Empower. Partner. Lead.

Ohio Supercomputer Center

Module 1: X10 Overview

Dave Hudak
Ohio Supercomputer Center
“The X10 Language and Methods for Advanced HPC Programming”

Module Overview

- Workshop goals
- Partitioned Global Address Space (PGAS) Programming Model
- X10 Project Overview
- My motivation for examining X10
- X10DT (briefly)

Workshop Goals and Prerequisites

- Provide rudimentary programming ability in X10
  - You won't be an expert, but you won't be baffled when presented with code
- Describe X10 approaches for multilevel parallelism through code reuse

Workshop Prerequisites

- Experience with parallel programming, either MPI or OpenMP.
- Basic knowledge of Java (e.g., objects, messages, classes, inheritance).
  - Online tutorials are available at http://java.sun.com/docs/books/tutorial/
  - The “Getting Started” and “Learning the Java Language” tutorials are recommended.
- Familiarity with basic linear algebra and matrix operations.

PGAS Background: Global and Local Views

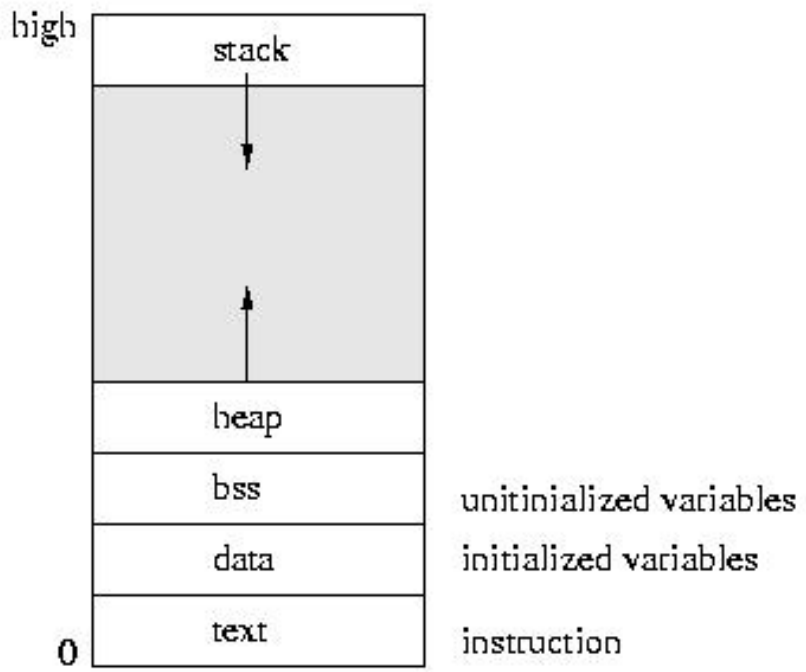
- A parallel program consists of a set of threads and at least one address space
- A program is said to have a global view if all threads share a single address space (e.g., OpenMP)
  - Tough to see when threads share same data
  - Bad data sharing causes race conditions (incorrect answers) and communication overhead (poor performance)
- A program is said to have a local view if the threads have distinct address spaces and pass messages to communicate (e.g., MPI)
  - Message passing code introduces a lot of bookkeeping to applications
  - Threads need individual copies of all data required to do their computations (which can lead to replicated data)

PGAS Overview

- "Partitioned Global View" (or PGAS)
  - Global Address Space: Every thread sees entire data set, so no need for replicated data
  - Partitioned: Divide global address space so programmer is aware of data sharing among threads
- Implementations
  - GA Library from PNNL
  - Unified Parallel C (UPC), FORTRAN 2009
  - X10, Chapel
- Concepts
  - Memories and structures
  - Partition and mapping
  - Threads and affinity
  - Local and non-local accesses
  - Collective operations and "Owner computes"

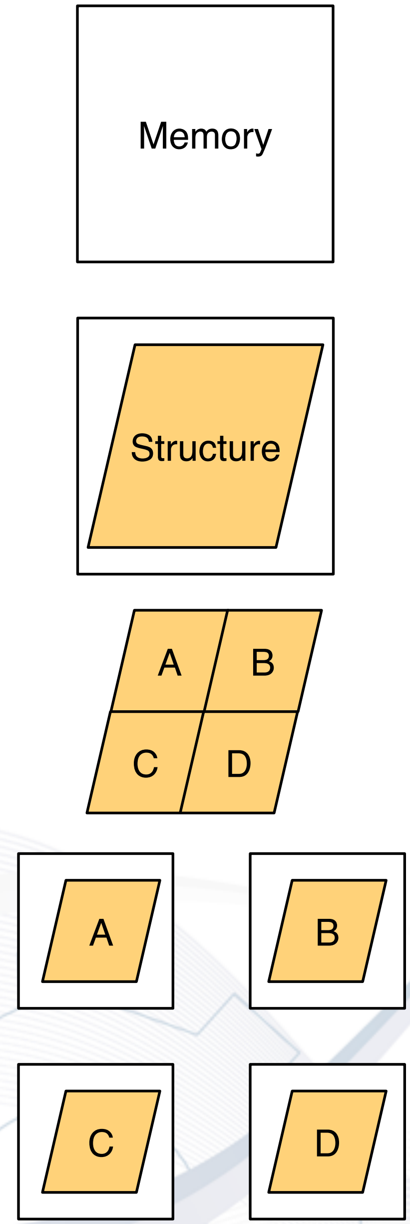
Software Memory Examples

- Executable Image at right
  - "Program linked, loaded and ready to run"
- Memories
  - Static memory
    - data segment
  - Heap memory
    - Holds allocated structures
    - Explicitly managed by programmer (malloc, free)
  - Stack memory
    - Holds function call records
    - Implicitly managed by runtime during execution



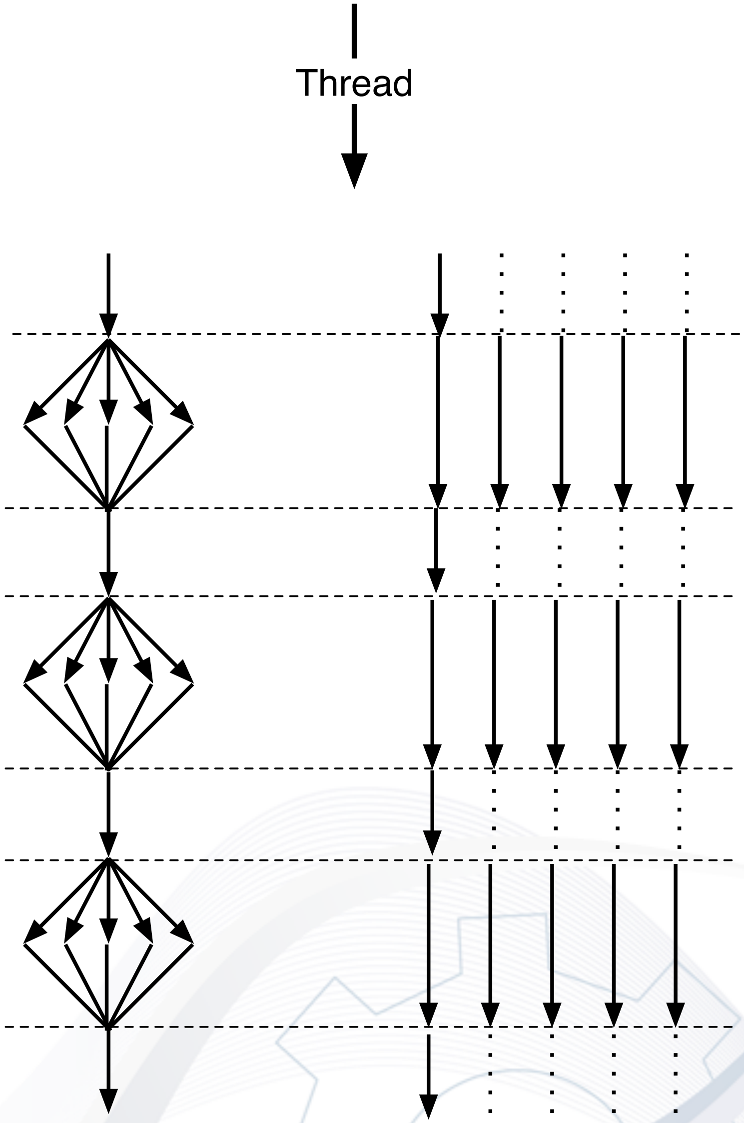
Memories and Distributions

- Software Memory
  - Distinct logical storage area in a computer program (e.g., heap or stack)
  - For parallel software, we use multiple memories
- In X10, a memory is called a place
- Structure
  - Collection of data created by program execution (arrays, trees, graphs, etc.)
- Partition
  - Division of structure into parts
- Mapping
  - Assignment of structure parts to memories
- In X10, partitioning and mapping information for an array are stored in a distribution



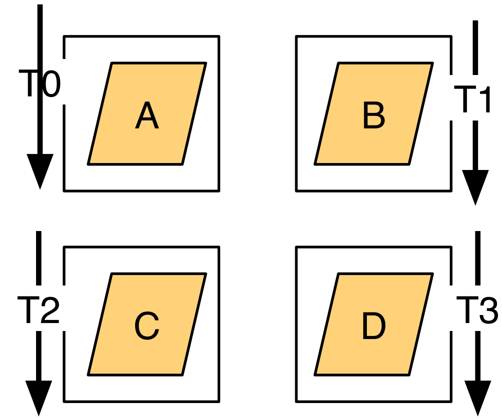
Threads

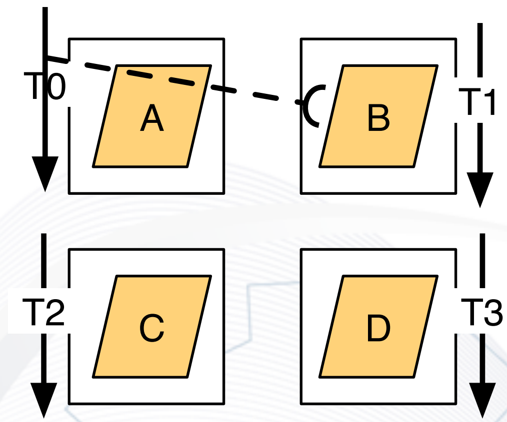
- Units of execution
- Structured threading
  - Dynamic threads: program creates threads during execution (e.g., OpenMP parallel loop)
  - Static threads: same number of threads running for duration of program
    - Single program, multiple data (SPMD)
- Threads in X10 (activities) are created with async and at



Affinity and Nonlocal Access

- Affinity is the association of a thread to a memory
  - If a thread has affinity with a memory, it can access its structures
  - Such a memory is called a local memory
- Nonlocal access
  - Thread 0 wants part B
  - Part B in Memory 1
  - Thread 0 does not have affinity to memory 1
- Nonlocal accesses often implemented via interprocess communication – which is expensive!





Collective operations and "Owner computes"

- Collective operations are performed by a set of threads to accomplish a single global activity
  - For example, allocation of a distributed array across multiple places
- "Owner computes" rule
  - Distributions map data to (or across) memories
  - Affinity binds each thread to a memory
  - Assign computations to threads with "owner computes" rule
    - Data must be updated (written) by a thread with affinity to the memory holding that data

Threads and Memories for Different Programming Methods

	Thread Count	Memory Count	Nonlocal Access
Sequential	1	1	N/A
OpenMP	Either 1 or p	1	N/A
MPI	p	p	No. Message required.
CUDA	1 (host) + p (device)	2 (Host + device)	No. DMA required.
UPC, FORTRAN	p	p	Supported.
X10	n	p	Supported.

X10 Overview

- X10 is an instance of the Asynchronous PGAS model in the Java family
  - Threads can be dynamically created under programmer control (as opposed to SPMD execution of MPI, UPC, FORTRAN)
  - n distinct threads, p distinct memories (n <> p)
- PGAS memories are called places in X10
- PGAS threads are called activities in X10
- Asynchronous extensions for other PGAS languages (UPC, FORTRAN 2009) entirely possible…

X10 Project Status

- X10 is developed by the IBM PERCS project as part of the DARPA program on High Productivity Computing Systems (HPCS)
- Target markets: Scientific computing, business analytics
- X10 is an open source project (Eclipse Public License)
  - Documentation, releases, mailing lists, code, etc. all publicly available via http://x10-lang.org
- X10 2.1.0 released October 19, 2010
  - Java back end: Single process (all places in 1 JVM)
    - any platform with Java 5
  - C++ back end: Multi-process (1 place per SMP node)
    - aix, linux, cygwin, MacOS X
    - x86, x86_64, PowerPC, Sparc

X10 Goals

- Simple
  - Start with a well-accepted programming model, build on strong technical foundations, add few core constructs
- Safe
  - Eliminate possibility of errors by design, and through static checking
- Powerful
  - Permit easy expression of high-level idioms
  - And permit expression of high-performance programs
- Scalable
  - Support high-end computing with millions of concurrent tasks
- Universal
  - Present one core programming model to abstract from the current plethora of architectures.

From "An Overview of X10 2.0", SC09 Tutorial

X10 Motivation

- Modern HPC architectures combine products
  - From desktop/enterprise market: processors, motherboards
  - HPC market: interconnects (IB, Myrinet), storage, packaging, cooling
- Computing dominated by power consumption
  - In desktop/enterprise market emergence of multicore
    - HPC will retain common processor architecture with enterprise
  - In HPC, we seek even higher flops/watt. Manycore is leading candidate
    - nVidia Fermi: 512 CUDA cores
    - Intel Knights Corner: >50 Cores, (Many Integrated Core) MIC Architecture (pronounced “Mike”)

X10 Motivation

- HPC node architectures will be increasingly
  - Complicated (e.g., multicore, multilevel caches, RAM and I/O contention, communication offload)
  - Heterogenous (e.g, parallelism across nodes, between motherboard and devices (GPUs, IB cards), among CPU cores)
- Programming Challenges
  - exhibit multiple levels of parallelism
  - synchronize data motion across multiple memories
  - regularly overlap computation with communication

Every parallel architecture has a dominant programming model

Parallel Architecture	Programming Model
Vector Machine (Cray 1)	Loop vectorization (IVDEP)
SIMD Machine (CM-2)	Data parallel (C*)
SMP Machine (SGI Origin)	Threads (OpenMP)
Clusters (IBM 1350)	Message Passing (MPI)
GPGPU (nVidia Fermi)	Data parallel (CUDA)
Accelerated Clusters	Asynchronous PGAS?

- Software Options
  - Pick existing model (MPI, OpenMP)
    - Kathy Yelick has interesting summary of challenges here
  - Hybrid software
    - MPI at node level
    - OpenMP at core level
    - CUDA at accelerator
  - Find a higher-level abstraction, map it to hardware

Conclusions

- PGAS fundamental concepts:
  - Data: Memory, partitioning and mapping
  - Threads: Static/Dynamic, affinity, nonlocal access
- PGAS models expose remote accesses to the programmer
- X10 is a general-purpose language providing asynchronous PGAS
- Asynchronous PGAS may be a unified model to address the upcoming changes in petascale and exascale architectures

Empower. Partner. Lead.

Ohio Supercomputer Center

Module 2: X10 Base Language

Dave Hudak
Ohio Supercomputer Center
"The X10 Language and Methods for Advanced HPC Programming"

Module Overview

- How this tutorial is different
- X10 Basics, Hello World, mathematical functions
- Classes and objects
- Functions and closures
- Arrays
- Putting it all together: Prefix Sum example

How this tutorial is different

- Lots of other X10 materials online
  - Mostly language overviews and project summaries
- Best way to learn a language is to use it
  - Focus on working code examples and introduce language topics and constructs as they arise
- Focus on HPC-style numeric computing
- Won't exhaustively cover features of the language
  - Interfaces, exceptions, inheritance, type constraints, …
- Won't exhaustively cover implementations
  - Java back end, CUDA interface, BlueGene support, …

X10 Basics

- X10 is an object-oriented language based on Java
- Base data types
  - Non-numeric: Boolean, Byte, Char and String
  - Fixed point: Short, Int and Long
  - Floating point: Float, Double and Complex
- Top level containers: classes and interfaces, grouped into packages
- Objects are instantiated from classes

Hello World

```
public class Hello {
  public static def main(var args: Array[String](1)):Void {
    Console.OUT.println("Hello X10 world");
  }
}
```

- Program execution starts with `main()` method
  - Only one class can have a main method
- Method declaration
  - Methods declared with `def`
  - Objects fields either methods (function) or members (data):
    - Access modifiers: `public`, `private` (like Java)
    - `static` declaration: field is contained in class and is immutable
  - Function return type here is `Void`
- I/O provided by library x10.io.Console

Hello World

```
public class Hello {
  public static def main(var args: Array[String](1)):Void {
    Console.OUT.println("Hello X10 world");
  }
}
```

- Variable Declarations: `var <name> : <type>`, like `var x:Int`
- Example of generic types (similar to templates)
  - Array (and other data structures) take a base type parameter
  - For example `Array[String]`, `Array[Int]`, `Array[Double]`, …
- Also, we provide dimension of Array, so Array [String](1) is a single-dimensional array of strings

Types in X10

```
public class MathTest {
  public static def main(args: Array[String](1)):Void {
    val w = 5;
    val x = w as Double;
    val y = 3.0;
    val z = y as Int;
    Console.OUT.println("w = " +w+ ", x = " +x+ ", y = " +y+ ", z = " +z);
    val d1 = (Math.log(8.0)/Math.log(2.0)) as Int;
    val d2 = Math.pow(2, d1) as Int;
    Console.OUT.println("d1 = " + d1 + ", d2 = " + d2);
  }
}
```

- X10 type casting (coercion) using `as`
- Calculate $\log_2$ of a number using $\log_{10}$
- X10 math functions provided by Math library
- val – declares a value (immutable)
  - Type inference used to deduce type, no declaration needed
  - X10 community says var/val = Java's non-final/final
- Declare everything val unless you explicitly need var
  - Let the type system infer types whenever possible

```
public class Counter {
  var counterValue:Int;

  public def this() {
    counterValue = 0;
  }

  public def this(initValue:Int) {
    counterValue = initValue;
  }

  public def count() {
    counterValue++;
  }

  public def getCount():Int {
    return counterValue;
  }
}
```

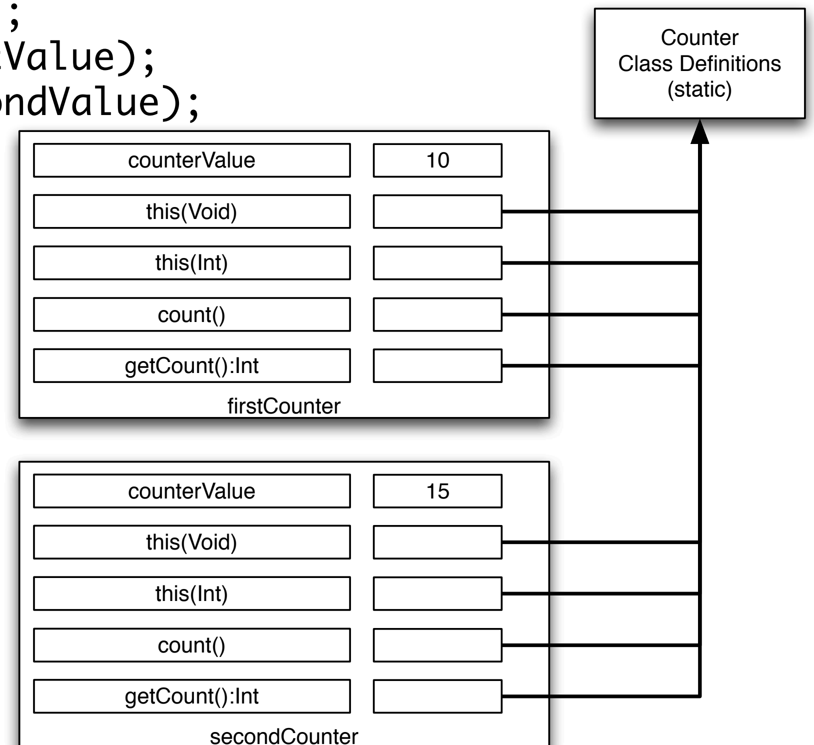
Classes

- Instance declarations allocated with each object (e.g., counterValue)
- Class declarations allocated once per class
  - `static`
- `this`
  - val containing reference to lexically enclosing class
    - Here, it is Counter
  - Constructors automatically called on object instantiation
    - In Java, use Counter(), in X10, use this()

Objects

```
class Driver {
  public static def main(args:Array[String](1)):Void {
    val firstCounter = new Counter();
    val secondCounter = new Counter(5);
    for (var i:Int=0; i<10; i++) {
      firstCounter.count();
      secondCounter.count();
    }
    val firstValue = firstCounter.getCount();
    val secondValue = secondCounter.getCount();
    Console.OUT.println("First value = "+firstValue);
    Console.OUT.println("Second value = "+secondValue);
  }
}
```

- Object instantiation with new
  - firstCounter uses default constructor, secondCounter uses initialization constructor
  - X10 has garbage collection, so no malloc/free. Object GC'ed when it leaves scope
- Example of C-style for loop
  - Modifying i, so use var



Arrays

```
public class Driver {
  public static def main(args: Array[String](1)): Void {
    val arraySize = 12;
    val regionTest = 1..arraySize;
    val testArray = new Array[Int](regionTest, (Point)=>0);
    for ([i] in testArray) {
      testArray(i) = i;
      Console.OUT.println("testArray("+i+") = " + testArray(i));

    }
    val p = [22, 55];
    val [i, j] = p;
```

- Points – used to access arrays, e.g., [5], [1,2]
  - i and j assigned using pattern matching (i = 22, j = 55)
- Regions – collection of points
  - One-dimensional 1..arraySize, Two-dimensional [1..100, 1..100]
- `Array` constructor requires:
  - Region (1..arraySize)
  - Initialization function to be called for each point in array (Point)=>0
- For loop runs over region of array
  - [i] is a pattern match so that i has type Int

Functions

```
public class Driver {
  public static def main(args: Array[String](1)): Void {
    val arraySize = 12;
    val regionTest = 1..arraySize;
    val testArray = new Array[Int](regionTest, (Point)=>0);
    for ([i] in testArray) {
      testArray(i) = i;
      Console.OUT.println("testArray("+i+") = " + testArray(i));

    }
```

- Anonymous function: `(Point)=>0`
  - Function with no name, just input type and return expression
  - Also called a function literal
- Functions are first-class data – they can be stored in lists, passed between activities, etc.
  - `val square = (i:Int) => i*i;`
- Anonymous functions implemented by creation and evaluation of a closure
  - An expression to be evaluated along with all necessary values
  - Closures very important under the hood of X10!

Prefix Sum Object

```
public class Driver {
  public static def main(args: Array[String](1)): Void {
    val arraySize = 5;
    Console.OUT.println("PrefixSum test:");
    val psObject = new PrefixSum(arraySize);
    val beforePS = psObject.str();
    Console.OUT.println("Initial array:  "+beforePS);
    psObject.computeSum();
    val afterPS = psObject.str();
    Console.OUT.println("After prefix sum:  "+afterPS);
  }
}
```

```
PrefixSum test:
Initial array:  1, 2, 3, 4, 5
After prefix sum:  1, 3, 6, 10, 15
```

- Prefix Sum definition
  - Given a[1], a[2], a[3], … a[n]
  - Return a[1], a[1]+a[2], a[1]+a[2]+a[3], …, a[1]+...+a[n]
- Example: PrefixSum object
  - Object holds an array
  - Methods include constructor, computeSum and str
- Used as an educational example only
  - In real life, you'd use X10's built-in `Array.scan()` method

Prefix Sum Class

```
public class PrefixSum {

  val prefixSumArray: Array[Int](1);

  public def this(length:Int) {
    prefixSumArray = (new Array[Int](1..length, (Point)=>0));
    for ([i] in prefixSumArray) {
      prefixSumArray(i) = i;
    }
  }
  public def computeSum()
  {
    for ([i] in prefixSumArray) {
      if (i != 1) {
        prefixSumArray(i) = prefixSumArray(i) + prefixSumArray(i-1);
      }
    }
  }
}
```

- Full code in example
- prefixSumArray is an instantiation variable, and local to each PrefixSum object
- `this` – initialization constructor creates array
- computeSum method – runs the algorithm

Conclusions

- X10 has a lot of ideas from OO languages
  - Classes, objects, inheritance, generic types
- X10 has a lot of ideas from functional languages
  - Type inference, anonymous functions, closures, pattern matching
- X10 is a lot like Java
  - Math functions, garbage collection
- Regions and points provide mechanisms to declare and access arrays

Empower. Partner. Lead.

Ohio Supercomputer Center

Module 3: X10 Intra-Place Parallelism

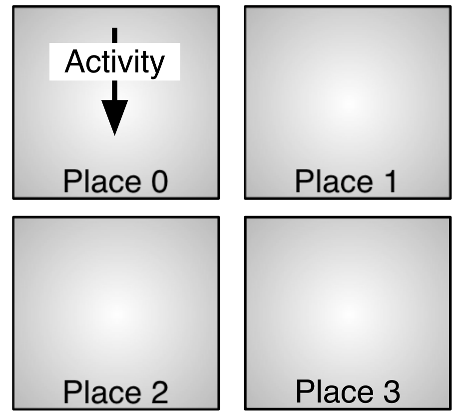
Dave Hudak
Ohio Supercomputer Center
“The X10 Language and Methods for Advanced HPC Programming”

Module Overview

- Parallelism = Activities + Places
- Basic parallel constructs (async, at, finish, atomic)
- Trivial parallel example: Pi approximation
- Shared memory (single place) Prefix Sum

Parallelism in X10

- Activities
  - All X10 programs begin with a single activity executing `main` in place 0
  - Create/control with `at`, `async`, `finish`, `atomic` (and many others!)



- Places hold activities and objects
  - class `x10.lang.Place`
    - Number of places fixed at launch time, available at `Place.MAX_PLACES`
    - `Place.FIRST_PLACE` is place 0
  - Launch an X10 app with mpirun
    - mpirun –np 4 HelloWholeWorld
    - Places numbered 0..3

async

- async S

◆ Creates a new child activity that evaluates expression S asynchronously

  ◆ Evaluation returns immediately

◆ S may reference vals in enclosing blocks

◆ Activities cannot be named

◆ Activity cannot be aborted or cancelled

Stmt* ::= async(p,l) *Stmt

cf Cilk's spawn

```
// Compute the Fibonacci
// sequence in parallel.
def run() {
  if (r < 2) return;
  val f1 = new Fib(r-1),
  val f2 = new Fib(r-2);
  finish {
    async f1.run();
    async f2.run();
  }
  r = f1.r + f2.r;
}
```

Based on "An Overview of X10 2.0", SC09 Tutorial

finish

- L: finish S

◆ Evaluate S, but wait until all (transitively) spawned asyncs have terminated.

◆ implicit finish at main activity

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

Stmt ::= finish Stmt

cf Cilk's sync

```
// Compute the Fibonacci
// sequence in parallel.
def run() {
  if (r < 2) return;
  val f1 = new Fib(r-1),
  val f2 = new Fib(r-2);
  finish {
    async f1.run();
    async f2.run();
  }
  r = f1.r + f2.r;
}
```

Based on "An Overview of X10 2.0", SC09 Tutorial

at

- at(p) S
- Evaluate expression S at place p
- Parent activity is blocked until S completes
- Can be used to
  - Read remote value
  - Write remote value
  - Invoke method on remote object
- As of X10 2.1.0, manipulating objects between places requires a GlobalRef (more on that next module)

Stmt* ::= at(p) *Stmt

```
// Copy field f from a to b
// a and b are GlobalRefs
def copyRemoteFields(a, b) {
  at (b.home) b.f =
    at (a.home) a.f;
}

// Invoke method m on obj
// m is a GlobalRef
def invoke(obj, arg) {
  at (obj.home) obj().m(arg);
}
```

Based on "An Overview of X10 2.0", SC09 Tutorial

atomic

- atomic S
- Evaluate expression S atomically
- Atomic blocks are conceptually executed in a single step while other activities are suspended: isolation and atomicity.
- An atomic block body (S) ...
  - must be nonblocking
  - must not create concurrent activities (sequential)
  - must not access remote data (local)

Based on “An Overview of X10 2.0”, SC09 Tutorial

Stmt ::= atomic _Statement_
MethodModifier ::= atomic

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

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

Single Place Example

- Monte Carlo approximation of π
- Algorithm
  - Consider a circle of radius 1
  - Let N = some large number (say 10000) and count = 0
  - Repeat the following procedure N times
    - Generate two random numbers x and y between 0 and 1 (use the `rand` function)
    - Check whether (x,y) lie inside the circle
    - Increment count if they do
  - Pi $\approx 4 *$ count / N

Pi Approximation

```
public class AsyncPi {
  public static def main(s: Array[String](!)):Void {
    val samplesPerActivity = 10000;
    val numActivities = 8;
    val activityCounts = new Array[Double](1..numActivities, (Point)=>0.0);
    finish for (activityID in 1..numActivities) {
      async {
        val [ActivityIndex] = activityID;
        val r = new Random(activityIndex);
        for (i in 1..samplesPerActivity) {
          val x = r.nextDouble();
          val y = r.nextDouble();
          val z = x*x+y*y;
          if ((x*x + y*y) <= 1.0) {
            activityCounts(activityID)++;
          }
        }
      }
    }
    var globalCount:Double = 0.0;
    for (activityID in 1..numActivities) {
      globalCount += activityCounts(activityID);
    }
    val pi = 4*(globalCount/(samplesPerActivity*numActivities as Double));
    Console.OUT.println("With ”+<snip>+" points, the value of pi is " + pi);
  }
}
```

- Array element per activity to hold count
- Async creates activities, finish for control
- Individual totals added up by main activity

Prefix Sum: Shared Memory Algorithm

- Implemented in X10 using a single place
- Use doubling technique (similar to tree-based reduction). $Log_2(n)$ steps, where
  - Step 1: All i>1, a[i] = a[i] + a[i-1]
  - Step 2: All i>2, a[i] = a[i] + a[i-2]
  - Step 3: All i>4, a[i] = a[i] + a[i-4], and so on…
- AsyncPrefixSum class inherits from PrefixSum
  - Only have to update computeSum method!

1	2	3	4	5	6	7	8
1	3	5	7	9	11	13	15
1	3	6	10	14	18	22	26
1	3	6	10	15	21	28	36

Ohio Supercomputer Center

```
public def computeSum()
{
  val chunkSize = 4;
  val tempArray = new Array[Int](1..prefixSumArray.size(), (Point)=>0);
  val numSteps = <snip> as Int;
  for ([stepNumber] in 1..numSteps) {
    val stepWidth = Math.pow(2, (stepNumber - 1)) as Int;
    val numActivities = Math.ceil(numChunks) as Int;
    Console.OUT.println("numActivities = "+numActivities);
    finish {
      for ([activityId] in 1..numActivities) {
        async {
          for ((j) in low..hi) {
            tempArray(j) = prefixSumArray(j) + prefixSumArray(j-stepWidth);
          } //for j
        } //async
      } //for activityId
    } //finish
```

- Example parallel implementation (not the best, but illustrative…)
- Fixed chunk size
  - At each step, spawn an activity to update each chunk
- tempArray used to avoid race conditions
  - Copied back to prefixSumArray at end of each step

Conclusion

- Activities and places
- async, finish, at, atomic
- Examples of single place programs
  - Pi approximation
  - Prefix Sum

Module 4: X10 Places and DistArrays

Dave Hudak
Ohio Supercomputer Center
"The X10 Language and Methods for Advanced HPC Programming"

Module Overview

- Parallel Hello and Place objects
- Referencing objects in different places
- DistArrays (distributed arrays)
- Distributed memory (multi-place) Prefix Sum

Parallel Hello

```
class HelloWholeWorld {
  public static def main(args:Array[String](1)):void {
    for (var i:Int=0; i<Place.MAX_PLACES; i++) {
      val iVal = i;
      async at (Place.places(iVal)) {
        Console.OUT.println("Hello World from place "+here.id);
      }
    }
  }
}
```

```
Hello World from place 0
Hello World from place 2
Hello World from place 3
Hello World from place 1
```

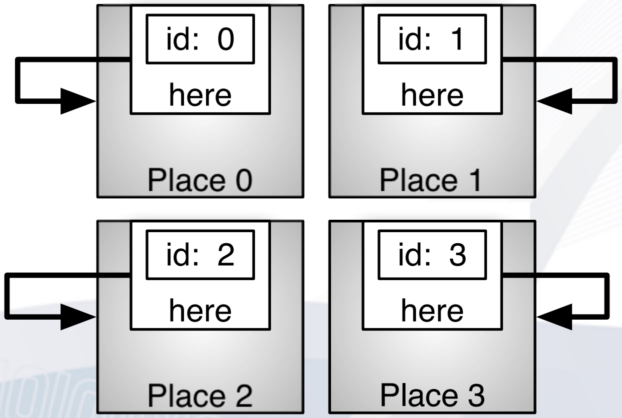
- `at` – place shift
  - Shift current activity to a place to evaluate an expression, then return
  - Copy necessary values from calling place to callee place, discard when done
- `async`
  - start new activity and don't wait for it to complete
- Note that `async at` != `at async`
- `async` and `at` should be thought of as executing via closure
  - We bundle up the values referenced in its code and create an anonymous function (in `at` statement, the bundle is copied to the other place!)
  - Can't reference external var in `async` or `at`, only `val`
  - For example, iVal is a val copy of i for use in at. i is a var and would generate an error

Place Objects

```
class HelloWholeWorld {
  public static def main(args:Array[String](1)):void {
    for (var i:Int=0; i<Place.MAX_PLACES; i++) {
      val iVal = i;
      async at (Place.places(iVal)) {
        Console.OUT.println("Hello World from place "+here.id);
      }
    }
  }
}
```

```
Hello World from place 0
Hello World from place 2
Hello World from place 3
Hello World from place 1
```

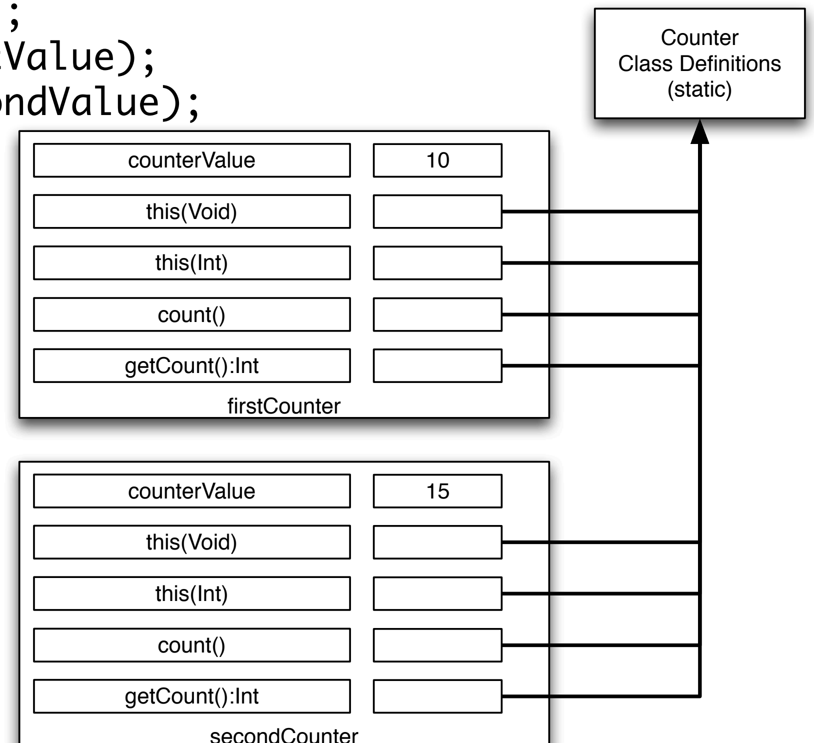
- Place objects have a field called `id` that contains the place number
- `here` – Place object always bound to current place



Objects (Review from Module 2)

```
class Driver {
  public static def main(args:Array[String](1)):Void {
    val firstCounter = new Counter();
    val secondCounter = new Counter(5);
    for (var i:Int=0; i<10; i++) {
      firstCounter.count();
      secondCounter.count();
    }
    val firstValue = firstCounter.getCount();
    val secondValue = secondCounter.getCount();
    Console.OUT.println("First value = "+firstValue);
    Console.OUT.println("Second value = "+secondValue);
  }
}
```

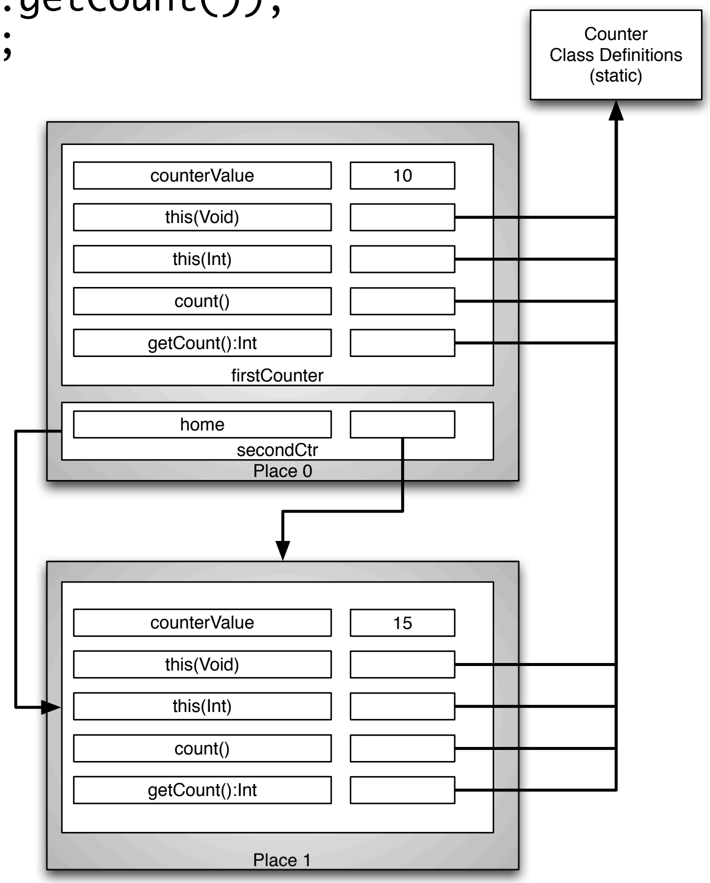
- Object instantiation with new
  - firstCounter uses default constructor, secondCounter uses initialization constructor
  - X10 has garbage collection, so no malloc/free. Object GC'ed when it leaves scope



Objects in Places

```
public static def main(args:Array[String](1)):Void {
  val secondCtr = (at (Place.places(1)) GlobalRef[Counter](new Counter(5)));
  for (var i:Int=0; i<10; i++) {
      at (secondCtr.home) {
        secondCtr().count();
      }
    }
  val secondValue = (at (secondCtr.home) secondCtr().getCount());
  Console.OUT.println("Second value = "+secondValue);
}
```

- Objects instantiated in a place
  - Access objects across places via global references
- secondCtr example
  - Object at Place 1, GlobalRef at Place 0
- GlobalRef object, say g
  - Contains home member: place where original object is instantiated
  - Contains a serialized reference to the original object
  - Supplies reference to original object through g.apply() method, often abbreviated g()
    - g.apply() can only be called when g.home == here

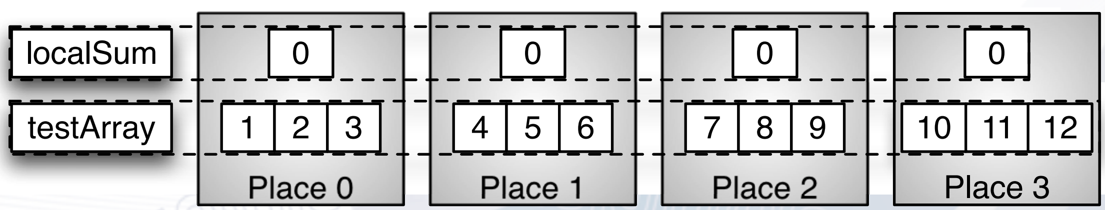


DistArray

```
public static def main(args:Array[String](1)):Void {
   val arraySize = 12;
   val R : Region = 1..arraySize;
   show("Dist.makeUnique() ", Dist.makeUnique());
   show("Dist.makeBlock(R) ", Dist.makeBlock(R));
   show("Dist.makeBlock(R)|here", Dist.makeBlock(R)|here);
   val testArray = DistArray.make[Int](Dist.makeBlock(R), ([i]:Point)=>i);
   val localSum = DistArray.make[Int](Dist.makeUnique(), ((Point)=>0));
```

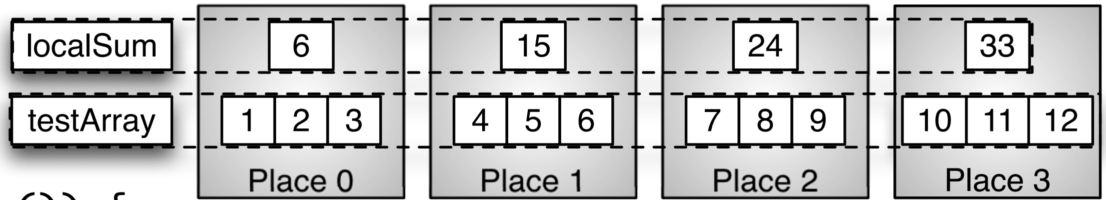
```
dhudak@dhudak-macbook-pro 47%> mpirun -np 4 Driver
Dist.makeUnique()  =  0 1 2 3
Dist.makeBlock(R)  =  0 0 0 1 1 1 2 2 2 3 3 3
Dist.makeBlock(R)|here =  0 0 0
```

- Distributions map regions to places
- Dist factory methods – `makeUnique`, `makeBlock`
  - Cyclic, block-cyclic distributions also supported
- Dist (and range) restrictions using `|` operator
- DistArray similar to Array instantiation
  - Dist object must be provided in addition to base type and initialization function
- DistArray name is visible at all places



DistArray Example

```
finish {
    for (p in testArray.dist.places()) {
      async at (p) {
        for (localPoint in testArray|here) {
          localSum(p.id) += testArray(localPoint);
        }
      }
    }
  }
  var globalSum:Int = 0;
  for (p in localSum.dist.places()) {
    globalSum += (at (p) localSum(p.id));
  }
}
```



- Let's compute the global sum of testArray
- Step 1: sum the subarray at each place
  - Every DistArray object has a member called `dist`
  - Every dist object has a method called `places` that returns an Array of Place objects
  - Create an activity at each place using `async`
- Step 2: `main` activity at place 0
  - retrieves local sum from each place and adds them together

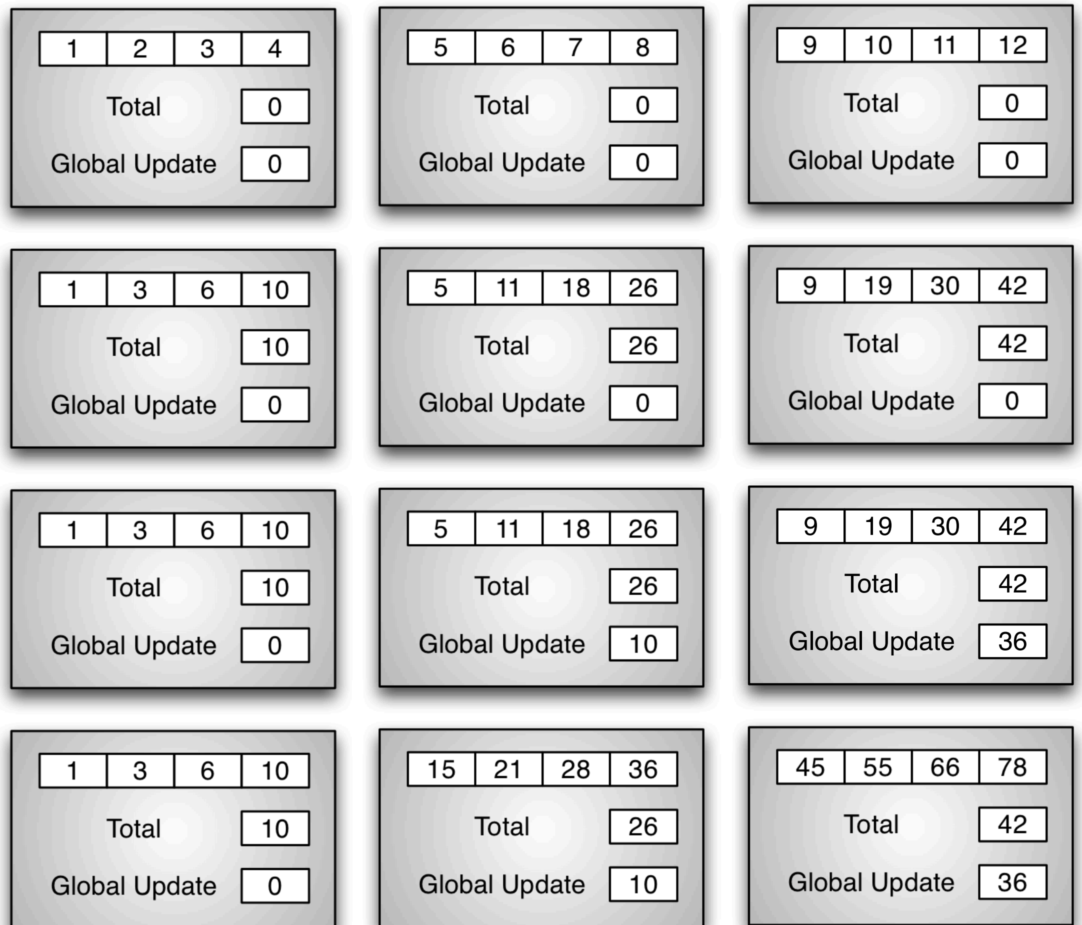
```
val counterArray = DistArray.make[Counter](Dist.makeUnique());
val counterArrayPlaces = counterArray.dist.places();
for (p in counterArrayPlaces) {
  at (p) {
    counterArray(p.id) = new Counter(p.id);
  }
}
for (p in counterArrayPlaces) {
  at (p) {
    val myCounter = counterArray(p.id);
    val myCounterValue = myCounter.getCount();
    Console.OUT.println("Start "+p.id+":  myCounter = "+myCounterValue);
  }
}
```

DistArray of Objects

- Allocate a DistArray of Counters
- Iterate over all places of the DistArray, constructing a Counter object at each place

Prefix Sum: Distributed Memory Algorithm

- Step 1: compute prefix sum and total at each place
- Step 2: each place calculates its global update (sum of preceding totals)
- Step 3: each place updates its elements with its global update



```
public def computeSum()
{
  finish {
    for (p in prefixSumArray.dist.places()) {
      async at (p) {
        localSums(here.id) = 0;
        var first : Boolean = true;
        for ([i] in prefixSumArray|here) {
          localSums(here.id) += prefixSumArray(i);
          if (first) {
            first = false;
          }
          else {
            prefixSumArray(i) = prefixSumArray(i) + prefixSumArray(i-1);
          }
        }  //for i
      } //at
```

Step 1

- Step 1 – compute prefix sum (and total) at each place
- Two distributed arrays in object, `prefixSumArray` and `localSums`

Steps 2 and 3

```
finish {
  for (p in prefixSumArray.dist.places()) {
    async at (p) {
      val placeId = here.id;
      var globalUpdate: Int = 0;
      for (var j:Int=0;j<placeId;j++) {
        val valj = j;
        globalUpdate += (at (Place.places()(valj)) localSums(here.id));
      }
      for ((i) in prefixSumArray.dist|here) {
        prefixSumArray(i) += globalUpdate;
      }  //for i
```

- Step 2 – calculate global offset
  - Place 3 needs to add totals from Place 0, 1 and 2
    - Place.places methods used to obtain place
    - `at` expression retrieves value
    - `valj` needed for closure created `at` expression
- Step 3 – update array with global offset

Conclusion

- Place objects and here for multi-place programming
- Global references
- Distributions map regions to places
- DistArray construction and access
- Distributed Prefix Sum algorithm