Next week’s lectures 9/24 and 9/26

Homework 2: Research and present (10-15 minutes) an array language or library next week

<table>
<thead>
<tr>
<th>Language / Library</th>
<th>Presentation lead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Arrays</td>
<td>Eric H</td>
</tr>
<tr>
<td>NumPy, DistNumPy</td>
<td>Michael</td>
</tr>
<tr>
<td>HTA</td>
<td>Scott</td>
</tr>
<tr>
<td>HPF</td>
<td>Eric L</td>
</tr>
<tr>
<td>BoxLib, Combo,…</td>
<td>Anshu</td>
</tr>
<tr>
<td>PBLAS</td>
<td>Richard</td>
</tr>
<tr>
<td>Data parallel Haskell (Thursday)</td>
<td>Mangpo</td>
</tr>
<tr>
<td>Arrays in Electronic Structures (TCE)</td>
<td>Edgar</td>
</tr>
<tr>
<td>RDD</td>
<td>TD</td>
</tr>
<tr>
<td>X10</td>
<td>Philip</td>
</tr>
<tr>
<td>Sequoia, Legion</td>
<td>Dan?</td>
</tr>
<tr>
<td>Chapel, Cilk, Fortress, NESL, CAF, Titanium, UPC</td>
<td></td>
</tr>
</tbody>
</table>
Developing Scalable Parallel Applications in X10

David Cunningham, David Grove, Vijay Saraswat, and Olivier Tardieu

IBM Research

Based on material from previous X10 Tutorials by Bard Bloom, Evelyn Duesterwald Steve Fink, Nate Nystrom, Igor Peshansky, Christoph von Praun, and Vivek Sarkar
Additional contributions from Anju Kambadur, Josh Milthorpe and Juemin Zhang

This material is based upon work supported in part by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

Please see x10-lang.org for the most up-to-date version of these slides and sample programs.
Programming Model Challenges

- **Scale Out**
  - Program must run across many nodes (distributed memory; network capabilities)

- **Scale Up**
  - Program must exploit multi-core and accelerators (concurrency; heterogeneity)

- **Both Productivity and Performance**
  - Bring modern commercial tooling/languages/practices to HPC programmers
  - Support high-level abstractions, code reuse, rapid prototyping
  - While still enabling full utilization of HPC hardware capabilities at scale
X10 Programming Model

Asynchronous PGAS (Partitioned Global Address Space)

- Global address space, but with explicit locality
- Activities can be dynamically created and joined

Abstracts hardware details, while still allowing programmer to express locality and concurrency.
X10 APGAS Constructs

Fine grained concurrency
• async $S$

Place-shifting operations
• at $(P) S$

Atomicity
• atomic $S$
• when (c) $S$

Ordering
• finish $S$
• clocked, next

Distributed object model
Replicate by default
GlobalRef
Global data structures
DistArray

Two central ideas: Places and Asynchrony
import x10.io.Console;

class HelloWholeWorld {
    public static def main(Array[String]) {
        finish for (p in Place.places()) {
            async at (p)
            Console.OUT.println("Hello World from place" + p.id);
        }
    }
}

(%1) x10c++ -o HelloWholeWorld -O HelloWholeWorld.x10

(%2) runx10 -n 4 HelloWholeWorld
Hello World from place 0
Hello World from place 2
Hello World from place 3
Hello World from place 1

(%3)
import x10.io.Console;
import x10.util.Random;

class MontyPi {
  public static def main(args:Array[String](1)) {
    val N = Int.parse(args(0));
    val r = new Random();
    var result:Double = 0;
    for (1..N) {
      val x = r.nextDouble();
      val y = r.nextDouble();
      if (x*x + y*y <= 1) result++;
    }
    val pi = 4*result/N;
    Console.OUT.println("The value of pi is \" + pi);
  }
}
Concurrent MontyPi

```
import x10.io.Console;
import x10.util.Random;

class MontyPi {
    public static def main(args:Array[String](1)) {
        val N = Int.parse(args(0));
        val P = Int.parse(args(1));
        val result = new Cell[Double](0);
        finish for (1..P) async {
            val r = new Random();
            var myResult:Double = 0;
            for (1..(N/P)) {
                val x = r.nextDouble();
                val y = r.nextDouble();
                if (x*x + y*y <= 1) myResult++;
            }
            atomic result() += myResult;
        }
        val pi = 4*(result())/N;
        Console.OUT.println("The value of pi is " + pi);
    }
}
```
import x10.io.Console;
import x10.util.Random;

class MontyPi {
    public static def main(args:Array[String](1)) {
        val N = Int.parse(args(0));
        val result = GlobalRef[Cell[Double]](new Cell[Double](0));
        finish for (p in Place.places()) at (p) async {
            val r = new Random();
            var myResult:Double = 0;
            for (1..(N/Place.MAX_PLACES)) {
                val x = r.nextDouble();
                val y = r.nextDouble();
                if (x*x + y*y <= 1) myResult++;
            }
            val ans = myResult;
            at (result) atomic result()() += ans;
        }
        val pi = 4*(result()())/N;
        Console.OUT.println("The value of pi is \" + pi);
    }
}
X10 Project

X10 is an open source project (http://x10-lang.org)
  – Released under Eclipse Public License (EPL) v1
  – Current Release: 2.2.1 (September 2011)

X10 Implementations
  – C++ based (“Native X10”)
    • Multi-process (one place per process; multi-node)
    • Linux, AIX, MacOS, Cygwin, BlueGene/P
    • x86, x86_64, PowerPC
  – JVM based (“Managed X10”)
    • Multi-process (one place per JVM process; multi-node)
      – Limitation on Windows to single process (single place)
    • Runs on any Java 5/Java 6 JVM

X10DT (X10 IDE) available for Windows, Linux, Mac OS X
  – Based on Eclipse 3.6
  – Supports many core development tasks including remote build/execute facilities
X10 Compilation Flow

X10 Source → Parsing / Type Check → X10 AST → AST Optimizations AST Lowering → X10 AST

C++ Back-End

C++ Code Generation → C++ Source → C++ Compiler → XRC → Native Code → Native Env

Java Back-End

Java Code Generation → Java Source → Java Compiler → XRX → Bytecode → JNI → Managed X10

Native X10

XRX → Native Code

X10RT → Java VMs
Overview of Language Features

- Many sequential features of Java inherited unchanged
  - Classes (w/ single inheritance)
  - Interfaces, (w/ multiple inheritance)
  - Instance and static fields
  - Constructors, (static) initializers
  - Overloaded, over-rideable methods
  - Garbage collection
  - Familiar control structures, etc.

- Structs

- Closures

- Multi-dimensional & distributed arrays

- Substantial extensions to the type system
  - Dependent types
  - Generic types
  - Function types
  - Type definitions, inference

Concurency
- Fine-grained concurrency:
  - `async S;`
- Atomicity and synchronization
  - `atomic S;`
  - `when (c) S;`
- Ordering
  - `finish S;`

Distribution
- `GlobalRef[T]`
- `at (p) S;`
Classes

- Single inheritance, multiple interfaces
- Static and instance methods
- Static val fields
- Instances fields may be val or var
- Values of class types may be null
- Heap allocated
**Structs**

- User defined primitives
  - No inheritance
  - May implement interfaces
  - All fields are `val`
  - All methods are final
  - Allocated “inline” in containing object, array or variable
  - Headerless

```scala
struct Complex {
  val real: Double;
  val img: Double;

  def this(r: Double, i: Double) {
    real = r; img = i;
  }

  def operator + (that: Complex) {
    return Complex(real + that.real,
                   img + that.img);
  }

  ....
}

val x = new Array[Complex](1..10)
```
Function Types

- \((T_1, T_2, ..., T_n) \Rightarrow U\)
  - type of functions that take arguments \(T_i\) and returns \(U\)

- If \(f: (T) \Rightarrow U\) and \(x: T\)
  then invoke with \(f(x): U\)

- Function types can be used as an interface
  - Define apply operator with the appropriate signature:
    operator this\((x:T): U\)

Closures

First-class functions
\((x: T): U \Rightarrow e\)
used in array initializers:
new Array[int](5, (i:int) => i*i )
yields the array \([ 0, 1, 4, 9, 16 ]\)
Classes, structs, and interfaces may have type parameters

class ArrayList[T]
  - Defines a type constructor
  - and a family of types
    ArrayList[Int], ArrayList[String], ArrayList[C], ...

ArrayList[C]: as if ArrayList class is copied and C substituted for T

Can instantiate on any type, including primitives (e.g., Int)
Classes and structs may have properties
  • public val instance fields
  • class Region(rank:Int, zeroBased:Boolean, rect:Boolean) { ... }

Can constrain properties with a boolean expression
  • Region{rank==3}
    ◆ type of all regions with rank 3
  • Array[Int]{region==R}
    ◆ type of all arrays defined over region R
    ◆ R must be a constant or a final variable in scope at the type

Dependent types are checked statically.

Dependent types used to statically check locality properties

Dependent types used heavily in Array library to check invariants

Dependent type system is extensible

Dependent types enable optimization (compile-time constant propagation of properties)
Type inference

- Field and local variable types inferred from initializer type
  
  ```
  val x = 1;
  - x has type `Int{self==1}`
  val y = 1..2*1..10;
  - y has type `Region{rank==2,rect}`
  ```

- Loop index types inferred from region
  
  ```
  R: Region{rank==2}
  for (p in R) { ... }
  - p has type `Point{rank==2}`
  ```

- Method return types inferred from method body
  
  ```
  def m() { ... return true ... return false ... }
  0 m has return type `Boolean`
  ```

Programming Tip: Often important to not declare types of local vals.
Enables type inference to compute the most precise possible type.
async S

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

$Stmt ::= \text{async } Stmt$

cf Cilk’s spawn

// Compute the Fibonacci sequence in parallel.
def fib(n:Int):Int {
    if (n < 2) return 1;
    val f1:Int;
    val f2:Int;
    finish {
        async f1 = fib(n-1);
        f2 = fib(n-2);
    }
    return f1+f2;
}
// Compute the Fibonacci sequence in parallel.

def fib(n:Int):Int {
    if (n < 2) return 1;
    val f1:Int;
    val f2:Int;
    finish {
        async f1 = fib(n-1);
        f2 = fib(n-2);
    }
    return f1+f2;
}

Stmt ::= finish Stmt

cf Cilk’s sync

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

Rooted exception model

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

▶ 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.
atomic

atomic S

◆ Execute statement $S$ atomically

◆ Atomic blocks are conceptually executed in a serialized order with respect to all other atomic blocks: isolation and weak atomicity.

◆ An atomic block body ($S$) ...
  • must be nonblocking
  • must not create concurrent activities (sequential)
  • must not access remote data (local)
  • restrictions checked dynamically

```java
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;
}
```
when

\[ \text{Stmt ::= WhenStmt} \]
\[ \text{WhenStmt ::= when (Expr) Stmt} \]
\[ \text{ | WhenStmt or (Expr) Stmt} \]

when \((E) \ S\)

- Activity suspends until a state in which the guard \(E\) is true.
- In that state, \(S\) is executed \text{atomically} and in \text{isolation}.
- Guard \(E\) is a boolean expression
  - must be \text{nonblocking}
  - must not create concurrent activities (\text{sequential})
  - must not access remote data (\text{local})
  - must not have side-effects (\text{const})

\begin{verbatim}
class OneBuffer {
  var datum:Object = null;
  var filled:Boolean = false;
  def send(v:Object) {
    when ( !filled ) {
      datum = v;
      filled = true;
    }
  }
  def receive():Object {
    when ( filled ) {
      val v = datum;
      datum = null;
      filled = false;
      return v;
    }
  }
}
\end{verbatim}
Performance Model: async, finish, and when

- Pool of worker threads in each Place
  - Each worker maintains a queue of pending asyncs
  - Worker pops async from its own queue, executes it, repeat...
  - If worker’s queue empty, steal from another worker’s queue

- Finish typically does not block worker (run subtask in queue instead of blocking)

- Finish may block because of thieves or remote async
  - worker 1 reaches end of finish block while worker 2 is still running async

- A blocking finish (or a when) incurs OS-level context switch
  - May incur thread creation cost (maintain set of active workers)

- Cost analysis
  - async
    - queue operations (fences, at worst CAS)
    - heap allocated task object (allocation + GC cost)
  - finish (single place)
    - synchronized counter
    - heap allocated finish object (allocation + GC cost)
Atomicity: atomic and when (and locks)

- X10 currently implements atomic and when trivially with a per-Place lock
  - All atomic/when statements serialized within a Place
  - Scheduler re-evaluates pending when conditions on exit of atomic section
  - Poor scalability on multi-core nodes; when especially inefficient

- Pragmatics: class library provides lower-level alternatives
  - x10.util.concurrent.Lock – pthread mutex
  - x10.util.concurrent.AtomicInteger et al – wrap machine atomic update operations

- An aspect of X10 where our implementation has not yet matched our ambitions…
  - Area for future research
  - Natural fit for transactional memory (STM/HTM/Hybrid)
Performance Model: Native X10 vs. Managed X10

Native X10: base performance model is C++
- Object-oriented features more expensive
  - C++ compilers don’t optimize virtual dispatch, interface invocation, instanceof, etc
  - Memory management: BDWGC
    - Allocation approx like malloc
    - Conservative GC, (non-moving)
    - Parallel, but not concurrent
- Generics map to C++ templates
- Structs map to C++ classes with no virtual methods that are inlined in containing object/array
- C++ compilation is mostly file-based, which limits cross-class optimization
- More deterministic execution (less runtime jitter)

Managed X10: base performance model is Java
- Object-oriented features less expensive
  - JVMs do optimize virtual dispatch, interface invocation, instanceof, etc
  - Memory management
    - Very fast allocation
    - Highly tuned garbage collectors
- Generics map to Java generics
  - Mostly erased
  - Primitives boxed
- Structs map to Java classes (no difference between X10 structs and X10 classes)
- JVM dynamically optimizes entire program
- Less deterministic execution (more jitter) due to JIT compilation, profile-directed optimization, GC
at (p) S

- Execute statement S at place p
- Current activity is blocked until S completes
- Deep copy of local object graph to target place; the variables referenced from S define the roots of the graph to be copied.
- GlobalRef[T] used to create remote pointers across at boundaries

Stmt ::= at (p) Stmt

class C {
  var x:int;
  def this(n:int) { x = n; }
}

// Increment remote counter
def inc(c:GlobalRef[C]) {
  at (c.home) c().x++;
}

// Create GR of C
static def make(init:int) {
  val c = new C(init);
  return GlobalRef[C](c);
}
Distributed Object Model

Objects live in a single place

Objects can only be accessed in the place where they live

Object graph reachable from the statements in an “at” will be deeply copied from the source to destination place and a new isomorphic graph created

Instances fields of an object may be declared transient to break deep copy

GlobalRef[T] enables cross-place pointers to be created

GlobalRef can be freely copied, but referent can only be accessed via the apply at its home place

```java
public struct GlobalRef[T](home: Place){T<:Object} {
    public native def this(t:T): GlobalRef{self.home==here};

    public native operator this(){here == this.home}:T;
}
```
Hello Whole World

```java
1/ class HelloWholeWorld {
2/   public static def main(args:Rail[String]) {
3/     finish
4/       for (p in Place.places())
5/         at (p)
6/           async
7/             Console.OUT.println(p + " says " + args(0));
8/           Console.OUT.println("Goodbye");
9/     }
10/}
```

% x10c++ HelloWholeWorld.x10
% X10_NPLACES=4; ./a.out hello
Place 0 says hello
Place 2 says hello
Place 3 says hello
Place 1 says hello
Goodbye
APGAS Idioms

- Remote evaluation
  \[ v = \texttt{at} (p) \texttt{evalThere}(arg1, arg2); \]

- Active message
  \[ \texttt{at} (p) \texttt{async} \texttt{runThere} (arg1, arg2); \]

- Recursive parallel decomposition
  \[
  \textbf{def} \; \texttt{fib}(n: \texttt{Int}): \texttt{Int} \; \{ \\
  \hspace{1em} \textbf{if} \; (n < 2) \; \texttt{return} \; 1; \\
  \hspace{1em} \texttt{val} \; f1: \texttt{Int}; \\
  \hspace{1em} \texttt{val} \; f2: \texttt{Int}; \\
  \hspace{1em} \texttt{finish} \; \{ \\
  \hspace{2em} \texttt{async} \; f1 = \texttt{fib}(n-1); \\
  \hspace{2em} f2 = \texttt{fib}(n-2); \\
  \hspace{1em} \} \\
  \hspace{1em} \texttt{return} \; f1 + f2; \\
  \}
  \]

- SPMD
  \[
  \texttt{finish for} \; (p \; \text{in} \; \texttt{Place.places}()) \; \{ \\
  \hspace{1em} \texttt{at}(p) \; \texttt{async} \; \texttt{runEverywhere}(); \\
  \}
  \]

- Atomic remote update
  \[ \texttt{at} (\texttt{ref}) \; \texttt{async} \; \texttt{atomic} \; \texttt{ref}() \; += \; v; \]

- Data exchange
  \[
  \text{\hspace{1em}} \text{// swap row } i \text{ local and } j \text{ remote} \\
  \hspace{1em} \texttt{val} \; h = \texttt{here}; \\
  \hspace{1em} \texttt{val} \; \texttt{row}_i = \texttt{rows}()(i); \\
  \hspace{1em} \texttt{finish} \; \texttt{at}(p) \; \texttt{async} \; \{ \\
  \hspace{2em} \texttt{val} \; \texttt{row}_j = \texttt{rows}()(j); \\
  \hspace{2em} \texttt{rows}()(j) = \texttt{row}_i; \\
  \hspace{2em} \texttt{at}(h) \; \texttt{async} \; \texttt{row}()(i) = \texttt{row}_j; \\
  \}\n  \]

A handful of key constructs cover a broad spectrum of patterns
Array and DistArray

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

- A **Region** is an ordered set of points, all of the same rank

- An **Array** maps every **Point** in its defining **Region** to a corresponding data element.

- Key operations
  - creation
  - apply/set (read/write elements)
  - map, reduce, scan
  - iteration, enhanced for loops
  - copy

- A **Dist** (distribution) maps every **Point** in its **Region** to a **Place**

- A **DistArray** (distributed array) is defined over a **Dist** and maps every **Point** in its distribution to a corresponding data element.

  Data elements in the **DistArray** may only be accessed in their home location (the place to which the **Dist** maps their **Point**)

- Key operations similar to Array, but bulk operations (map, reduce, etc) are distributed operations

With the exception of Array literals (\([1, 2, 3] === \text{new Array[int]}(3,(i:\text{int})=>i+1)\) ),

Array and DistArray are implemented completely as library classes in the x10.array package with no special compiler support.
DistArrays in Detail

We’ll examine the code of DistArray, Dist, BlockDist
- LocalStorage accessed by PlaceLocalHandle
- Use of transient fields to control serialization
- Indexing operations
  - Key idea: Dist.offset is an abstract method that subclasses must implement that is responsible for mapping Points in the global index space into offsets in local storage
- Bulk operations (eg reduce) implemented internally using distributed collecting finish
LU Factorization

- HPL-like algorithm
- 2-d block-cyclic data distribution
- right-looking variant with row partial pivoting (*no look ahead*)
- recursive panel factorization with pivot search and column broadcast combined
- SPMD implementation
- one top-level async per place finish at each (p in Dist.makeUnique())
- globally synchronous progress
- collectives: barriers, broadcast, reductions (row, column, WORLD)
- point-to-point RDMAs for row swaps
- uses BLAS for local computations
Global Matrix Library

GML
- Dense
- SparseCSC
- Vector
- Block matrix
  - Distr. block
  - Dupl. block
- Dense matrix
  - BLAS wrap
- Sparse matrix
  - X10 driver
- Dense matrix
  - X10 driver

X10
- Native C/C++ back end
  - Team
    - MPI
    - Socket
    - PAMI
    - PGAS
- Managed Java back end
  - Socket

3rd party C-MPI library
BLAS
Multi-thread BLAS (GotoBLAS)
LAPACK

© 2012 IBM Corporation
Matrix Partitioning and Distribution

Grid partitioning

Balanced partitioning

User-specified partitioning

Block matrix

Dense, sparse, or mixed blocks

Block distribution: block - place mapping

*Unique*, constant, cyclic, grid, vertical, horizontal and random
Matrix Partitioning and Distribution

Grid partitioning

Balanced partitioning

User-specified partitioning

Block matrix

Dense, sparse, or mixed blocks

Block distribution: block - place mapping

Unique, constant, cyclic, grid, vertical, horizontal and random
Matrix Partitioning and Distribution

Grid partitioning

Balanced partitioning
User-specified partitioning

Block matrix

Dense, sparse, or mixed blocks

Block distribution: block - place mapping

Unique, constant, cyclic, grid, vertical, horizontal and random
Row Swap from X10 LU Benchmark

// swap row with index srcRow located here with row with index dstRow located at place dst

```scala
// swap row with index srcRow located here with row with index dstRow located at place dst
def rowSwap(matrix: PlaceLocalHandle[Matrix[Double]], srcRow: Int, dstRow: Int, dst: Place) {
    val srcBuffer = buffers();
    val srcBufferRef = new RemoteRail(srcBuffer);
    val size = matrix().getRow(srcRow, srcBuffer);
    finish {
        at (dst) async {
            finish {
                val dstBuffer = buffers();
                Array.asyncCopy[Double](srcBufferRef, 0, dstBuffer, 0, size);
            }
            matrix().swapRow(dstRow, dstBuffer);
            Array.asyncCopy[Double](dstBuffer, 0, srcBufferRef, 0, size);
        }
    }
    matrix().setRow(srcRow, srcBuffer);
}
```
Extending DistArray: MortonDist

MortonDist (3D) used in FMM to define a DistArray that distributes octree boxes (key data structure) to X10 places

See au.edu.anu.mm.MortonDist.x10
Extending DistArray: PeriodicDist

Enables modular ("wrap-around") indexing at each Place

Any Dist can be made periodic by encapsulating it in a PeriodicDist. Can then be used in DistArray to get modular indexing

See x10.array.PeriodicDist
Arrays in Detail

We’ll examine the implementation of x10.array.Array

- Array indexing
  - Specialized implementations based on properties.
  - Optimized for common case of Regions of rank<5, zero-based and rectangular (no holes in index space)
  - General case fully supported with minimal time overhead on indexing (space usage based on bounding box of Region)
  - Constructor does computations to optimize later indexing
  - If bounds-checks are disabled, Native X10 indexing performance identical to equivalent C array

- Array iteration
  - Compiler optimizes iteration over rectangular Regions into counted for loops.
  - Non-rectangular regions fully supported, but uses Point-based Region iterator API (significant performance overhead)
Global Load Balancing

- Unbalanced Tree Search Benchmark
  - count nodes in randomly generated tree
  - separable cryptographic random number generator
  - highly unbalanced trees
  - unpredictable
  - tree traversal can be easily relocated (no data dependencies, no locality)

- Problems to be solved:
  - Dynamically balance work across a large number of places efficiently
  - Detect termination of the computation quickly.

- Conventional Approach:
  - Worker steals from randomly chosen victim when local work is exhausted.
  - Key problem: When does a worker know there is no more work?
Global Load Balance: The Lifeline Solution

- Intuition: X0 already has a framework for distributed termination detection – finish. Use it!

- But this requires activities terminate at some point!

- Idea: Let activities terminate after w rounds of failed steals. But ensure that a worker with some work can “distribute” work (= use `at(p) async S`) to nodes known to be idle.
  – Lifeline graphs. Before a worker dies it creates z lifelines to other nodes.
  – These form a distribution “overlay” graph.
    Our paper/implementation uses hyper-cube.
Scalable Global Load Balancing

Unbalanced Tree Search

- Lifeline-based global work stealing [PPoPP’11]
  - $n$ random victims then $p$ lifelines (hypercube)
    - fixed graph with low degree and low diameter
  - synchronous (steal) then asynchronous (deal)

- Root finish accounts for
  - startup asyncs + lifeline asyncs
  - not random steal attempts

- Compact work queue (for shallow trees)
  - represent intervals of siblings
  - thief steals half of each work item

- Sparse communication graph
  - bounded list of potential random victims
  - finish trades contention for latency

**genuine APGAS algorithm**
X10 At Scale

X10 has places and asyncs in each place

We want to
- Handle millions of asyncs (\(\Rightarrow\) billions)
- Handle tens of thousands of places (\(\Rightarrow\) millions)

We need to
- Scale up
  - shared memory parallelism (today: 32 cores per place)
  - schedule many asyncs with a few hardware threads
- Scale out
  - distributed memory parallelism (today: 50K places)
  - provide mechanisms for efficient distribution (data & control)
  - support distributed load balancing
DARPA PERCS Prototype (Power 775)

- **Compute Node**
  - 32 Power7 cores 3.84 GHz
  - 128 GB DRAM
  - peak performance: 982 Gflops
  - *Torrent* interconnect

- **Drawer**
  - 8 nodes

- **Rack**
  - 8 to 12 drawers

- **Full Prototype**
  - up to 1,740 compute nodes
  - up to 55,680 cores
  - up to 1.7 petaflops
    - 1 petaflops with 1,024 compute nodes
Eight Benchmarks

- HPC Challenge benchmarks
  - Linpack: TOP500 (flops)
  - Stream Triad: local memory bandwidth
  - Random Access: distributed memory bandwidth
  - Fast Fourier Transform: mix

- Machine learning kernels
  - KMEANS: graph clustering
  - SSCA1: pattern matching
  - SSCA2: irregular graph traversal
  - UTS: unbalanced tree traversal

Implemented in X10 as pure scale out tests

- One core = one place = one main async

- Native libraries for sequential math kernels: ESSL, FFTW, SHA1
### Performance at Scale (Weak Scaling)

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Cores</th>
<th>Absolute Performance at Scale</th>
<th>Parallel Efficiency (Weak Scaling)</th>
<th>Performance Relative to Best Implementation Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream</td>
<td>55,680</td>
<td>397 TB/s</td>
<td>98%</td>
<td>85% (lack of prefetching)</td>
</tr>
<tr>
<td>FFT</td>
<td>32,768</td>
<td>27 Tflops</td>
<td>93%</td>
<td>40% (no tuning of seq. code)</td>
</tr>
<tr>
<td>Linpack</td>
<td>32,768</td>
<td>589 Tflops</td>
<td>80%</td>
<td>80% (mix of limitations)</td>
</tr>
<tr>
<td>RandomAccess</td>
<td>32,768</td>
<td>843 Gups</td>
<td>100%</td>
<td>76% (network stack overhead)</td>
</tr>
<tr>
<td>KMeans</td>
<td>47,040</td>
<td>depends on parameters</td>
<td>97.8%</td>
<td>66% (vectorization issue)</td>
</tr>
<tr>
<td>SSCA1</td>
<td>47,040</td>
<td>depends on parameters</td>
<td>98.5%</td>
<td>100%</td>
</tr>
<tr>
<td>SSCA2</td>
<td>47,040</td>
<td>245 B edges/s</td>
<td>&gt; 75%</td>
<td>no comparison data</td>
</tr>
<tr>
<td>UTS (geometric)</td>
<td>55,680</td>
<td>596 B nodes/s</td>
<td>98%</td>
<td>reference code does not scale 4x to 16x faster than UPC code</td>
</tr>
</tbody>
</table>
IBM Mega Traffic Simulator (Megaffic)

IBM Research has created a Large-scale multi-agent traffic simulator.

Analysis by Mathematical Modeling
- Link Cost Prediction
- Driver Behavior Modeling
- Traffic Demands Estimation

Road Transition cost
Driver Behavior Model
Traffic Demand Data
Map Data

Traffic Simulator
Megaffic

Simulation Base
IBM eXtensible Agent eXecution InfraStructure (XAXIS)

Visualization
e.g.
- CO2 emission
- moving time
- speed
- vehicle num

Scalable simulation engine for supporting parallel execution environment

Graph Editor

Driver Behavior Modeling:
Automatically generate preferences of drivers from probe car data. A preference is a set of weights over different policies such as minimum travel time, minimum # of turns, etc. The weights are automatically learned from the data.

This project is funded by the PREDICT project of Ministry of Internal Affairs and Communications, Japan.
M3R – A Hadoop re-implementation in Main Memory, using X10

- **Hadoop**
  - Popular Java API for Map/Reduce programming
  - Out of core, resilient, scalable (1000 nodes)
  - Based on HDFS (a resilient distributed filesystem)

- **M3R/Hadoop**
  - Reimplementation of Hadoop API using Managed X10 (X10 compiled to Java)
    - X10 provides scalable multi-JVM runtime with efficient communication
  - Existing Hadoop 1.0 applications ‘just work’
  - Reuse HDFS (and some other parts of Hadoop)
  - In-memory: problem size must fit in aggregate cluster RAM
  - Not resilient: cluster scales until MTBF barrier
  - But considerably faster (closer to HPC speeds)
  - Trade resilience for performance (both latency and throughput)
The core M3R engine provides X10 and Java Map/Reduce APIs against which application programs can be written.

On top of the engine, a Hadoop API compatibility layer allows unmodified Hadoop Map/Reduce jobs to be executed on the engine.

The compatibility layer is written using a mix of X10 and Java code and heavily uses the Java interoperability capabilities of Managed X10.
Intuition: Where does M3R gain performance relative to Hadoop?

- Reducing Disk I/O
- Reducing network communication
- Reducing serialization/deserialization
  - Reduce translation of object graphs to byte buffers & back
- Reducing “other” costs
  - Job submission speed
  - Startup time (JVM reuse synergistic with JIT compilation)
  - ...
- Partition Stability (iterative jobs)
  - Enable application programmer to “pin” data in memory within Hadoop APIs
  - The reducer associated with a given partition number will always be run in the same JVM

For details, see [Shinnar et al VLDB’12]
Summary

Whirlwind tour through X10 language
Additional details will be introduced via case studies
See http://x10-lang.org for specification, samples, etc.