Programming Technologies

X10: Computing at scale

Vijay Saraswat
July 24, 2006
IBM Research

What?

- **Context and overview**
- **The X10 Programming Model**
- **Programming in X10**
- **Research topics**

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

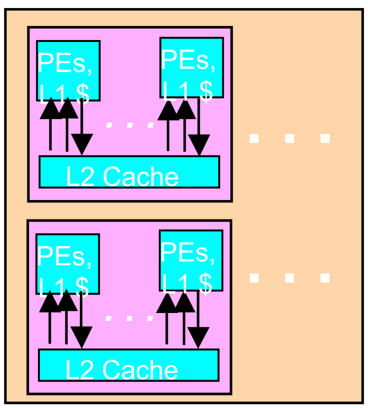
- PACT 2006, OOPSLA 2006

Programming Technologies

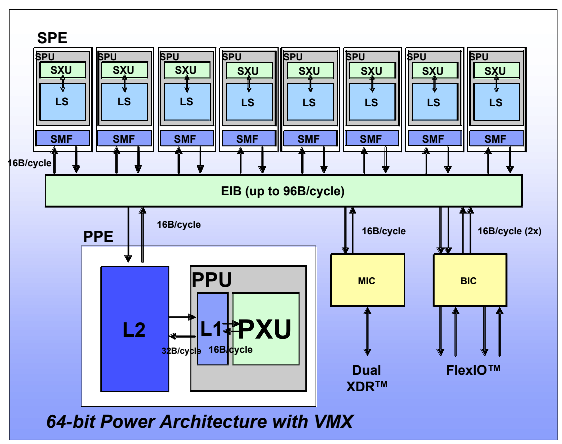
A new Era of Mainstream Parallel Processing

<u>The Challenge:</u>
Parallelism scaling replaces frequency scaling as foundation for increased performance ➔ Profound impact on future software

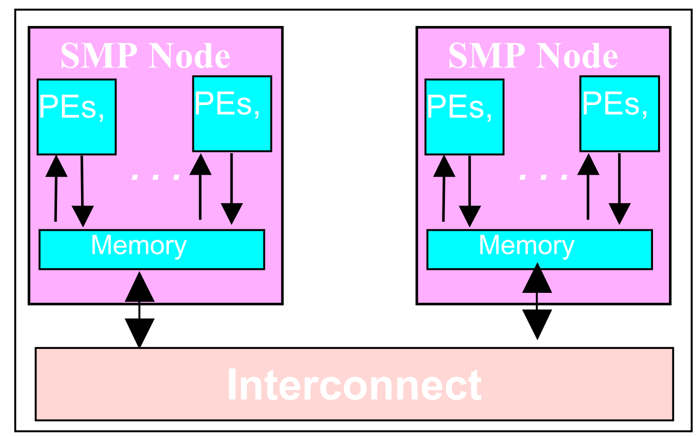
Multi-core chips



Heterogeneous Parallelism



Cluster Parallelism



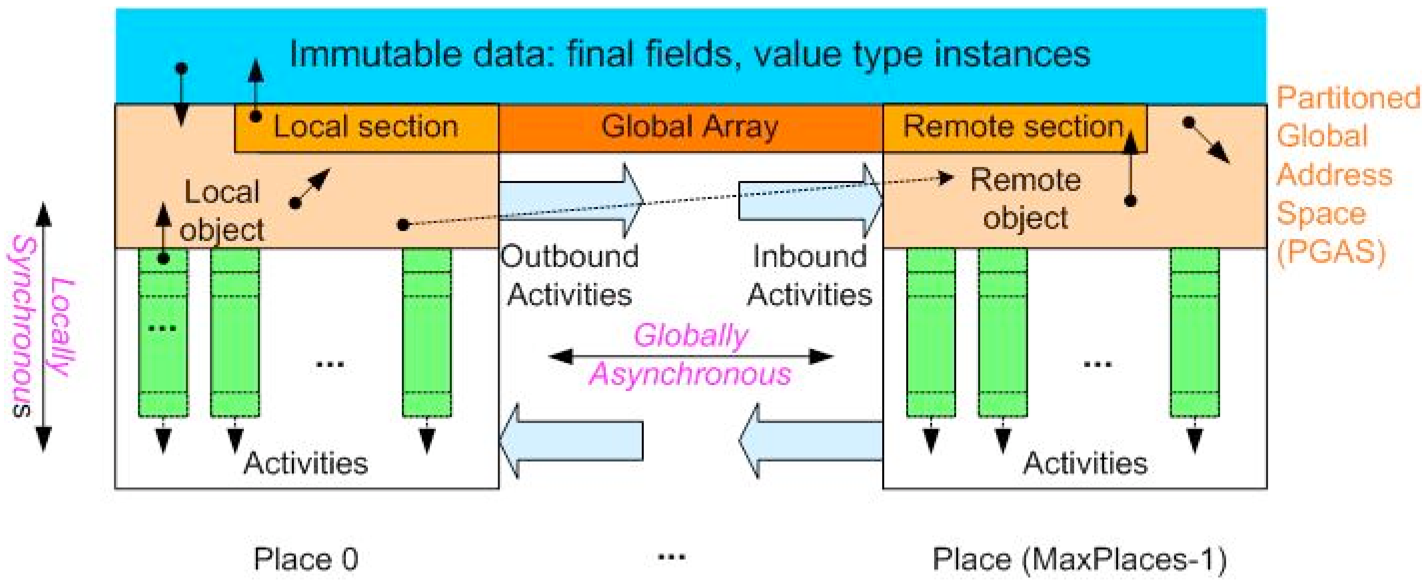
<u>Our response:</u>
Use X10 as a new language for parallel hardware that builds on existing tools, compilers, runtimes, virtual machines and libraries

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:** Itegrate 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.

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

Few concepts, done right.

Programming Technologies

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

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

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.

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.

Hello, World!

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

```
public class HelloWorld2 {
  public static void main(String[] args) {
    finish foreach(point [p] : [1:10])
      System.out.println("Hello, world from async " + p + "!");
  }
}
```

Programming Technologies

Value types : immutable instances

- **Value class**
  - Can only extend value class or x10.lang.Object.
  - Have only final fields
  - Can only be extended by value classes.
  - May contain fields at reference type.
  - May be implemented by reference or copy.
- **Two values are equal if their corresponding fields are equal.**

- **nullable _ provided as a type constructor.**

```
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);
 } … }
```

async, finish

Programming Technologies

async *PlaceExpressionSingleListopt Statement*

- async (P) S
  - **Parent activity creates a new child activity at place P, to execute statement S; returns immediately.**
  - **S may reference *final* variables in enclosing blocks.**

```
double[D] A =…;  // Global dist. array
final int k = …;
async ( A.distribution[99] ) {
   // Executed at A[99]'s place
   atomic A[99] = k;
}
```

Statement ::= finish Statement

- finish S
  - Execute S, but wait until all (transitively) spawned async's have terminated.
  - Trap all exceptions thrown by spawned activities, throw aggregate exception when all activities terminate.

```
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 spawn, sync

atomic, when

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

- **Atomic blocks are**
  - Executed in a single step, conceptually, while other activities are suspended.
- **An atomic block may not**
  - Block
  - Access remote data.
  - Create activities.
  - Contain a conditional block.
- **Essentially, body is a bounded, sequential, non-blocking activity**
  - Hence executing in a single place.

Statement ::= *WhenStatement*
WhenStatement ::= when (*Expression*) *Statement*

- **Conditional atomic blocks**
  - Activity suspends until a state in which guard is true; in that state it executes body atomically.
- **Body has same restructions as unconditional atomic block.**
- `await(e)=def=when(e);`
- **X10 does not assume retry semantics for atomics.**

X10 has only one synchronization construct: conditional atomic block.

Programming Technologies

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

Aside: Memory Model

- **X10 v 0.41 specifies sequential consistency per place.**
- **We are considering a weaker memory model.**
- **Built on the notion of atomic: identify a step as the basic building block.**
- **A process is a pomset of steps closed under certain transformations:**
  - Composition
  - Decomposition
  - Augmentation
  - Linking
- **There may be opportunity for a weak notion of atomic: decouple *atomicity* from *ordering*.**

Bounded buffer

```
class OneBuffer {
    nulable Object datum = null;
    public void send( Object v) {
        when (datum == null) {
            datum = v;
        }
    }

    public Object receive() {
        when (datum != null) {
            Object v = (Object) datum;
            datum = null;
            return v;
        }
    }
}
```

Atomic examples: future

```
class Latch implements future {
  boolean forced = false;
  nullable boxed result = null;
  nullable exception z = null;
  atomic boolean set( nullable Object val ) {
     return set( val, null); }
  atomic boolean set( nullable Exception z ) {
      return set( null, z); }
  atomic boolean set( nullable Object val,
                      nullable Exception z ) {
       if ( forced ) return false;
       // these assignment happens only once.
       this.result = val;
       this.z = z;
       this.forced = true;
       return true; }
```

```
atomic boolean forced() {
   return forced;
}
Object force() {
   when ( forced ) {
     if (z != null)
       throw z;
      return result;
    }
 }
}
```

Atomic examples: future

```
future  (P)  { e}
```

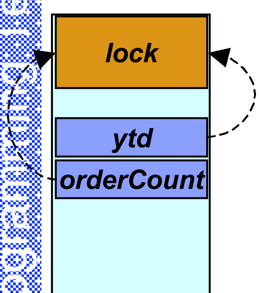
```
new RunnableLatch() {
    public Latch run() {
        Latch L = new Latch();
        async ( P ) {
            Object X;
            try {
                finish X = e;
                async ( L )  L.setValue( X );
            } catch ( exception Z ) {
                async ( L )  L.setValue( Z );
            }
        }
        return l;
    }
}.run()
```

Atomic Blocks: SPECjbb Example #2

Java:

```
public class Stock extends Entity {…
private float  ytd;
private short orderCount; …
public synchronized void
  incrementYTD(short ol_quantity) { …
      ytd += ol_quantity; …}…
public synchronized void
  incrementOrderCount() { …
      ++orderCount; …} …
}
```

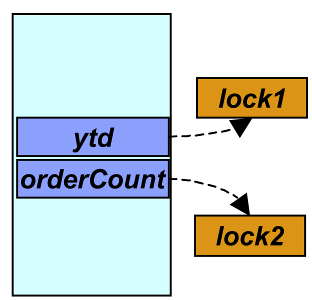
Layout of a “Stock” object



NOTE: these two methods cannot be executed simultaneously because they use the same lock

X10:

```
public class Stock extends Entity {…
private float  ytd;
private short orderCount; …
public atomic void
  incrementYTD(short ol_quantity) { …
      ytd += ol_quantity; …}…
public atomic void
  incrementOrderCount() { …
      ++orderCount; …} …
}
```



NOTE: with atomic blocks, these two methods can be executed simultaneously

Atomic blocks: 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) {…}
} …
```

EQUIVALENT CODE WITH ATOMIC SECTIONS

Main thread: …

```
// Wait for all threads to start.
when(company.initThreadsCount==thre
    adCount) {
    mode = RAMP_UP;
    initThreadsCountReached = true;
}  …
```

Worker thread: …

```
atomic {
    company.initThreadsCount++;
}

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

Determinate, dynamic barriers: clocks

- Operations

  clock c = new clock();

  c.resume();

  - Signals completion of work by activity in this clock phase.

  next;

  - Blocks until **all** clocks it is registered on can advance. Implicitly resumes all clocks.

  c.drop();

  - Unregister activity with **c**.

async(P)clocked($c_1,\ldots,c_n$)S

- (Clocked async): activity is registered on the clocks **($c_1,\ldots,c_n$)**

- Static Semantics
  - An activity may operate only on those clocks for which it is live.
  - 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().**

No explicit operation to register a clock.

Supports over-sampling, hierarchical nesting.

Programming Technologies

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.

Programming Technologies

Programming Technologies

Synchronous Kahn networks are CF (and DD-free)

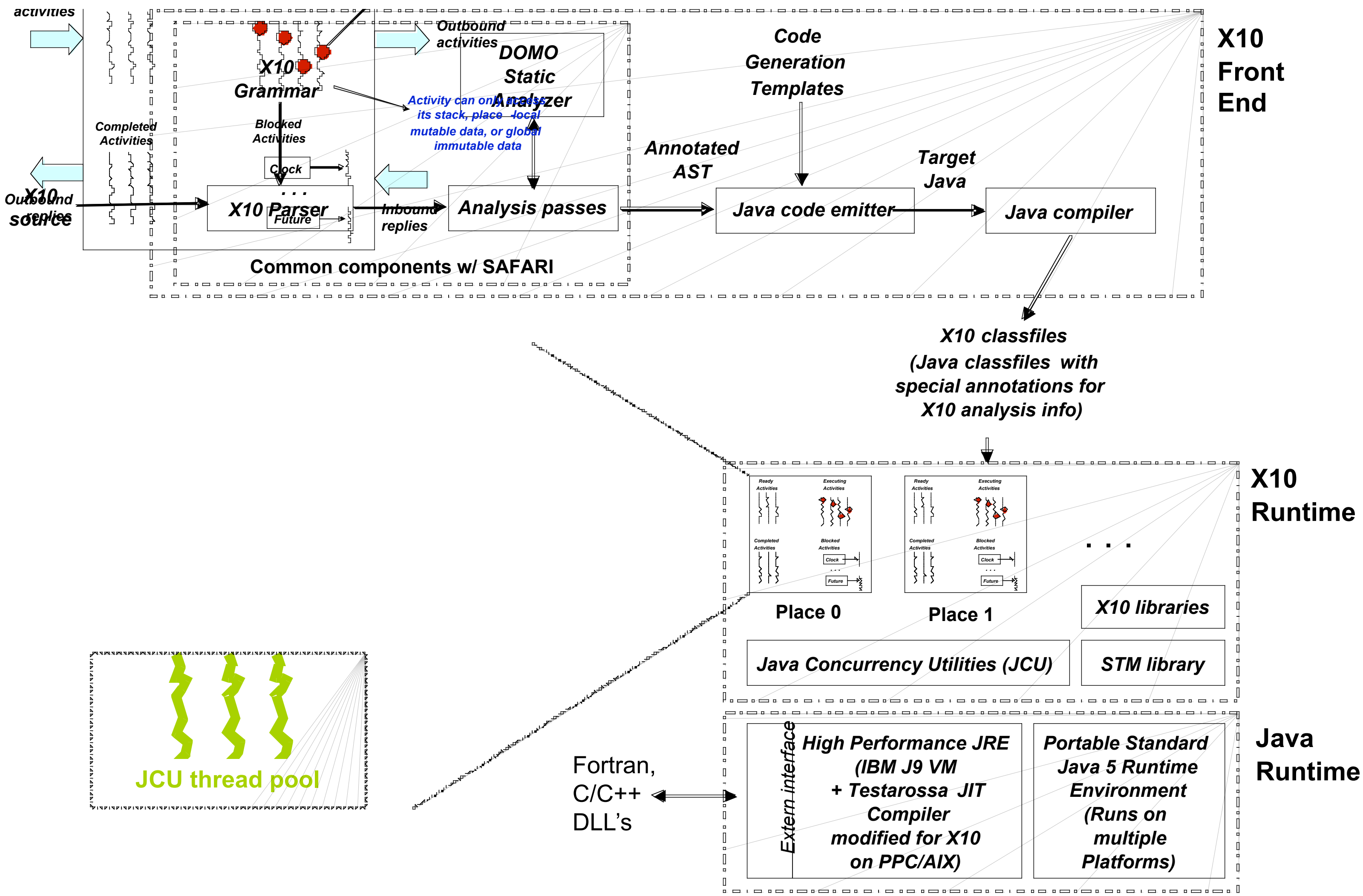
- **This idea may be generalized to arbitrary mutable variables.**
  - Determinate imperative programming.
- **Each variable has an implicit clock.**
- **Each variable has a stream of values.**
- **Each activity maintains its own index into stream.**
- **An activity performs reads/writes per its index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

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

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

Current Status

Single Node SMP X10 Implementation



Programming Technologies

Current Status 07/2006

Programming Technologies

09/03
PERCS Kickoff

02/04
X10 Kickoff

07/04
X10 0.32 Spec Draft

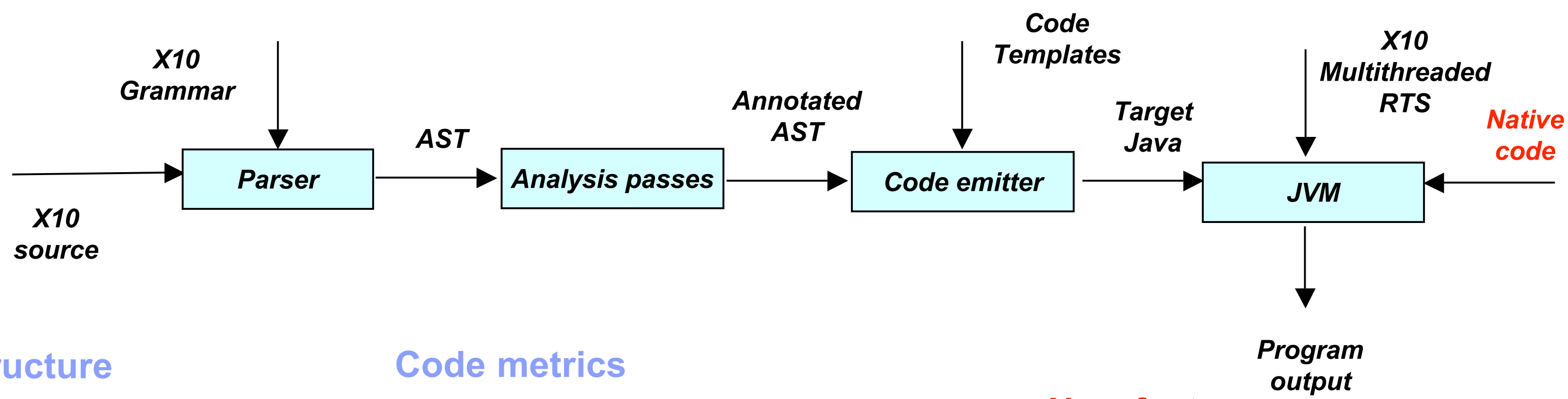
02/05
X10 Prototype #1

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.

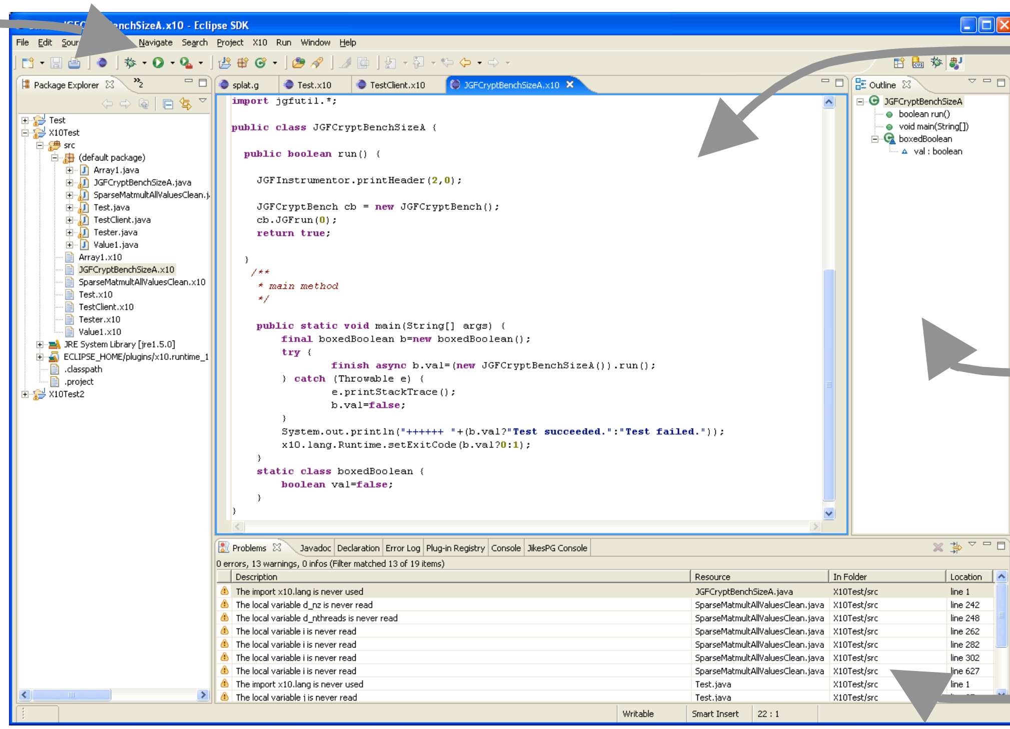
(* classes+interfaces/LOC)

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 (IBM)
  - 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.

Programming Technologies

Arrays

regions, distributions

- Region
  - a (multi-dimensional) set of indices
- Distribution
  - A mapping from indices to places
- High level algebraic operations are provided on regions and distributions

```
region R = 0:100;
region R1 = [0:100, 0:200];
region RInner = [1:99, 1:199];
// a local distribution
dist D1=R-> here;
// a blocked distribution
dist D = block(R);
// union of two distributions
dist D = (0:1) -> P0 || (2:N) -> P1;
dist DBoundary = D – RInner;
```

Based on ZPL

arrays

- Arrays may be
  - Multidimensional
  - Distributed
  - Value types
  - Initialized in parallel:

```
int [D] A= new int[D]
   (point [i,j]) {return N*i+j;};
```

- Array section
  - **A[RInner]**
- High level parallel array, reduction and span operators
  - Highly parallel library implementation
  - A-B (array subtraction)
  - A.reduce(intArray.add,0)
  - A.sum()

Ateach, foreach

- ateach (point p : A) S
  - Creates |region(A)| async statements
  - Instance p of statement S is executed at the place where A[p] is located
- foreach (point p : R) S
  - Creates |R| async statements in parallel at current place
- Termination of all activities can be ensured using finish.

```
ateach ( FormalParam: Expression ) Statement
foreach ( FormalParam: Expression ) Statement
```

```
public boolean run() {
  dist D = dist.factory.block(TABLE_SIZE);
  long[.] table = new long[D] (point [i]) { return i; }
  long[.] RanStarts = new long[distribution.factory.unique()]
      (point [i]) { return starts(i);};
  long[.] SmallTable = new long value[TABLE_SIZE]
     (point [i]){return i*S_TABLE_INIT;};
  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 atomic table[J] ^= K;
     ran = nextRandom(ran);
    }}
return table.sum() == EXPECTED_RESULT;
}
```

Programming Technologies

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 tasks array (of size nRunsMC) is initialized withToTask 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

```
dist D = [0:(nRunsMC-1)] -> here;
double[.] expectedReturnRate = new double[D];
//….
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();
}
```

- A tasks array (of size nRunsMC) is initialized withToTask 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 -- 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();
}
```

- A tasks array (of size nRunsMC) is initialized withToTask instances at each index.
- Task:
  - Simulate stock trajectory,
  - Compute expected rate of return and volatility,
  - Report average expected rate of return and volatility.

RandomAccess

```
public boolean run() {
  dist D = dist.factory.block(TABLE_SIZE);
  long[.] table = new long[D] (point [i]) { return i; }
  long[.] RanStarts = new long[dist.factory.unique()]
       (point [i]) { return starts(i);};
  long[.] SmallTable = new long value[TABLE_SIZE]
       (point [i]) {return i*S_TABLE_INIT;};
  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 atomic table[J] ^= K;
      ran = nextRandom(ran);
    }
  }
return table.sum() == EXPECTED_RESULT;
}
```

Allocate and initialize table as a block-distributed array.

Allocate and initialize RanStarts with one random number seed for each place.

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

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

Jacobi

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);
  }
….
```

Programming Technologies

Programming Technologies

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

```
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);
 …
```

```
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.

Programming Technologies

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

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! (→ HPL 2010?)

Programming Technologies

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