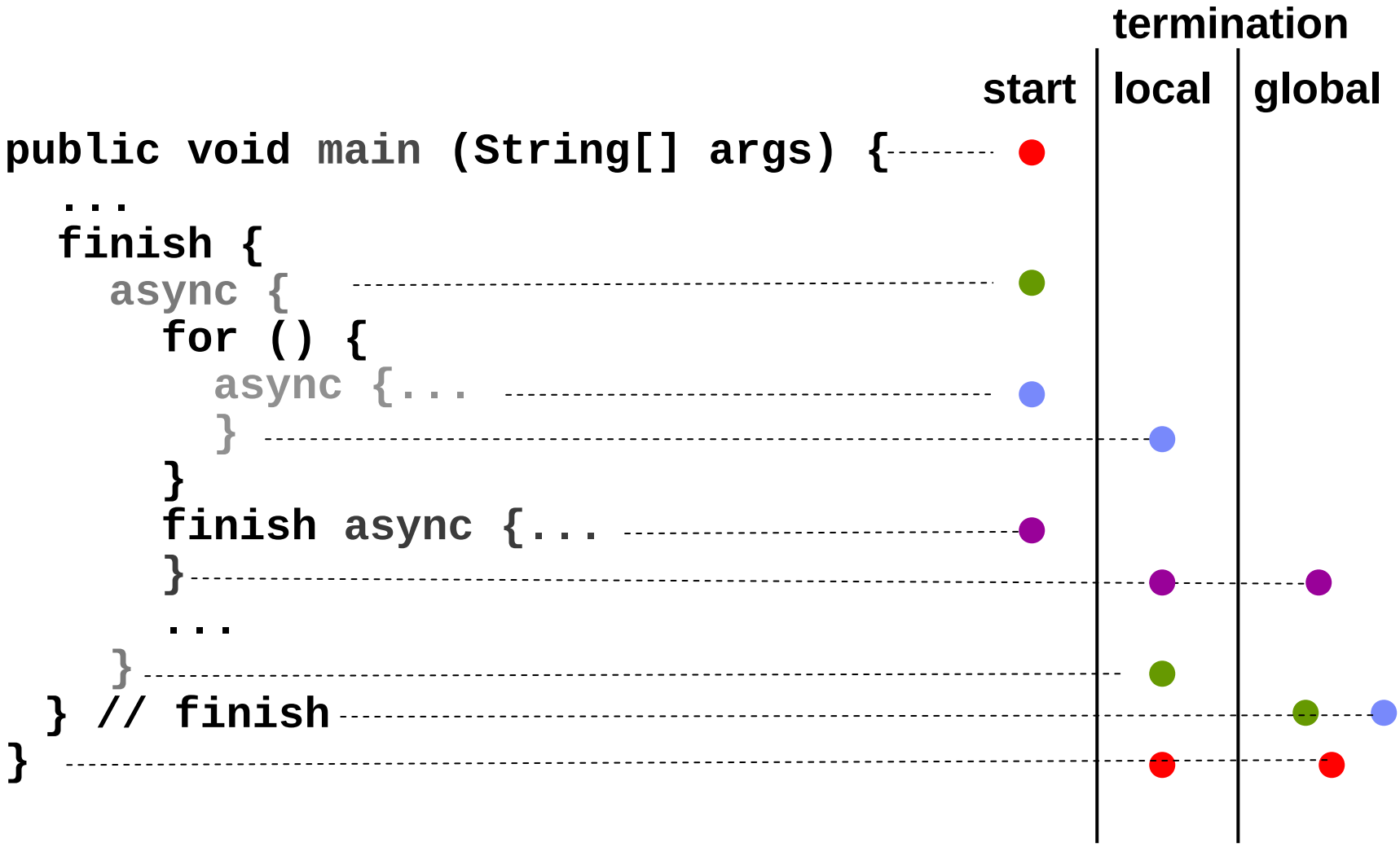
Local termination + activities that have been spawned by *s* terminated globally (recursive definition)

→ main function is **root activity**

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

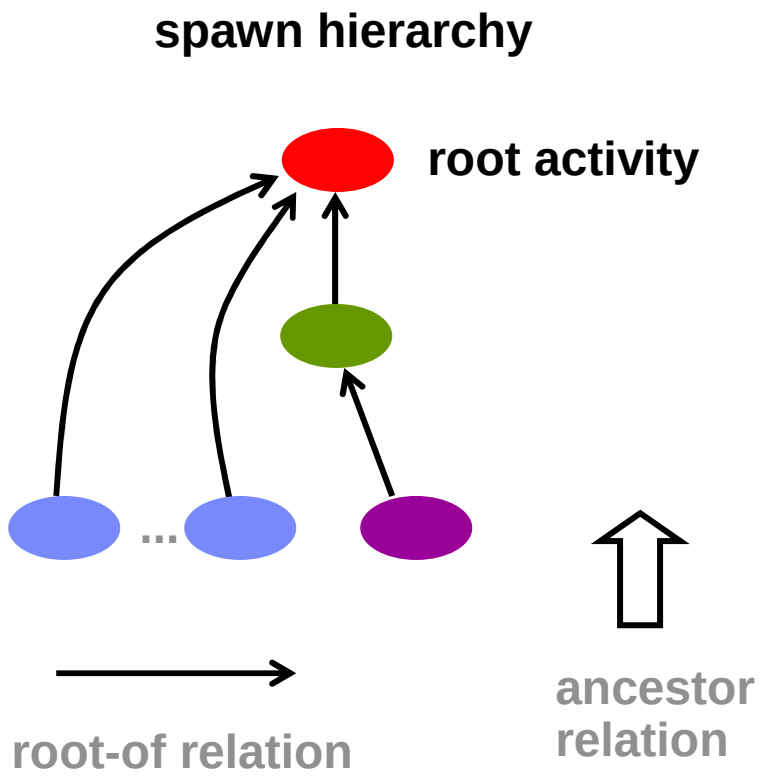
→ 'daemon threads' (child outlives root activity) not allowed in X10

Termination (Example)



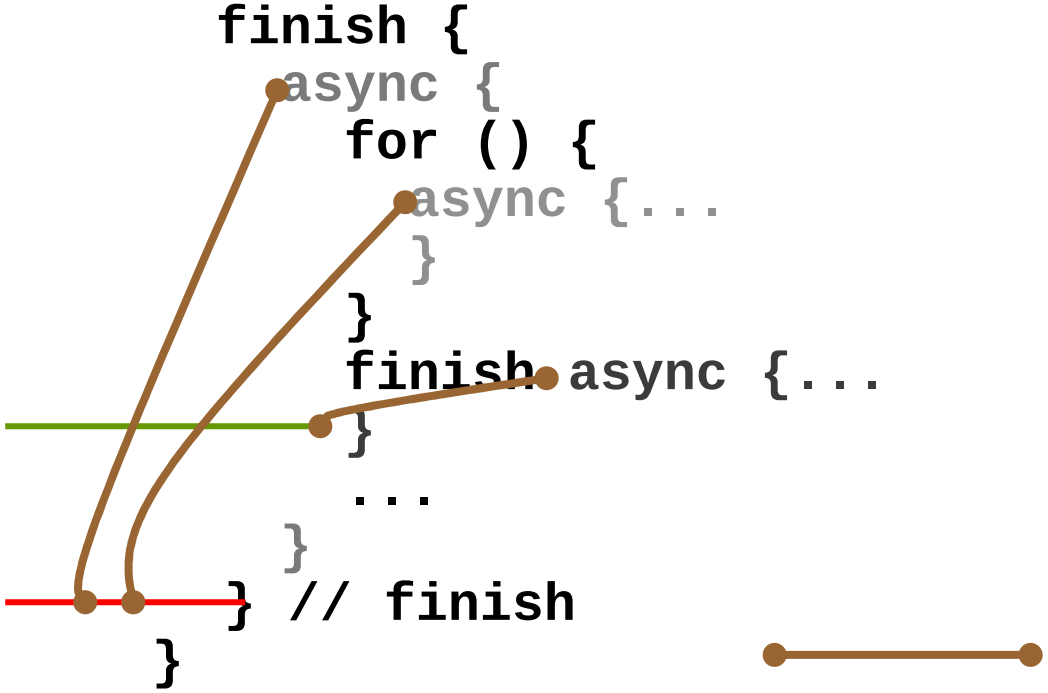
Rooted computation X10

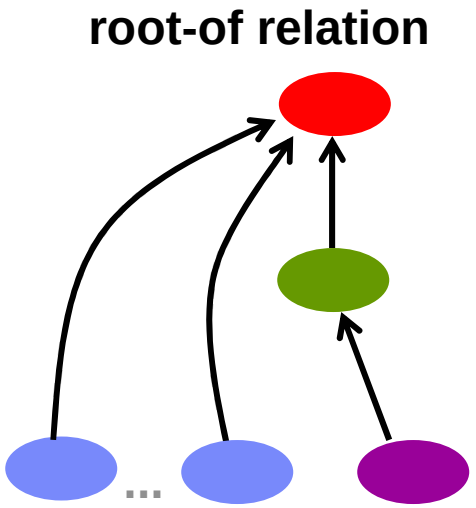
```
public void main (String[] args) {
  ...
  finish {
    async {
      for () {
        async {...
        }
      }
      finish async {...
      }
      ...
    }
  } // finish
}
```



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

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;
        else if ( n == 1 ) return 1;
        else {
            future<int> fn_1 = future { fib(n-1) };
            future<int> fn_2 = future { fib(n-2) };
            return fn_1.force() + fn_2.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.

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)

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.

Note: Checking is not currently implemented.

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)

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();
}
```

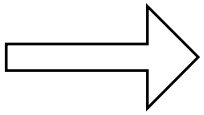
X10 guidelines to avoid deadlock:

- avoid futures as shared variables
- force called by same activity that created body of future

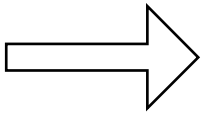
Compilation aspects

Activity inlining

```
foreach ( point[i,j] : a.region )
  a[i,j] = f (a[i,j]);
```



```
foreach ( point[i] : a.region.dim(0) )
  for (point[j] : a.region.dim(1) )
    a[i,j] = f (a[i,j]);
```



```
for ( point[i,j] : a.region )
  a[i,j] = f (a[i,j]);
```

Conditions

- body is **recursively non-blocking**
- body is **local**

Memory Model

Aside: Memory Model

- **X10 v 0.41 specifies sequential consistency per place.**
  - atomic blocks / finish / force have acquire-release semantics.
- **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*.**

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

Concurrency Control: Transactional Memory

- **Atomic blocks**
- **Conditional atomic blocks, when, await**
- **Fallacies, synchronization defects**
- **Compilation aspects**

Atomic blocks simplify parallel programming

- **No explicit locking**
  - No need to worry about lock management details: What to lock, in what order to lock.
- **No underlocking/overlocking issues.**
- **No need for explicit consistency management**
  - No need to carry mapping between locks and data in your head.
- **System can manage locks and consistency better than user**
- **Enhanced performance scalability**
  - X10 distinguishes intra-place atomics from inter-place atomics.
  - Appropriate hardware design (e.g. conflict detection) can improve performance.
- **Enhanced analyzability**
  - First class programming construct
- **Enhanced debuggability**
  - Easier to understand data races with atomic blocks than with critical sections/synchronization blocks

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**)

```
// 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

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

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
- "The atomic statement only guarantees atomicity on successful execution, not on faulty execution"

Transactions: Design rationale

Minimal requirements on runtime support for atomic blocks
- no rollback
- lock-based implementation possible

Weak atomicity model
- atomicity and isolation are only guaranteed with respect to other transactions

→ concurrent transactional and non-transactional access foils transaction semantics.

→ see memory model

Ordering
- Transactions issued by a thread are performed in program order.

Nesting
- atomic blocks: closed nesting as an optimization, no open nesting
- conditional atomic blocks: cannot be nested in other atomic blocks.

Example: Loop parallelization

serial program

```
for (point p[i]: indexset)
{ ti; }
```

---

data parallel (doall): only correct if ti have no data dependences.

```
finish foreach (point p[i]: indexset)
{ ti; }
```

---

task parallel: only correct if ti are commutative and associative.

```
finish foreach (point p[i]: indexset)
{ atomic ti; }
```

Example: Loop parallelization

speculative parallelization: always correct

```
// global shared var
final boolean [.] ti_done = new boolean [indexset.region];

finish foreach (point p[i]: indexset) {
   if (i==0)
     atomic { ti; ti_done[i] = true; }
   else
     when (ti_done[i-1]) { ti; ti_done[i] = true; }
}
```

- Transactions commit in program order.
- Implementations that are not based on speculative execution will serialize this loop.

Example use of atomic blocks: latching variable

```
class LatchVar {
    boolean available = false;
    double value;
    atomic void set (double val) {
        if ( available ) return false;
        // these assignments happen only once.
        this.value = val;
        this.available = true;
    }
    double get () {
        when ( available ) {
            return this.value;
        }
    }
    atomic boolean ready () { return available; }
}
```

Example use of atomic blocks: future

```
LatchVar lv =
  new RunnableLatch() {
     public LatchVar run() {
        LatchVar l = new LatchVar();
        async ( P ) {
           double X;
           finish X = e;
           l.setValue( X );
        }
        return l;
    }
  }.run();
double d = lv.get();
```

Exception handling and type genericity are omitted for clarity.

```
future<double> fv = future (P) { e }
double d = fv.force();
```

X10 language equivalent.

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

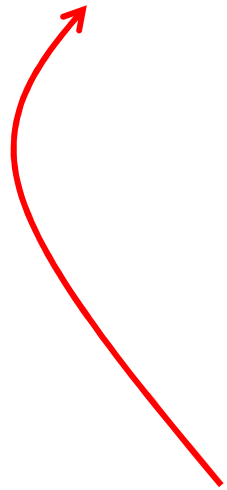
Compilation aspects

Combine atomic blocks

```
      // for all lines in parallel
      finish foreach (...) {

        // for each pixel of the line
atomic for (point [x] : [0:interval.width-1]) {
          Vec col = .... // determine pixel

          // computes the color of the ray
          int red = (int)(col.x * 255.0);
          if (red > 255) red = 255;
          int green = (int)(col.y * 255.0);
          if (green > 255) green = 255;
          int blue = (int)(col.z * 255.0);
          if (blue > 255) blue = 255;
          atomic checksum += red + green + blue;
        } // end for (x)
      }
```

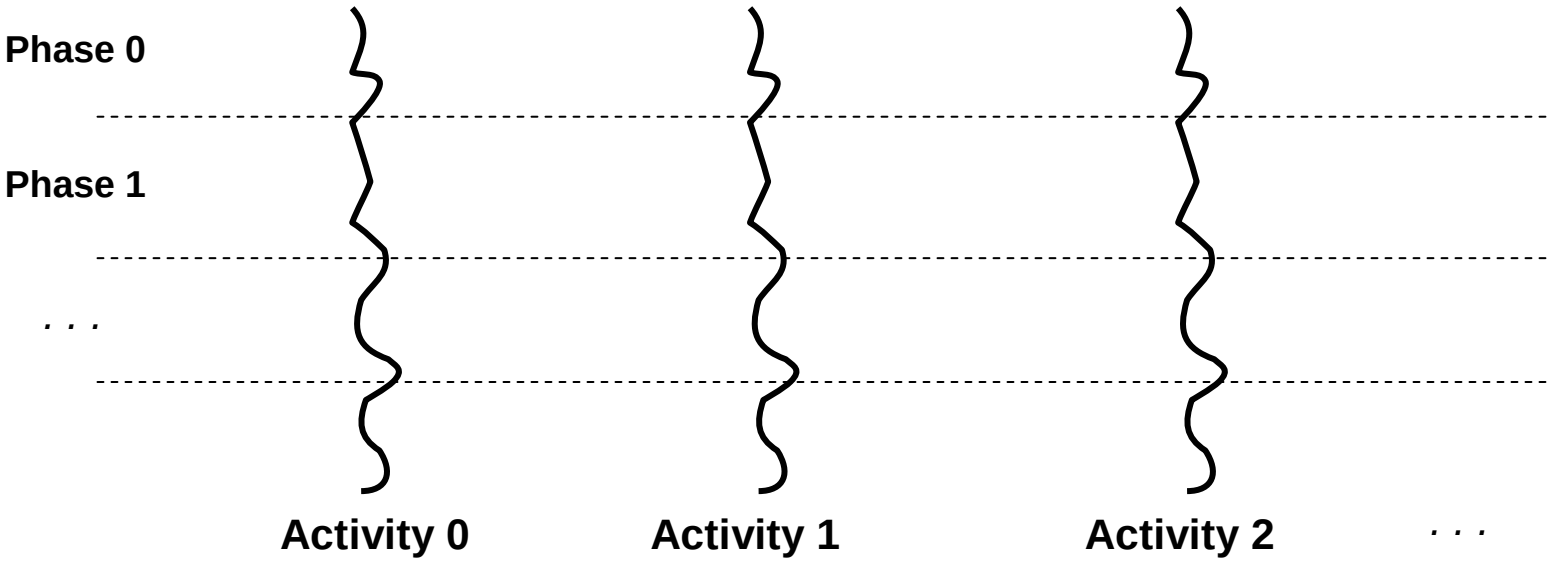


Concurrency Control: Clocks

- **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

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

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
  c.drop();
} // 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.)

Clocked final

- **Clocks permit an elegant form of determinate, synchronous programming.**
- **Introduce a data annotation on variables.**
  - **clocked(c) T f = …;**
  - f is thought of as being "clocked final" – it takes on a single value in each phase of the clock,
- **Introduce a new statement:**
  - **next f = e;**
- **Statically checked properties:**
  - Variable read and written only by activities clocked on c.
  - For each activity registered on c, there are no assignments to f.
  - **next f = e;** is executed by evaluating e and assigning value to **shadow variable** for f.
- **When c advances, each variable clocked on c is given the value of its shadow variable *before* activities advance.**

If activities communicate only via (clocked) final variables, program is determinate.

Not yet implemented.

Synchronous Kahn networks are CF (and DD-free)

- This idea may be generalized to arbitrary mutable variables.
  - Determinate imperative programming.
- Each variable has an implicit clock.
- Each variable has a stream of values.
- Each activity maintains its own index into stream.
- An activity performs reads/writes per its index (and advances index).
- Reads block.

```
clock c = new clock();
clocked(c) int x = 1, y=1;
async clocked (c)
  while (true) {
  next x = y; next;
}
async clocked (c)
  while (true) {
  next y = x+y; next;
}
```

Guaranteed determinate, though programs may deadlock (cf. asynchronous Kahn networks.)

Clock safety

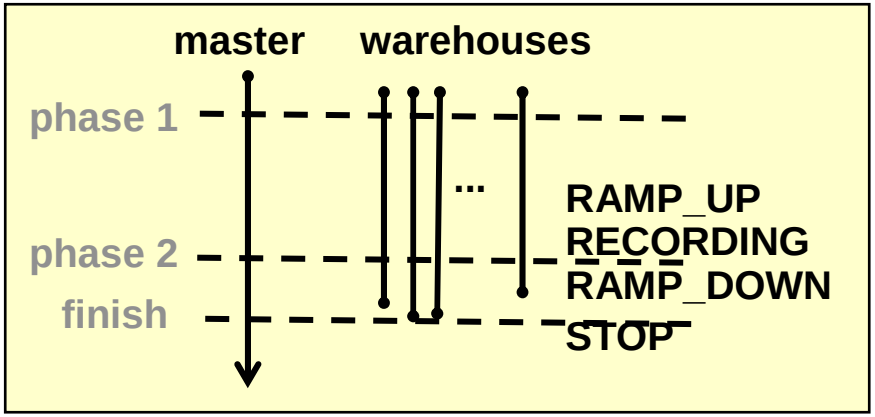
- An activity may be registered on one or more clocks
- Clock c can advance only when all activities registered with the clock have executed c.resume() and all posted activities have terminated globally.

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

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

Tutorial outline

1) X10 Project

2) X10 Introduction
- cheat sheets
- Hello world
- comparison to Java

3) Sequential X10

4) Concurrency in X10
- activities
- atomic blocks
- clocks, clocked variables

5) Distributed X10
- places
- distributions and distributed arrays

6) X10 Array Language

7) Current Status and Future Work

Distributed X10

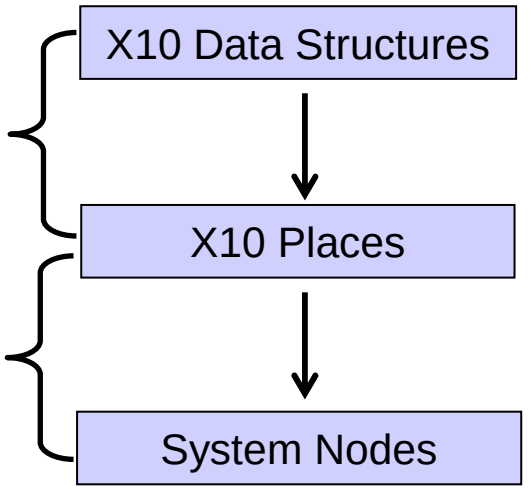
- **Places**
- **Locality rule**
- **Distributions**
- **async, futures**
- **ateach**
- **Distributed arrays**

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.

Place safety

- **The X10 type system is place sensitive.**
- **The static type of each location is a pair T@P, where T is a datatype and P is a placetype.**
  - *PlaceType:*
    here | **place** | **activity** | **current** | *Place* | *?*

Runtime invariants:

- **A reference stored in the location must point to an object located at the place specified by the placetype.**
- **Activity local objects are not shared**

Currently being implemented, in collaboration with Palsberg and Grothoff.

Activity-local objects known to be not shared.

Place-local objects known to not need “fat pointer” references

Placetype system will help eliminate BadPlace checks

We believe this will lead to significant performance gains.

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] } ;
```

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
```

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.

Example: converting foreach to ateach (TutAteach2)

Case 1: All loop iterations are independent.

- **foreach version:**

```
finish foreach ( point[i,j] : a.region )
        a[i,j] = f (a[i,j]);
```

- **ateach version #1:**

```
finish ateach ( point[i,j] : a.distribution)
        a[i,j] = f (a[i,j]);
```

- **ateach version #2 (create only one activity per place):**

```
finish ateach ( point p : dist.factory.unique() )
   for ( point[i,j] : a.distribution | here )
        a[i,j] = f(a[i,j]);
```

Example: converting foreach to ateach (TutAteach2)

Case 2: Iteration across rows are independent (only outer loop can execute in parallel)

- **foreach version:**

```
finish foreach ( point [i]: [1:N] )
      for ( point[j]: [2:N] )
          a[i,j] = f(a[i,j-1])
```

- **ateach version:**

```
// Assume that N is a multiple of place.MAX_PLACES
finish ateach ( point[i] : dist.factory.block([1:N]) )
      for ( point[j]: [2:N] )
          a[i,j] = f(a[i,j-1])
```

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 Project

2) X10 Introduction
- cheat sheets
- Hello world
- comparison to Java

3) Sequential X10

4) Concurrency in X10
- activities
- atomic blocks
- clocks, clocked variables

5) Distributed X10
- places
- distributions and distributed arrays

6) X10 Array Language

7) Current Status and Future Work

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
```

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

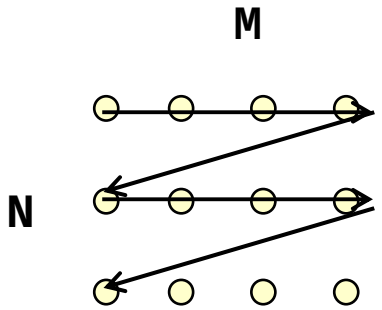
```
region r = [0:N];
dist d = r->here
    → dist d = dist.factory.constant(r, here);
dist d = 1000->here
    → dist d = dist.factory.constant([0,1000],
 here);
```

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] = ..;
```

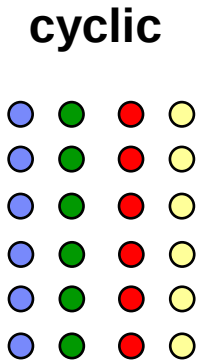


- **initial values in darr are `0.0`**
- **Iteration schema**
  - ‘lexicographical order’ (standard, fix)
  - **`[0,0], [0,1], [0,2], ...`**
- **Storage layout**
  - row major (fix)
  - spatial access locality with standard iteration schema

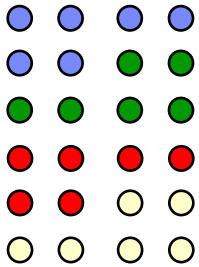
Distributed multidimensional arrays

```
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

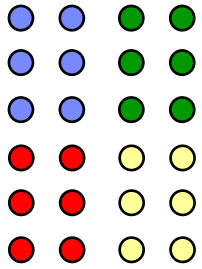


cyclic



block cyclic

for 1D arrays: cf. UPC



tiled

Future work: hierarchically tiled regions

Optimization of **rank independent** code

```
for (point p: darr.region)
   darr[p] = ...;
```

Information about darr.region:

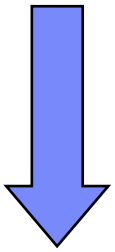
- number of dimensions
- shape of region (rectangular, triangular, ...)
- bounds and step

Determined by

- context sensitive data-flow analysis
- dependent types can provide this information

Optimization of **rank independent** code

```
for (point p: darr.region)
   darr[p] = ...;
```



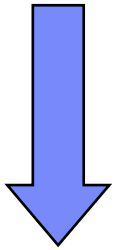
Optimized for dim=2

darr.region is rectangular and dense

```
for (int i = darr.region.rank(0).low();
          i < darr.region.rank(0).high(); ++i)
   for (int j = darr.region.rank(1).low();
             j < darr.region.rank(1).high(); ++j)
      darr[i,j] = ...;
```

Optimization of **rank independent** code

```
for (point p: darr.region) {
     __place_check(here, darr.distribution[p]);
     darr[p] = ...;
}
```



Optimized: darr.distribution is constant distribution

```
if (!darr.distribution.isLocal())
    throw new BadPlaceException();
for (point p: darr.region) {
     darr[p] = ...;
}
```

Distributed arraycopy (first version)

```
static void arraycopy( double[.] src, double[.] dst)
                       throws RegionMismatchException {
  if ( src.distribution.region !=
       dst.distribution.region )
      throw new RegionMismatchException (src, dst);

    ateach (point i : dst.distribution)
      dst[i] = future(src[i]){src[i]}.force();
}
```

- Spawn activity for every index point.
- Code is independent of the rank of the array

Distributed arraycopy (second version)

```
static void arraycopy( double[.] src, double[.] dst)
                       throws RegionMismatchException {
  if ( src.distribution.region !=
       dst.distribution.region )
       throw new RegionMismatchException (src, dst);


   ateach ( distribution.unique(dst.distribution.places) )
       for ( i : dst.distribution | here )
          dst[i] = future(src[i]){src[i]}.force();
}
```

- Spawn one activity in each place that hosts a part of the destination array.

Distributed arraycopy (third version)

```
static void arraycopy( double[.] src, double[.] dst)
                       throws RegionMismatchException {
  if ( src.distribution.region !=
       dst.distribution.region )
              throw new RegionMismatchException (src, dst);
     ateach (point _ : dist.unique(dst.places) ) {
     region local = (dst.distribution | here).region;
    foreach (place p : (src.distribution | local).places) {
       region remote = (src.distribution | p).region;
       region common = local && remote;
       a[common] = future (p){src[common]}.force();
     }
   }
}
```

local array copy

- Spawn one activity per dst-place and
- Create one future per place p to which src maps an index in (dest.distribution | here).

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

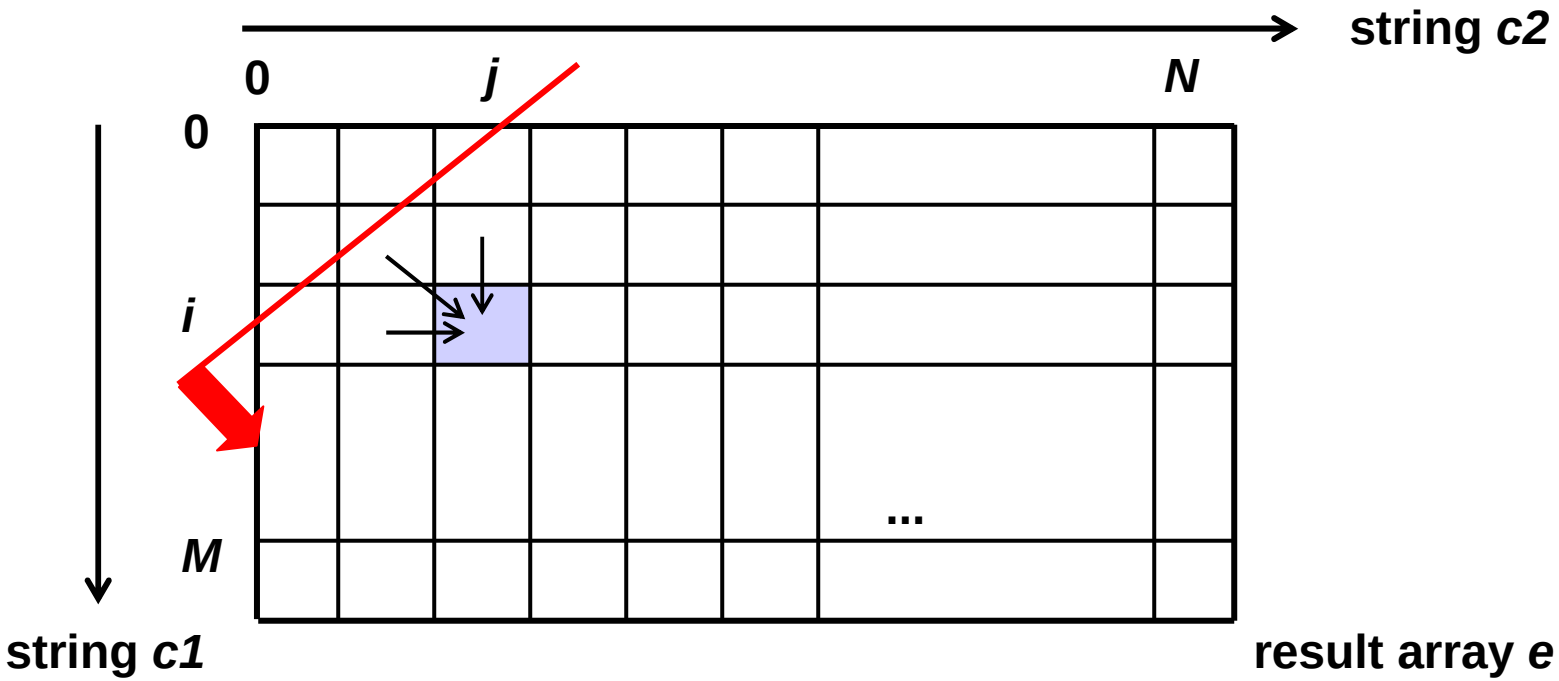
Updating one array with another.

Programming Technologies

Edmiston

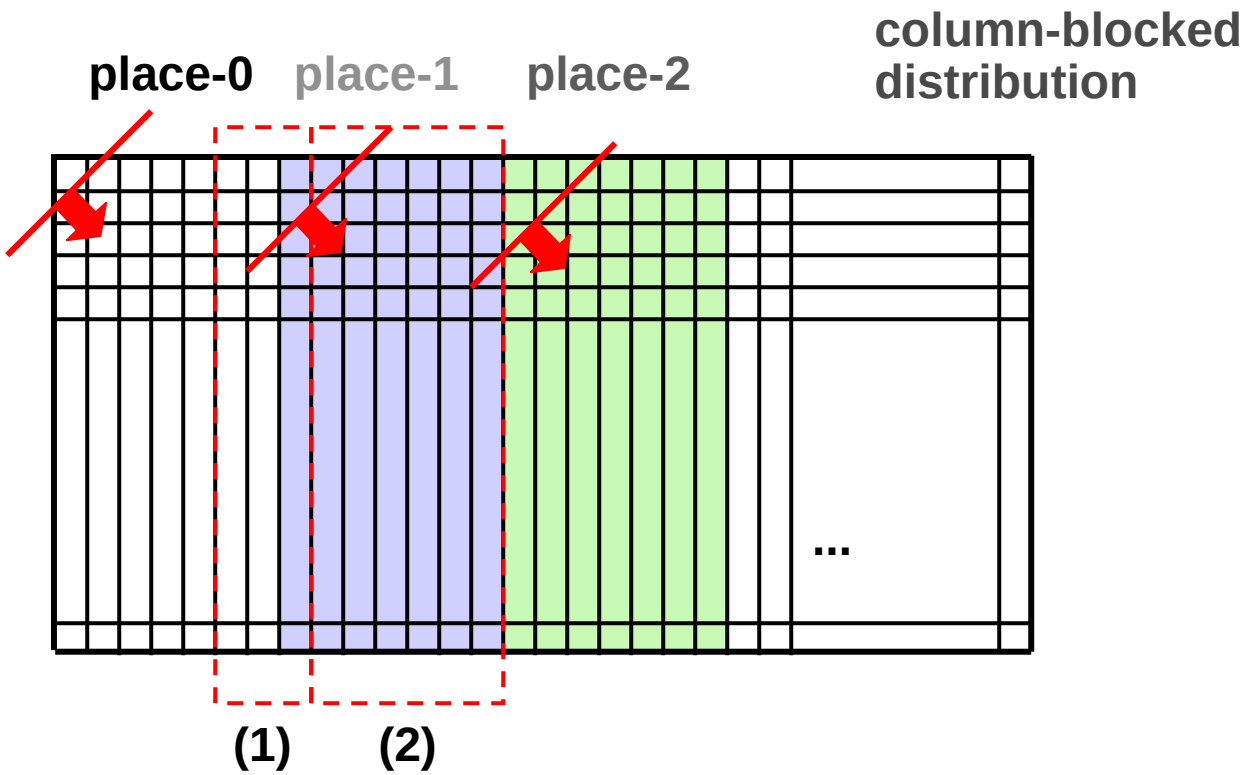
wavefront computation

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

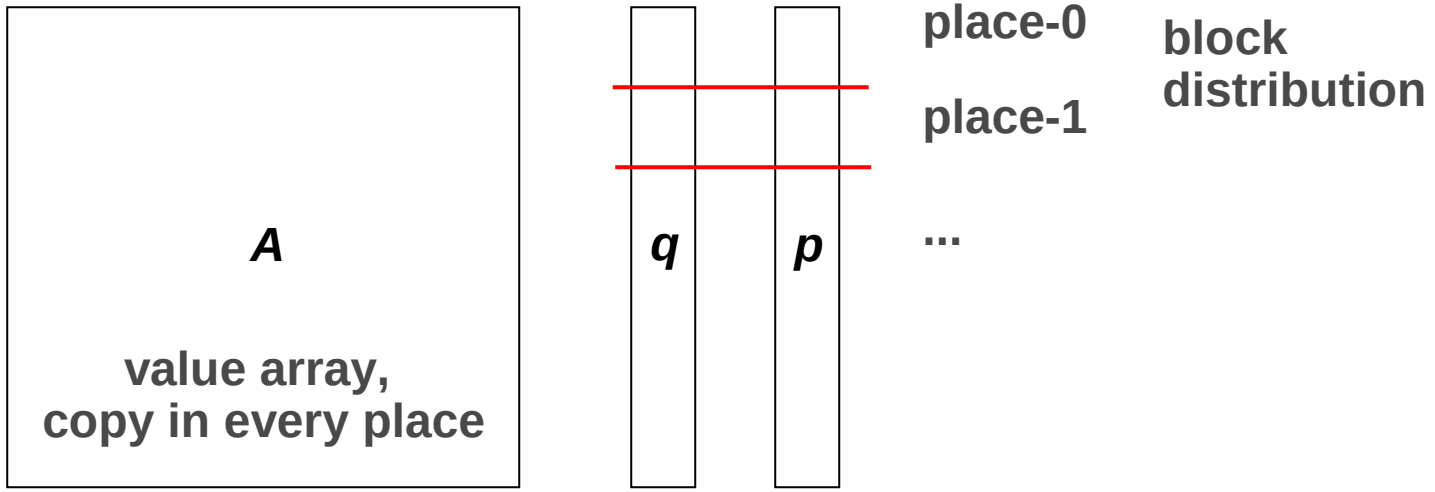
```
final RandCharStr c1, c2;
final int N = c1.s.length-1, int M = c2.s.length-1;
final dist D = columnBlocked([0:N],[0:M]);
final int[.] e = new int[D];

// 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

X10 in Comparison

- **MPI + OpenMP**
- **UPC**
- **Exemplary stencil computations in**
  - C/MPI
  - Titanium
  - UPC
  - X10
  - C++ / htalib

X10, in comparison with MPI+OpenMP …

MPI / OpenMP

- **Processes**
- **Programmer-managed global data structures**
- **Message passing w/ programmer-managed marshalling**
  - Includes reductions
- **Low-level message envelopes**
  - <source, destination, tag, communicator>
- **Barriers**
- **OpenMP threads**
- **Locks, critical sections**
- **Affinity directives**
- **INDEPENDENT directive**

X10

- **Places**
- **Partitioned Global Address Space**
- **Asynchronous activities w/ objects and futures**
  - Includes reductions
- **Strongly-typed invocations and return values (futures)**
- **Clocks**
- **Asynchronous activities**
- **Atomic sections**
- **Placetype system (@-clauses)**
- **foreach, ateach statements**

X10 in comparison with UPC

Simple syntax for remote memory accesses: Read is *rval*, write is *lval*	**Same in X10**
Block cyclic distribution of 1D arrays	**More general distributions in X10**
SPMD model with standard synchronizations (barriers, locks), inquiry functions, etc.	**X10 supports both fork-join and SPMD models**
split barriers w/ notify & wait	**Clock now & next ops**
Work sharing supported by upc_forall	**X10 has foreach and ateach**
Type system identifies private vs. shared data. Four classes of pointers (SP & SS pointer operations are expensive): – PP: Private space pointed by Private pointer e.g., int *p1 – SP: Shared space pointed by Private pointer e.g., shared int *p2 – PS: Private space pointed by Shared pointer e.g., int *shared p3 (not recommended!) – SS: Shared space pointed by Shared pointer e.g., shared int *shared p4;	**(X10 may have @activity annotations.) X10 has type-safe object references, not pointers**
Memory consistency can be controlled by user (relaxed vs. strict)	**X10 has two different memory consistency models: within and across places**
Portable (to the extent that ANSI C is portable)	**X10 has stronger portability (like Java)**

Programming Technologies

2D-stencil in C / MPI

code works only with 4 procs and 12x12 mesh

```
#include "mpi.h"
int main( argc, argv )
int argc;
char **argv;
{
    int        rank, value, size, errcnt, toterr, i, j,
   itcnt;
    int        i_first, i_last;
    MPI_Status status;
    double     xlocal[(12/4)+2][12];
    double     xnew[(12/3)+2][12];
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    if (size != 4) MPI_Abort( MPI_COMM_WORLD, 1 );
    /* xlocal[][0] is lower ghostpoints,
   xlocal[][maxn+2] is upper */

    /* Note that top and bottom processes have one less
   row of interior
       points */
    i_first = 1;
    i_last  = maxn/size;
    if (rank == 0)        i_first++;
    if (rank == size - 1) i_last--;

    /* Fill the data as specified */
    for (i=1; i<=maxn/size; i++)
   for (j=0; j<maxn; j++)
       xlocal[i][j] = rank;
    for (j=0; j<maxn; j++) {
   xlocal[i_first-1][j] = -1;
   xlocal[i_last+1][j] = -1;
    }

    /* Send leftunless I am s I'm at the top, then
    receive from below */
    /* Note the use of xlocal[i] for &xlocal[i][0] */
    if (rank < size - 1)
        MPI_Send( xlocal[maxn/size], maxn, MPI_DOUBLE,
    rank + 1, 0,
                  MPI_COMM_WORLD );
    if (rank > 0)
        MPI_Recv( xlocal[0], maxn, MPI_DOUBLE, rank - 1,
    0,
                  MPI_COMM_WORLD, &status );

    /* Send down unless I'm at the bottom */
    if (rank > 0)
        MPI_Send( xlocal[1], maxn, MPI_DOUBLE, rank - 1,
    1,
                  MPI_COMM_WORLD );
    if (rank < size - 1)
        MPI_Recv( xlocal[maxn/size+1], maxn, MPI_DOUBLE,
    rank + 1, 1,
                  MPI_COMM_WORLD, &status );

    itcnt ++;
    for (i=i_first; i<=i_last; i++)
      for (j=1; j<maxn-1; j++) {
        xnew[i][j] = (xlocal[i][j+1] + xlocal[i][j-1] +
                              xlocal[i+1][j] + xlocal[i-
    1][j]) / 4.0;
      }


    MPI_Finalize( );
    return 0;
}
```

data declaration **initialization** **communication** **computation**

2D-stencil in Titanium

```
final static int DIM=2; //space dimension
final static Point<DIM> startPoint=Point<DIM>.all(0);
final static Point<DIM> endPoint=Point<DIM>.all(1)+Point<DIM>.direction(DIM,1);
public static single void main (String single [] single args){
  final int single numThreads=Ti.numProcs();
  final int threadID=Ti.thisProc();
  final RectDomain<DIM> problemDomain=[startPoint:endPoint];
  final int size=endPoint[DIM]-startPoint[DIM]+1;
  if (numThreads>size) System.exit(-1);
    final int localSize=size/numThreads;
    final Point<DIM> startPoint0=startPoint-Point<DIM>.direction(DIM,startPoint[DIM]);
    final Point<DIM> endPoint0=endPoint-Point<DIM>.direction(DIM,endPoint[DIM]);
    RectDomain<DIM> localDomain;
    //construct local domain
    if (threadID==numThreads-1){
      localDomain=[startPoint0+Point<DIM>.direction(DIM,localSize*threadID):endPoint];
    } else{
      localDomain=[startPoint0+Point<DIM>.direction(DIM,localSize*threadID):
               endPoint0+Point<DIM>.direction(DIM,localSize*(threadID+1)-1)];
    }
    //construct a distributed array
    double [1d] single local  [DIM d]  distArrayA=new double [0:numThreads-1]   [DIM d];
    double [DIM d] local localArrayA = new double [localDomain.accrete(1)]; //construct local subarray
    distArrayA.exchange(localArrayA); //exchange references to local subarray
    double [1d] single local  [DIM d]  distArrayB=new double [0:numThreads-1]   [DIM d];
    double [DIM d] local localArrayB = new double [localDomain]; //construct local subarray
    distArrayB.exchange(localArrayB); //exchange references to local subarray
    //initialize the array
    foreach(p in localDomain)
      localArrayA[p]=1;
    //exchange ghost values for distArrayA. The boundary values are zeroes by default.
    RectDomain<DIM> tempDomain;
    if (threadID>0){
      tempDomain=distArrayA[threadID-1].domain().shrink(1);
      localArrayA.copy(distArrayA[threadID-1].restrict(tempDomain));
    }
    if (threadID<numThreads-1){
      tempDomain=distArrayA[threadID+1].domain().shrink(1);
      localArrayA.copy(distArrayA[threadID+1].restrict(tempDomain));
    }
    Ti.barrier();
    //local stencil operation
    Point<DIM> disp=Point<DIM>.direction(DIM,1);
    foreach (p in localDomain) localArrayB[p]=(localArrayA[p-disp]+localArrayA[p+disp])*0.5;
}
```

code is rank-independent

data declaration

initialization

communication

computation

2D-stencil in UPC

```
shared [N] double a[M][N];
shared [N] double b[M][N];

int main() {
  int i, j;

  // initialize a
  upc_forall(i = 0; i < M; i++; continue)
    upc_forall(j = 0; j < N; j++; &a[i][j]) {
      a[i][j] = rand();
    }
  }
  upc_barrier();
  // exchange ghosts
  upc_forall(i = 0; i < M; i++; &b[i][0]) {
    b[i][0] = a[(i-1)%M][N-1];
    b[i][N] = a[(i+1)%M][1];
  }
  upc_barrier();
  // compute b
  upc_forall(i = 0; i < M; i++; continue)
    upc_forall(j = 1; j < N-1; j++; &b[i][j]) {
      b[i][j] = (a[i][j+1] + a[i][j-1])*0.5;
    }
  }
}
```

data declaration

initialization

communication

computation

2D-stencil in X10 (similar to NAS-MG)

```
public static void main(String[] args) {

  region R = [0:M, 0:N];
  region RInner = [1:M-1, 1:N-1];
  double[.] a = new double[R] (point p) { a[p] = Math.random(); };
  double[.] b = new double[R];

 finish foreach(point p[i] : RInner.rank(0))
    b[i,0] = a[(i-1)%M, N-1];
    b[i,N] = a[(i+1)%M, 1];

  finish foreach(point p[i,j] : RInner)
    b[i,j] = (a[i,j+1] + a[i,j-1])*0.5;

}
```

data declaration

initialization

communication

computation

2D stencil with C++ / htalib

data declaration

```
#include "htalib.h"

typedef HTA<double, 2, 0> H;
typedef Triplet R;

int main() {

  Tuple<2> tiling [] = {Tuple<2>(NPROC, 1), Tuple<2>(N/NPROC, M)};
  H a = H::alloc(tiling);
  H b = H::alloc(tiling);
```

initialization

```
  // initialize a
  a.map (Operator::rand(), a);
```

communication

```
  // exchange ghosts
  b()[0,R(0, M)]       = a(R((0:NPROC)%+1),0)[N/NPROC-1, R(0, M)];
  b()[NPROC/N,R(0, M)] = a(R((0:NPROC)%-1),0)[1, R(0, M)];
```

computation

```
  // compute b
  b() [R (1,N/NPROC-1), R (0,M)] =
    0.5 * ( a() [R(0,N/NPROC-2), R(0,M)] +
            a() [R(2,N/NPROC), R(0,M)] );

}
```

Tutorial outline

1) X10 Project

2) X10 Introduction
- cheat sheets
- Hello world
- comparison to Java

3) Sequential X10

4) Concurrency in X10
- activities
- atomic blocks
- clocks, clocked variables

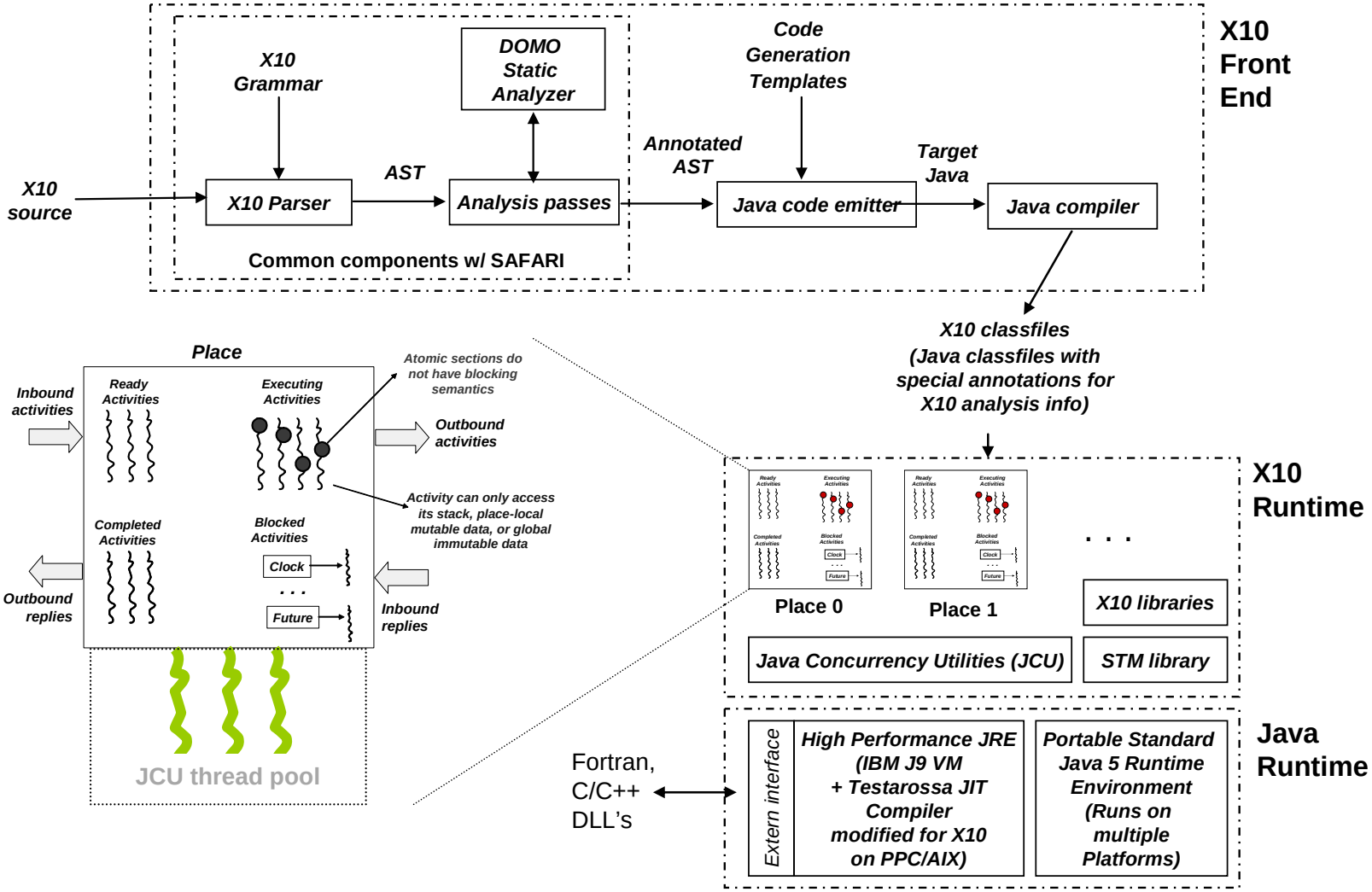
5) Distributed X10
- places
- distributions and distributed arrays

6) X10 Array Language

7) Current Status and Future Work

Current Status

Single Node SMP X10 Implementation



Current Status 07/2006

09/03 PERCS Kickoff

02/04 X10 Kickoff

07/04 X10 0.32 Spec Draft

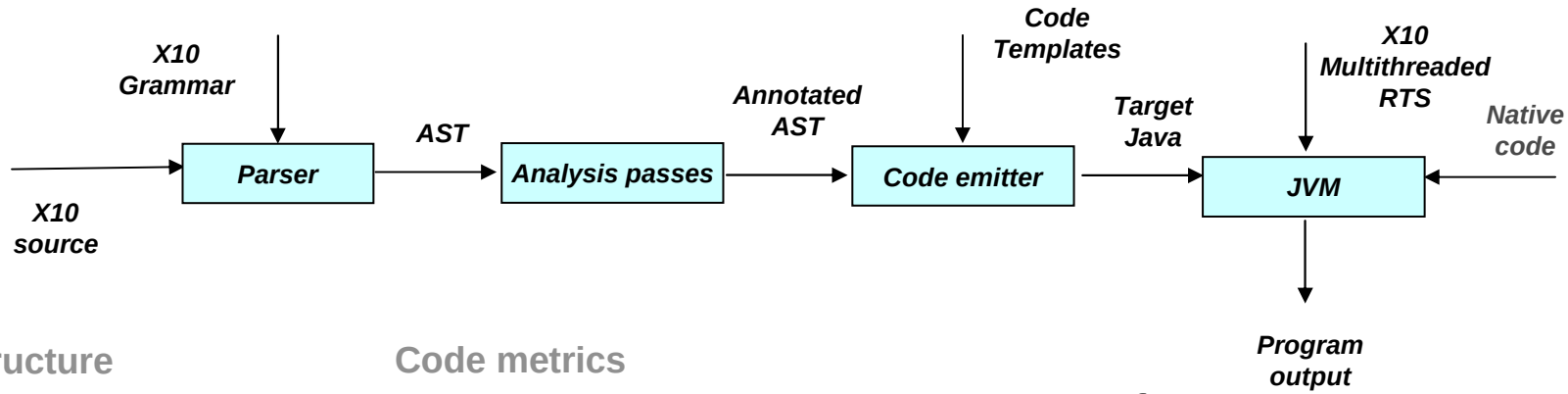
02/05 X10 Prototype #1

07/05 X10 Productivity Study

12/05 X10 Prototype #2

09/06 Open Source Release

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.

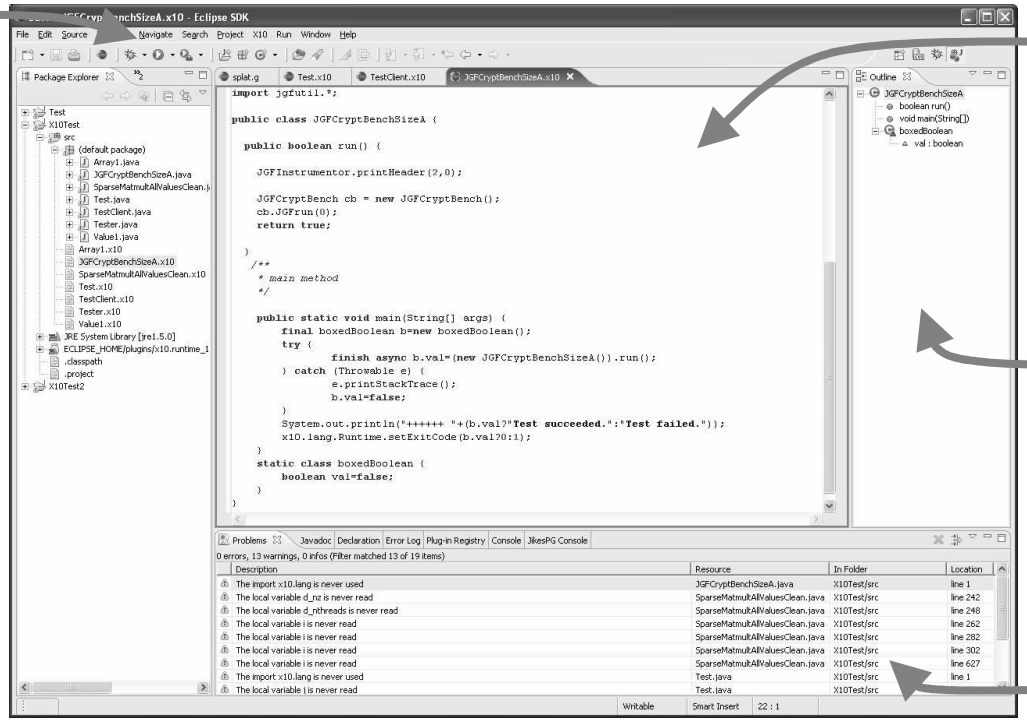
(* classes+interfaces/LOC)

New features

- Dependent types (places, arrays)
- Better codegen.
- Implicit syntax support.
- More functionality for points, arrays.

X10DT: Enhancing productivity

X10 Launch Configuration



Source editor w/ syntax highlighting, auto indenting, some content assist

Outline View populated w/ X10 types, members, loops

X10 Incremental Builder; Problems View populated w/ X10 compiler messages

- **Code editing**
- **Refactoring**
- **Code visualization**
- **Data visualization**
- **Debugging**
- **Static performance analysis**

Vision: State-of-the-art IDE for a modern OO language for HPC

X10 Applications/Benchmarks

- **Java Grande Forum**
  - OOPSLA Onwards! 2005
  - Showed substantial (SLOC) benefit in serial → parallel → distributed transition for X10 vs Java (qua C-like language).
- **SSCA**
  - SSCA#1 (PSC study)
  - SSCA#2 (Bader et al, UNM/GT)
  - SSCA#3 (Rabbah, MIT)
- **Sweep3d**
  - Jim Browne (UT Austin)
- **NAS PB**
  - CG, MG (IBM)
  - CG, FT, EP (Padua et al, UIUC)
  - Cannon, LU variant (UIUC)
- **AMR (port from Titanium)**
  - In progress, IBM
- **SpecJBB**
  - In progress, Purdue

Measures: SLOC as a "stand in" + process measures.

Advanced Topics

Dependent types

- **Class or interface that is a function of values.**
- **Programmer specifies properties of a type – public final instance fields.**
- **Programmer may specify refinement types as predicates on properties**
  - $T(v_1,\ldots,v_n : c)$
  - all instances of t with the values $f_i==v_i$ satisfying c.
  - c is a boolean expression over predefined predicates.

```
public class List( int(: n >=0) n) {
  this(:n>0) Object  value;
  this(:n>0) List(n-1) tail;
  List(t.n+1) (Object o, List t) {
        n=t.n+1; tail=t;value=o;}
  List(0) () { n = 0; }
  this(0) List(l.n) a(List l) {
    return l; }
  this(:n>0) List(n+l.n) a(List l) {
    return  new List(value, tail.append(l));
  }
  List(n+l.n) append(List l) {
    return n==0?
       this(0).a(l) : this(:n>0) .a(l);
  }
….
```

Place types

- **Every X10 reference inherits the property (place loc) from X10RefClass.**
- **The following types are permitted:**
  - Foo@? ➔ Foo
  - Foo ➔ Foo(: loc == here)
  - Foo@x ➔ Foo(: loc == x.loc)
- **Place types are checked by place-shifting operators (async, future).**

```
class Tree (boolean ll) {
  nullable<Tree>(:this.ll =>
                 (ll& loc==here))@? left;
  nullable<Tree> right;
  int node;
  Tree(l) (final boolean l,
    nullable<Tree>(:l =>
                 (ll&loc==here))@? left,
    nullable<Tree> right,
    int s) {
  ll=l; this.left=left; this.right=right;
  node=s;
  }
…
}
```

Region and distribution types (1/2)

```
abstract  value class point (nat rank) {
   type nat = int(: self >= 0) ;
   abstract static value class factory  {
     abstract point(val.length) point(final int[] val);
     abstract point(1) point(int v1);
     abstract point(2) point(int v1, int v2);
   … }
   …
   point(rank) (nat rank) { this.rank = rank; }
   abstract int get( nat(: i <= n) n);
   abstract boolean onUpperBoundary(region r,
                       nat(:i <= r.rank) i);
   abstract public boolean onLowerBoundary(region r,
                       nat(:i <= r.rank) i);
   abstract boolean gt( point(rank) p);
   abstract boolean lt( point(rank) p);
   abstract point(rank) mul( point(rank) p);
 …
```

Dependent types statically express many important relationships between data.

Region and distribution types (2/2)

```
class point (nat rank ) { ... }

class region (nat rank, boolean rect, boolean lowZero ) { ... }

class dist(nat rank, boolean rect, boolean lowZero,
  region(rank,rect,lowZero) region,
  boolean local, boolean safe ) { ... }

class Array<T>(nat rank, boolean rect,
   boolean lowZero,
   region(rank,rect,lowZero) region,
   boolean local, boolean safe,
   boolean(:self==(this.rank==1)&rect&lowZero&local) rail,
   dist(rank, rect, lowZero, region,local,safe) dist) { ... }

...
```

Dependent types statically express many important relationships between data.

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.**

Tiled regions

- **Tiled region (TR) is a region or an array (indexed by a region) of tiled regions.**

```
region(2) R = [1:N*K];
region(1:rect)[] S =
  new region[[1:K]]
    (point [i]){[(i-1)*N+1:I*N]};
region[] S1 = new region[]
    {[1:N],[N+1:2*N]};
```

- **Examples:**
  - Blocked, cyclic, block cyclic
  - Arbitrary, irregular cutsets

- **Tiled region is a tree with leaves labeled with regions.**
  - TR depth = depth of tree
  - TR uniform = all leaves at same depth
  - Tile = region labeling a leaf
  - Orthogonal TR = tiles do not overlap
  - Convex TR = each tile is convex.

- **A tiled region provides natural structure for distribution.**

User defined distributions

Open Issues and Future Work

Future Plans

- **X10 API in C, Java**
  - X10 Core Library
    - asyncs, future, finish, atomic, clocks, remote references
  - X10 Global Structures Library
    - Arrays, points, regions, distributions
- **Optimized SMP imp**
  - Locality-aware
  - Good single-thread perf.
  - Efficient inter-language calls
- **Annotations**
  - Externalized AST representation for source to source transformations.
  - Meta-language for programmers to specify their own annotations and transformations
- **SAFARI**
  - Support for annotations.
  - Support for refactorings
- **Application development**

HPC Landscape: 20K view

Our view!

Programming Technologies

		MPI + C/Fortran	C.OMP	ZPL	CAF	UPC	Ti	X10	HPL 2010?
Productivity	**Convenient?**	**X+**	√	**√?**	√–	√–	√–	√?	√+
	Global view?	**X**	**X**	√	√	√	√	√	√+
	Object-oriented?	**X**	**X**	**X**	**X**	**X**	√	√	√+
	Strong-typing?	√?	**X**	√?	√?	**X**	√	√+	√+
	Exceptions?	**X**	**X**	**X**	**X**	**X**	√	√+	√+
	Managed Runtime?	**X**	**X**	**X**	**X**	**X**	√–	√+	√+
Perf	**Perf Transparency**	√	√	√	√	√	√	√?	√+
	Perf Portability	√	**X**	√	√	√?	√	√?	√+
	Perf Scalability	√	**X**	√	√	√	√?	√?	√+
Expr.	**Data-structures?**	**X**	√	**X**	**X**	√	√	√+	√+
	Explicit parallelism?	√	√	**X**	√	√	√	√+	√+
	Task parallelism?	**X**	√		**X**	**X**	**X**	√+	√+
	Fork-join parallelism?		√					√+	√+