APGAS Programming in X10

http://x10-lang.org

This tutorial was originally given by Olivier Tardieu as part of the Hartree Centre Summer School 2013 "Programming for Petascale".

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

IBM

Foreword

X10 is

- A language
  - Scala-like syntax
  - object-oriented, imperative, strongly typed, garbage collected
  - focus on scale ➔ focus on parallelism and distribution
  - focus on productivity
- An implementation of the APGAS programming model
  - Asynchronous Partitioned Global Address Space
    - PGAS: single address space but with internal structure (➔ locality control)
    - asynchronous: task-based parallelism, active-message-based distribution
- A tool chain
  - compiler, runtime, standard library, IDE
  - open-source *research* prototype

Objectives of this tutorial: learn about X10, (A)PGAS, and the X10 tool chain

Links

- Main X10 website
  http://x10-lang.org

- X10 Language Specification
  http://x10.sourceforge.net/documentation/languagespec/x10-240.pdf

- A Brief Introduction to X10 (for the HPC Programmer)
  http://x10.sourceforge.net/documentation/intro/2.4.0/html/

- X10 2.4.0 (command line tools only)
  https://sourceforge.net/projects/x10/files/x10/2.4.0/

- X10DT 2.4.0 (Eclipse-based IDE)
  https://sourceforge.net/projects/x10/files/x10dt/2.4.0/

Tutorial Outline

Part 1

- Overview
- Sequential language

Part 2

- Task parallelism

Part 3

- Distribution

Part 4

- Programming for Scale

This tutorial is about X10 2.4 (released Sept 2013; major revision of arrays)

Part 1

X10 and APGAS Overview

X10: Productivity and Performance at Scale

>9 years of R&D by IBM Research supported by DARPA (HPCS/PERCS)

- Bring Java-like productivity to HPC
  - evolution of Java with input from Scala, ZPL, CCP, …
  - imperative OO language, garbage collected, type and memory safe
  - rich data types and type system
  - few simple constructs for parallelism, concurrency control, and distribution
  - tools

- Design for scale
  - scale out
    - run across many compute nodes
  - scale up
    - exploit multi-cores and accelerators
  - enable full utilization of HPC hardware capabilities

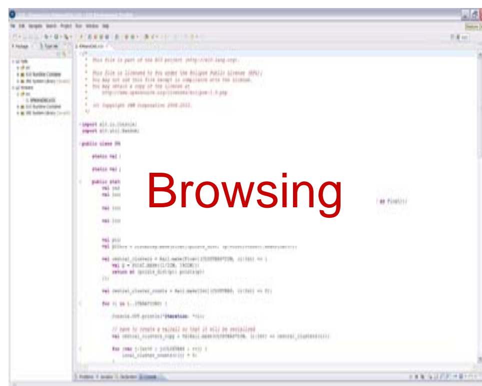
X10 Tool Chain

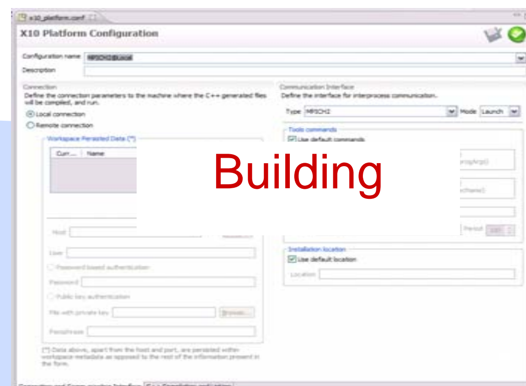
Open-source compiler, runtime, standard library, IDE

- Dual path
  - compiles X10 to C++ or Java
- Command-line compiler and launcher
  - OS: Linux, Mac OSX, Windows, *AIX*
  - CPU: Power and x86
  - transport: shared memory, TCP/IP sockets, MPI, PAMI, *DCMF*
  - backend C++ compiler: g++ and xlC
  - backend JVM: IBM and Oracle JVMs, Java v6 and v7
- Eclipse-based IDE
  - edit, browse, compile and launch, remote compile and launch

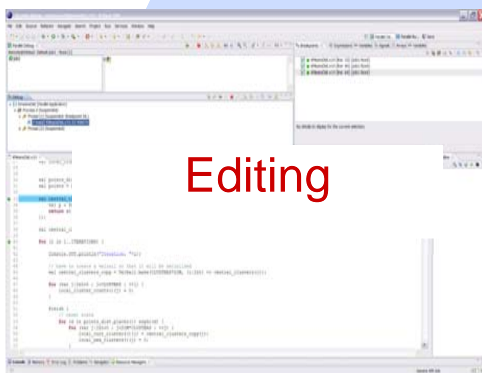
X10DT

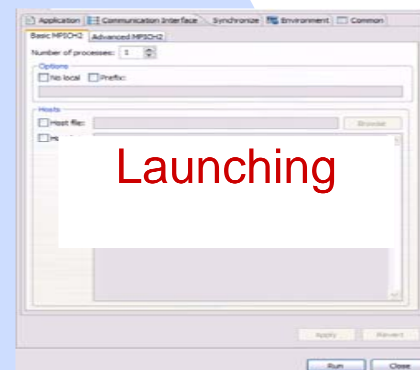
Source navigation, syntax highlighting, parsing errors, folding, hyperlinking, outline and quick outline, hover help, content assist, type hierarchy, format, search, call graph, quick fixes





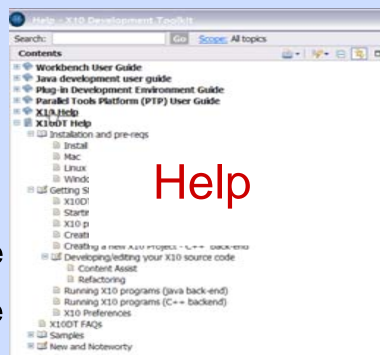
- Java/C++ support
- Local and remote





X10 programmer guide

X10DT usage guide



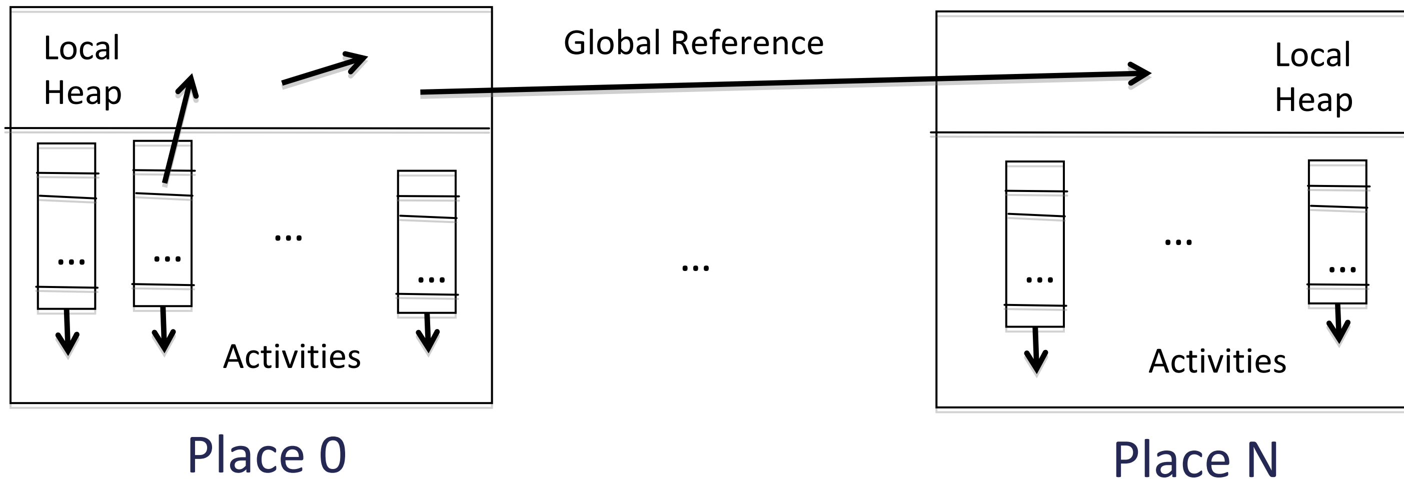
Partitioned Global Address Space (PGAS)

- Message passing
  - each task lives in its own address space
  - example: MPI

- Shared memory
  - shared address space for all the tasks
  - example: OpenMP

- PGAS
  - global address space: single address space across all tasks
    - in X10 any task can refer to any object (local or remote)
  - partitioned address space: *clear* distinction between local and remote memory
    - each partition must fit within a shared-memory node
    - in X10 a task can only operate on local objects
  - examples: Titanium, UPC, Co-array Fortran, X10, Chapel

APGAS in X10: Places and Tasks



Task parallelism

- **async** S
- **finish** S

Concurrency control within a place

- **when**(c) S
- **atomic** S

Place-shifting operations

- **at**(p) S
- **at**(p) e

Distributed heap

- **GlobalRef**[T]
- **PlaceLocalHandle**[T]

APGAS Idioms

- Remote evaluation

```
v = at(p) evalThere(arg1, arg2);
```

- Active message

```
at(p) async runThere(arg1, arg2);
```

- Recursive parallel decomposition

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

- SPMD

```
finish for(p in Place.places()) {
  at(p) async runEverywhere();
}
```

- Atomic remote update

```
at(ref) async atomic ref() += v;
```

- Data exchange

```
// swap l() local and r() remote
val _l = l();
finish at(r) async {
  val _r = r();
  r() = _l;
  at(l) async l() = _r;
}
```

IBM

Sequential Language

Java-like Features

- Objects
  - classes and interfaces
    - single-class inheritance, multiple interfaces
  - fields, methods, constructors
  - virtual dispatch, overriding, overloading, static methods
- Packages and files
- Garbage collected
- Variables and values (final variables, but final is the default)
  - definite assignment
- Expressions and statements
  - control statements: if, switch, for, while, do-while, break, continue, return
- Exceptions
  - try-catch-finally, throw
- Comprehension loops and iterators

Beyond Java: Syntax and Types

- Syntax

types	“x:Int” rather than “Int x”
declarations	val, var, def
function literals	(a:Int, b:Int) => a < b ? a : b
ranges	0..(size-1)
operators	user-defined behavior for standard operators

- Types

local type inference	val b = false;
function types	(Int, Int) => Int
typedefs	type BinOp[T] = (T, T) => T;
structs	headerless inline objects
arrays	multi-dimensional, distributed
properties and constraints	extended static checking
reified generics	~ templates

to be continued…

Hello.x10

```
package examples;
import x10.io.Console;

public class Hello {                                    // class
  protected val n:Long;                                 // field

  public def this(n:Long) { this.n = n; }               // constructor

  public def test() = n > 0;                            // method

  public static def main(args:Rail[String]) {           // main method
    Console.OUT.println("Hello world! ");
    val foo = new Hello(args.size);                     // inferred type
    var result:Boolean = foo.test();                    // no inference for vars
    if(result) Console.OUT.println("The first arg is: " + args(0));
  }
}
```

Compiling and Running X10 Programs

- C++ backend

  x10c++ -O Hello.x10 -o hello; ./hello

- Java backend

  x10c -O Hello.x10; x10 examples.Hello

- Compiler flags

-O	generate optimized code
-NO_CHECKS	disable generation of all runtime checks (null, bounds…)
-x10rt <impl>	select x10rt implementation: sockets (default), pami, mpi…
	(cf. runtime flag for Java backend)

- Runtime configuration: environment variables

X10_NPLACES=<n>	number of places (x10rt sockets on localhost)
X10_NTHREADS=<n>	number of worker threads per place

IBM

Primitive Types and Structs

- Structs cannot extend other data types or be extended or have mutable fields
  - Structs are allocated inline and have no header
- Primitive types are structs with native implementations
  - Boolean, Char, Byte, Short, Int, Long, Float, Double, UByte, UShort, UInt, ULong

```
public struct Complex implements Arithmetic[Complex] {
  public val re:Double;
  public val im:Double;

  public @Inline def this(re:Double, im:Double) { this.re = re; this.im = im;  }

  public operator this + (that:Complex) = Complex(re + that.re, im + that.im);

  // and more
}

// a:Rail[Complex](N) has same layout as b:Rail[Double](2*N) in memory
// a(i).re ~ b(2*i) and a(i).im ~ b(2*i+1)
```

Arrays in X10 2.4

Primitive arrays

- x10.lang.Rail[T]
  - fixed-size, zero-based, dense, 1d array with elements of type T
  - long indices, bounds checking
  - *generic X10 class with native implementations*

x10.array package

- x10.array.Array[T]
  - fixed-size, zero-based, dense, multi-dimensional, rectangular array of type T
  - abstract class, implementations provided for row-major 1d, 2d, 3d arrays
  - *pure X10 code built on top of Rail[T] (easy to copy and tweak)*
- x10.array.DistArray[T]
  - fixed-size, zero-based, dense, multi-dimensional, distributed rectangular array
  - abstract class with a small set of possible implementations (growing)
  - *pure X10 code built on top of Rail[T] and PlaceLocalHandle[T]*

ArraySum.x10

```
package examples;
import x10.array.*;

public class ArraySum {
  static N = 10;

  static def reduce[T](a:Array[T], f:(T,T)=>T){T haszero} {
    var result:T = Zero.get[T]();
    for(v in a) result = f(result, v);
    return result;
  }

  public static def main(Rail[String]) {
    val a = new Array_2[Double](N, N);
    for(var i:Long=0; i<N; ++i) for(j in 0..(N-1)) a(i,j) = i+j;
    Console.OUT.println("Sum: " + reduce(a, (x:Double,y:Double)=>x+y));
  }
}
```

Properties and Constraints

- Classes and structs may specify property fields (~ public final fields)
- Constraints are Boolean expressions over properties and constants
  - equality and inequality constraints, T haszero, subtyping constraint, isref constraint
- Constraints appear on
  - types: restrict the possible values of the type
  - methods: guard on method receiver and parameters
  - and classes: invariant valid for all instances of the class
- Constraints are checked at compile time. Failed checks can
  - be ignored — use -NO_CHECKS flag
  - abort compilation — use -STATIC_CHECKS flag
  - be deferred to runtime if possible — default, use -VERBOSE_CHECKS for details

Vector.x10

```
package examples;

public class Vector[T](size:Long){T haszero,T<:Arithmetic[T]} {
  val raw:Rail[T]{self!=null,self.size==this.size};
  // this refers to the object instance, self to the value being constrained

  def this(size:Long) { property(size); raw = new Rail[T](size); }

  def add(vec:Vector[T]){vec.size==this.size}:Vector[T]{self.size==this.size} {
    for(i in 0..(size-1)) raw(i) += vec.raw(i);
    return this;
  }

  public static def main(Rail[String]) {
    val v = new Vector[Int](4);
    val w = new Vector[Int](5);
    v.add(w);
  }
}
// fails compiling with -STATIC_CHECKS or throws x10.lang.FailedDynamicCheckException
```

Gotchas

- No mutable static fields

- Default integral type is Long
  - To represent Int literals, use n (or N) (e.g. 0n instead of 0)

- Standard coercions except for == and !=
  - 0n<=10 is ok, 0n==10 is not

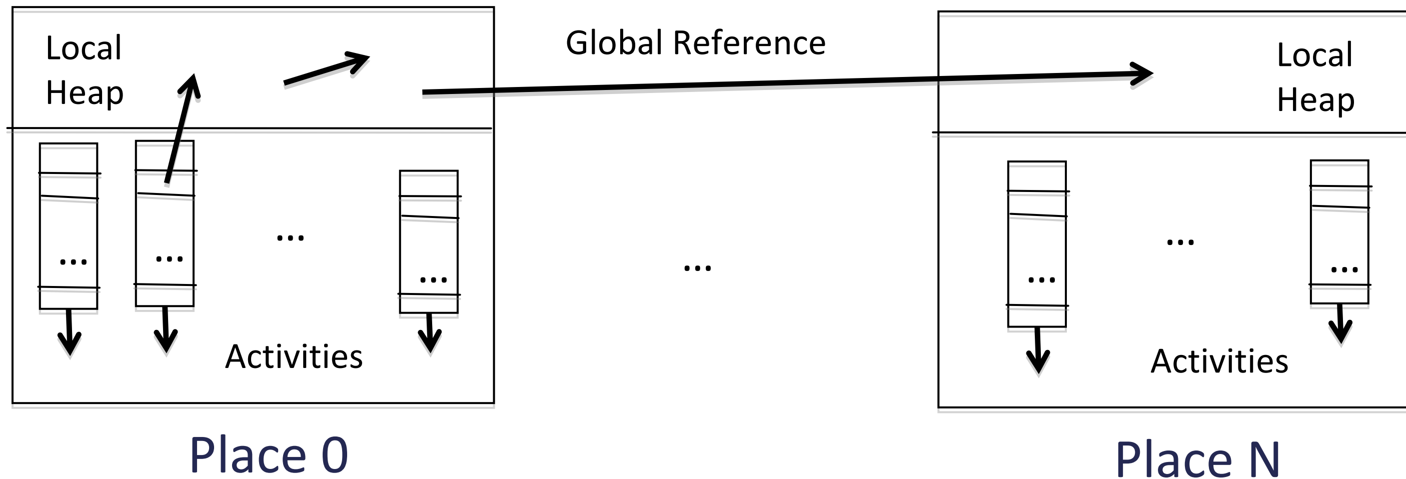
- No root Object class but a root Any interface
  - every type implicitly implements the Any interface (including Int, String…)
  - Any declares toString(), typeName(), equals(Any), hashCode()
  - default implementations are provided for all types

- Type inference
  - too much: def f()=7 has inferred return type Long{self==7L}
  - too little:

```
val a:Array[Long] = new Array_2[Long](3,4);
a(0,0) = 0;         // does not type check
```

Part 2

APGAS in X10: Places and Tasks



Task parallelism

- `async S`
- `finish S`

Place-shifting operations

- `at(p) S`
- `at(p) e`

Concurrency control within a place

- `when(c) S`
- `atomic S`

Distributed heap

- `GlobalRef[T]`
- `PlaceLocalHandle[T]`

Task Parallelism

Task Parallelism: async and finish

- async S
  - creates a new task that executes S
  - returns immediately
  - S may reference values in scope
    - S may initialize values declared above the enclosing finish
  - S may reference variables declared above the enclosing finish
  - tasks cannot be named or cancelled

- finish S
  - executes S
  - then waits until all transitively spawned tasks in S have terminated
  - rooted exception model
    - trap all exceptions and throw a multi-exception if any spawned task terminates abnormally
    - exception is thrown after all tasks have completed
  - collecting finish combines finish with reduction over values offered by subtasks

IBM

Happen-Before and May-Happen in Parallel

```
def m1() {
  async S1;
  async { finish { async S2; } }
  S3;
  async S4;
}

def m2() {
  async S0;
  finish {        // F1
   finish {       // F2
     m1();
     async S5;
   }
   async S6;
  }
}
```

- F2 waits for S1, S2, S3, S4, S5
- F1 waits for F2, S6
- F1 waits for S1 to S6

- S4 starts after S3 completes
- S5 starts after S3 completes

- S6 starts after F2 finishes

- S0 and S3 may run in parallel
- S0 and S6 may run in parallel
- S3 and S6 may not run in parallel
- S5 and S6 may not run in parallel

Variable Scopes: AsyncScope.x10 and Fib.x10

```
val val1 = 1;
var var1:Long = 2;
finish {
  val val2 = 3;
  var var2:Long = 4;
  async {
    val tmp1 = val1;  // ok
    val tmp2 = val2;  // ok
    val tmp3 = var1;  // ok
    val tmp4 = var2;  // illegal (*)
  }
  async {
    var1 = 5;         // ok
    var2 = 6;         // illegal (*)
  }
}
// var1 has a race
```

```
// asynchronous initialization

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

// f1 is declared before finish
// f1 is accessed by one task only
// f1 is read after finish
// f1 is guaranteed race free
```

(*) Local variable cannot be captured in an async if there is no enclosing finish in the same scoping-level.

Concurrency Control: atomic and when

- atomic S
  - executes statement S atomically
    - atomic blocks are conceptually executed in a serialized order with respect to all other atomic blocks in a place (weak atomicity)
  - S must be non-blocking, sequential, and local
    - no when, at, async…
- when(c) S
  - the current task suspends until a state is reached where c is true
  - in that state, S is executed atomically
  - Boolean expression c must be non-blocking, sequential, local, and pure
    - no when, at, async, no side effects
- Gotcha: S in when(c) S is not guaranteed to execute
  - if c is not set to true within an atomic block
  - or if c oscillates

Examples

```
class Account {
  public var value:Int;

  def transfer(src:Account, v:Int) {
    atomic {
       src.value -= v;
       this.value += v;
    }
  }
}
```

```
class Latch {
  private var b:Boolean = false;
  def release() { atomic b = true; }
  def await() { when(b); }
}
```

```
class Buffer[T]{T isref,T haszero} {
  protected var datum:T = null;

  public def send(v:T){v!=null} {
    when(datum == null) {
      datum = v;
    }
  }

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

Implementation Status

- X10 currently implements atomic and when trivially with a per-place lock
  - all atomic and when statements are serialized within a place
  - scheduler re-evaluates pending when conditions on exit of all atomic sections
  - poor scalability on multi-core nodes; when especially inefficient
- For pragmatic reasons the class library provides lower-level alternatives
  - x10.util.concurrent.Lock – pthread mutex
  - x10.util.concurrent.AtomicInteger et al. – wrap machine atomic update operations
  - x10.util.concurrent.Latch
  - …
- Our implementation has not yet matched our ambitions
  - area for future research
  - natural fit for transactional memory (STM/HTM/Hybrid)

Clocks

APGAS barriers

- synchronize dynamic sets of tasks

x10.lang.Clock

- anonymous or named
- task instantiating the clock is registered with the clock
- spawned tasks can be registered with a clock at creation time
- tasks can deregister from the clock
- tasks can use multiple clocks
- split-phase clocks
  - clock.resume(), clock.advance()
- compatible with distribution

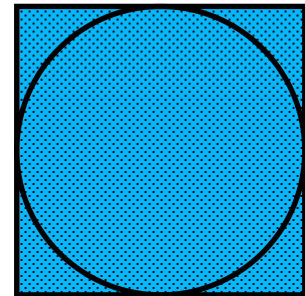
```
// anonymous clock
clocked finish {
  for(1..4) clocked async {
    Console.OUT.println("Phase 1");
    Clock.advanceAll();
    Console.OUT.println("Phase 2");
  }
}
// named clock
finish {
  val c = Clock.make();
  for(1..4) async clocked(c) {
    Console.OUT.println("Phase 3");
    c.advance();
    Console.OUT.println("Phase 4");
  }
  c.drop();
}
```

Monte Carlo Pi

Sequential Monte Carlo Pi

```
package examples;
import x10.util.Random;

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



Parallel Monte Carlo Pi with Atomic

```
public class ParPi {
  public static def main(args:Rail[String]) {
    val N = Int.parse(args(0)); val P = Int.parse(args(1));
    var result:Double = 0;
    finish for(1..P) async {
      val myRand = new Random();
      var myResult:Double = 0;
      for(1..(N/P)) {
        val x = myRand.nextDouble();
        val y = myRand.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);
  }
}
```

IBM

Parallel Monte Carlo Pi with Collecting Finish

```
public class CollectPi {
  public static def main(args:Rail[String]) {
    val N = Int.parse(args(0)); val P = Int.parse(args(1));
    val result = finish(Reducible.SumReducer[Double]()) {
      for(1..P) async {
        val myRand = new Random();
        var myResult:Double = 0;
        for(1..(N/P)) {
          val x = myRand.nextDouble();
          val y = myRand.nextDouble();
          if(x*x + y*y <= 1) myResult++;
        }
        offer myResult;
      }
    };
    val pi = 4*result/N;
    Console.OUT.println("The value of pi is " + pi);
  }
}
```

Implementation Highlights

Execution Strategy

- One process per place
  - one thread pool per place with X10_NTHREADS active worker threads
- Work-stealing scheduler
  - per-worker deque of pending tasks (double-ended queue)
  - idle worker steals from others
- Local finish implemented as one synchronized counter
  - *very different story with multiple places*
  - fork-join optimization: thread blocked on finish executes subtasks if any
- atomic and when implemented with one per-place lock and thread parking
  - OS-level thread count varies dynamically to compensate for parked threads
- Collecting finish implemented with thread-local storage

IBM

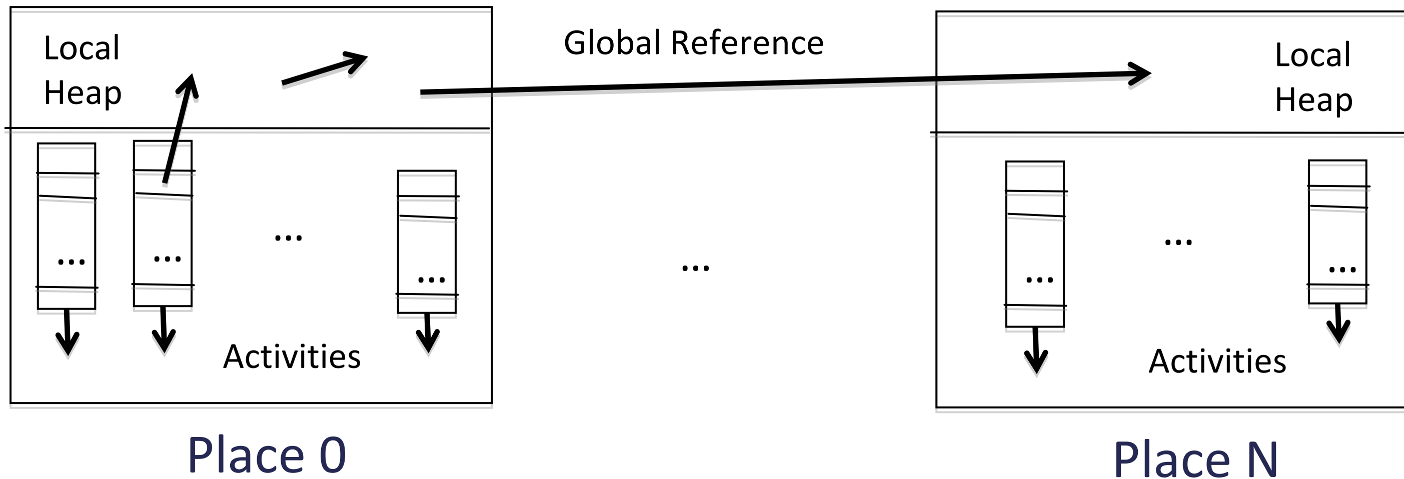
Gotchas

- Avoid too small tasks
  - fib is not a good example!
- Create enough tasks
  - especially when irregular in duration
- Avoid synchronizations
  - stick to finish as much as possible
- When conditions must be updated atomically
- Set X10_NTHREADS to the number of cores available (to the place)
- Console.OUT and Console.ERR are not atomic

IBM

Part 3

APGAS in X10: Places and Tasks



Task parallelism

- `async S`
- `finish S`

Place-shifting operations

- `at(p) S`
- `at(p) e`

Concurrency control within a place

- `when(c) S`
- `atomic S`

Distributed heap

- `GlobalRef[T]`
- `PlaceLocalHandle[T]`

Distribution

Distribution: Places

An X10 application runs with a fixed number of places decided at launch time

x10.lang.Place

- The available places are numbered from 0 to Place.MAX_PLACES-1
- for(p in Place.places()) iterates over all the available places
- here always evaluates to the current place
- Place(n) is the nth place
- If p is a place then p.id is the index of place p
- *Each place has its own copy of static variables*
- *Static variables are initialized per place and per variable at the first access*

The main method is invoked at place Place(0)
Other places are initially idle

X10 programs are typically parametric in the number of places

Distribution: at

A task can "shift" place using at

- at(p) S
  - executes statement S at place p
  - current task is blocked until S completes
  - S may spawn async tasks
    - at does not wait for these tasks
    - the enclosing finish does

- at(p) e
  - evaluates expression e at place p and returns the computed value

- at(p) async S
  - creates a task at place p to run S
  - returns immediately

HelloWholeWorld.x10

```
class HelloWholeWorld {
  public static def main(args:Rail[String]) {
    finish
      for(p in Place.places())
        at(p) async
          Console.OUT.println(p + " says " + args(0));
    Console.OUT.println("Bye");
  }
}
```

```
$ 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
Bye
```

IBM

Distributed Object Model

- Objects live in a single place
  - an object belong to the place of the task that constructed the object
  - objects can only be accessed in the place where they live
  - tasks must shift place accordingly
- Object references are always local
  - rail:Rail[Int] refers to a rail in the current place (if not null)
- Global references (possibly remote) have to be constructed explicitly
  - val ref:GlobalRef[Rail[Int]] = GlobalRef(rail);
- Global references can only be dereferenced at the place of origin "home"
  - at(ref.home) Console.OUT.println(ref());
  - at(ref) Console.OUT.println(ref()); // shorthand syntax
  - ref as GlobalRef[T]{self.home==here} // place cast

IBM

At: Scopes and Copy Semantics

Scopes

- S in at(p) cannot refer to local variables, can refer to local values

Copy semantics

- at copies the reachable local object graph to the target place
  - the compiler identifies the values declared outside of S and accessed inside of S
  - the runtime serializes and sends the graph reachable from these values
  - the runtime recreates an isomorphic graph at the destination place

- But blindly copying is not always the right thing to do
  - ids of GlobalRefs are serialized, not content
  - instances field declared transient are not copied
  - classes may implement custom serialization with arbitrary behavior
  - optimized copy methods for arrays (non-reference types)

GlobalRef[T] and PlaceLocalHandle[T]

GlobalRef[T]

- is a reference possibly remote
  - T must be a reference type (not a struct)
  - val ref:GlobalRef[List] = GlobalRef(myList);
- is the basis of all remote things
  - GlobalCell[T] is a GlobalRef[Cell[T]] (for when T is a struct type)
  - GlobalRail[T] is a GlobalRef[Rail[T]] plus a size to permit source-side bounds checks

PlaceLocalHandle[T]

- is a global handle to per-place objects of type T
  - T must be a reference type (not a struct)
  - val plh = PlaceLocalHandle.make(Place.places(), ()=>new Rail[Int](N));
- is a kind of optimized collection of GlobalRef[T]
- is the basis of all distributed data-structures

DistRail.x10

```
public class DistRail[T](size:Long) {
  protected val chunk:Long;
  protected val raw:PlaceLocalHandle[Rail[T]];

  public def this(size:Long){T haszero} {
    property(size);
    assert(size%Place.MAX_PLACES == 0); // to keep it simple
    val chunk = size/Place.MAX_PLACES; this.chunk = chunk;
    raw = PlaceLocalHandle.make[Rail[T]](Place.places(), ()=>new Rail[T](chunk));
  }

  public operator this(i:Long) = (v:T) { at(Place(i/chunk)) raw()(i%chunk) = v; }

  public operator this(i:Long) = at(Place(i/chunk)) raw()(i%chunk);

  public static def main(Rail[String]) {
    val v = new DistRail[Long](256);
    v(135) = Place.MAX_PLACES; Console.OUT.println(v(135));
  }
}
```

IBM

PlaceLocalHandle[T] and Copying Semantics

```
package examples;

public class PLH {
  static val places = Place.places();
  static val c =
    PlaceLocalHandle.make[Cell[Long]](places, ()=>new Cell[Long](-1));

  static public def main(Rail[String]) {
    for(p in places) at(p) { c()(p.id); }

    Console.OUT.println("static");
    for(p in places) at(p) { Console.OUT.println(here.id + " " + c()()); }

    val c = PLH.c;
    Console.OUT.println("local");
    for(p in places) at(p) { Console.OUT.println(here.id + " " + c()()); }
  }
}
```

```
$ x10c PLH.x10
$ X10_NPLACES=4 x10 examples.PLH
static
0 0
1 1
2 2
3 3
local
0 0
1 -1
2 -1
3 -1
```

	p0	p1	p2	p3
c0	0	-1	-1	-1
c1	-1	1	-1	-1
c2	-1	-1	2	-1
c3	-1	-1	-1	3

DistArraySum.x10

```
import x10.array.*;
public class DistArraySum {
  static N = 10;
  static def sumSimple(a:DistArray[Double]):Double {
    var sum:Double = 0;
    for(pt in a) sum += at(a.place(pt)) a(pt);
    return sum;
  }
  static def sumOpt(a:DistArray_BlockBlock_2[Double]):Double {
    val sum = finish(Reducible.SumReducer[Double]()) {
      for(p in a.placeGroup()) at(p) async {
        var localSum:Double = 0;
        // for(pt in a.localIndices()) localSum += a(pt);
        for([i,j] in a.localIndices()) localSum += a(i,j);
        offer localSum;
      }
    };
    return sum;
  }
  public static def main(Rail[String]) {
    val a = new DistArray_BlockBlock_2[Double](N, N, (i:Long,j:Long)=>(i+j) as Double);
    Console.OUT.println("Sum: " + sumOpt(a));
  }
}
```

Distributed Monte Carlo Pi

Parallel Monte Carlo Pi with Atomic

```
public class ParPi {
  public static def main(args:Rail[String]) {
    val N = Int.parse(args(0)); val P = Int.parse(args(1));
    var result:Double = 0;
    finish for(1..P) async {
      val myRand = new Random();
      var myResult:Double = 0;
      for(1..(N/P)) {
        val x = myRand.nextDouble();
        val y = myRand.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);
  }
}
```

IBM

Distributed Monte Carlo Pi with Atomic and GlobalRef

```
public class DistPi {
  public static def main(args:Rail[String]) {
    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 myRand = new Random();
      var myResult:Double = 0;
      for(1..(N/Place.MAX_PLACES)) {
        val x = myRand.nextDouble();
        val y = myRand.nextDouble();
        if(x*x + y*y <= 1) myResult++;
      }
      val myFinalResult = myResult;
      at(result) async atomic result()() += myFinalResult;
    }
    val pi = 4*result()()/N;
    Console.OUT.println("The value of pi is " + pi);
  }
}
```

Parallel Monte Carlo Pi with Collecting Finish

```
public class CollectPi {
  public static def main(args:Rail[String]) {
    val N = Int.parse(args(0)); val P = Int.parse(args(1));
    val result = finish(Reducible.SumReducer[Double]()) {
      for(1..P) async {
        val myRand = new Random();
        var myResult:Double = 0;
        for(1..(N/P)) {
          val x = myRand.nextDouble();
          val y = myRand.nextDouble();
          if(x*x + y*y <= 1) myResult++;
        }
        offer myResult;
      }
    };
    val pi = 4*result/N;
    Console.OUT.println("The value of pi is " + pi);
  }
}
```

Distributed Monte Carlo Pi with Collecting Finish

```
public class MontyPi {
  public static def main(args:Rail[String]) {
    val N = Int.parse(args(0));
    val result = finish(Reducible.SumReducer[Double]()) {
      for(p in Place.places()) at(p) async {
        val myRand = new Random();
        var myResult:Double = 0;
        for(1..(N/Place.MAX_PLACES)) {
          val x = myRand.nextDouble();
          val y = myRand.nextDouble();
          if(x*x + y*y <= 1) myResult++;
        }
        offer myResult;
      }
    };
    val pi = 4*result/N;
    Console.OUT.println("The value of pi is " + pi);
  }
}
```

Implementation Highlights

X10RT

The X10 runtime is built on top of a transport API called X10RT

- X10RT abstracts network details to enable X10 on a range of systems
- We provide several implementations of X10RT
  - standalone (shared mem), sockets (TCP/IP), PAMI, DCMF, MPI, *CUDA*
  - X10RT implementation is chosen at application compile time (-x10rt <impl> option) (cf. at runtime for Java backend)
  - Each X10RT backend is tied to a launcher
    - custom launcher for sockets and standalone
    - mpirun for MPI, poe or loadleveler for PAMI, etc.
    - ad hoc configuration (number of places, mapping from places to hosts…)
- Core API for active messages
- Optional API for direct array copies and collectives
  - emulation layer

Implementation Highlights

- at(p) async
  - source side: synthetize active message
    - async id + serialized heap + control state (finish, clocks)
    - compiler identifies captured variables (roots)
    - runtime serializes heap reachable from roots
  - destination side: decode active message
    - polling (when idle + on runtime entry)
    - incoming task pushed to worker's deque
- at(p)
  - implemented as "at(p) async" + return message
  - parent activity blocks waiting for return message
    - normal or abnormal termination (propagate exceptions and stack traces)
- Distributed finish
  - complex and potentially costly due to message reordering

to be continued…

Gotchas

- Prefer “at(p) async” to “async at(p)”
  - p in “async at(p)” is computed in parallel with parent task (unless constant)
  - “async at(p)” may require new tasks both at the source and destination places
- Don’t capture this
  - referring to a field of this in an “at” pulls the entire object across
- Be fair!
  - non-preemptive scheduler: long sequential loops can prevent servicing the network
    - prevent message to be received and processed and send (due to chunking)
  - break long sequential computation with invocations of Runtime.x10rtProbe()
- Objects exposed as GlobalRefs are not collected (for now…) (cf. collected for Java backend)
  - immortal by default
  - alternatively classes implementing the x10.lang.Runtime.Mortal interface are collected irrespective of remote reference (➔ back to manual lifetime management)

APGAS Idioms

- Remote evaluation

```
v = at(p) evalThere(arg1, arg2);
```

- Active message

```
at(p) async runThere(arg1, arg2);
```

- Recursive parallel decomposition

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

- SPMD

```
finish for(p in Place.places()) {
  at(p) async runEverywhere();
}
```

- Atomic remote update

```
at(ref) async atomic ref() += v;
```

- Data exchange

```
// swap l() local and r() remote
val _l = l();
finish at(r) async {
  val _r = r();
  r() = _l;
  at(l) async l() = _r;
}
```

Part 4

SPMD Computations at Scale

Scalable HelloWholeWorld.x10?

```
class HelloWholeWorld {
  public static def main(args:Rail[String]) {
    finish
      for(p in Place.places())
        at(p) async
          Console.OUT.println(p + " says " + args(0));
    Console.OUT.println("Bye");
  }
}
```

Problems at scale

- finish does not scale
- sequential for loop does not scale
- serializing more data than necessary

IBM

Toward a Scalable HelloWholeWorld

```
class HelloWholeWorld {
  public static def main(args:Rail[String]) {
    val arg = args(0);
    finish
      for(p in Place.places())
        at(p) async
          Console.OUT.println(here + " says " + arg);
    Console.OUT.println("Bye");
  }
}
```

Step 1

- Optimize serialization to reduce message size

- Compiler will eventually do the right thing automatically

IBM

Toward a Scalable HelloWholeWorld

```
class HelloWholeWorld {
  public static def main(args:Rail[String]) {
    val arg = args(0);
    @Pragma(Pragma.FINISH_SPMD) finish
     for(p in Place.places())
      at(p) async
       Console.OUT.println(here + " says " + arg);
    Console.OUT.println("Bye");
  }
}
```

Step 2

- Optimize finish implementation via Pragma annotation
  - See x10.compiler.Pragma class

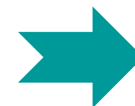
- Compiler will eventually do the right thing automatically

Scalable Distributed Termination Detection

- Distributed termination detection is hard
  - arbitrary message reordering

- Base algorithm
  - one row of *n* counters per place with *n* places
  - increment on spawn, decrement on termination, message on decrement
  - finish triggered when sum of each column is zero

- Optimized algorithms
  - local aggregation and message batching (up to local quiescence)
  - pattern-based specialization
    - local finish, SPMD finish, ping pong, single async
  - software routing
  - uncounted asyncs
  - runtime optimizations + static analysis + pragmas



scalable finish

Toward a Scalable HelloWholeWorld

```
class HelloWholeWorld {
  public static def main(args:Rail[String]) {
    val arg = args(0);
    @Pragma(Pragma.FINISH_SPMD) finish
     for(var i:Long=Place.MAX_PLACES-1; i>=0; i-=32) at(Place(i)) async {
       val max = here.id; val min = Math.max(max-31, 0);
       @Pragma(Pragma.FINISH_SPMD) finish
        for(j in min..max) at(Place(j)) async
          Console.OUT.println(here + " says " + arg);
     }
    Console.OUT.println("Bye");
  }
}
```

Step 3

- Parallelize for loop… this is getting complicated!

Toward a Scalable HelloWholeWorld

```
class HelloWholeWorld {
  public static def main(args:Rail[String]) {
    val arg = args(0);
    Place.places().broadcastFlat(()=>{
      Console.OUT.println(here + " says " + arg);
    });
    Console.OUT.println("Bye");
  }
}
```

Final step

- Abstract pattern as a library method!
- broadcastFlat encapsulates the chunked loop and pragmas

Communication Optimizations

- Copy-avoidance and RDMAs
  - efficient remote memory operations
  - fundamentally asynchronous
    - async semantics

```
Array.asyncCopy[Double](src, srcIndex, dst, dstIndex, size);
```

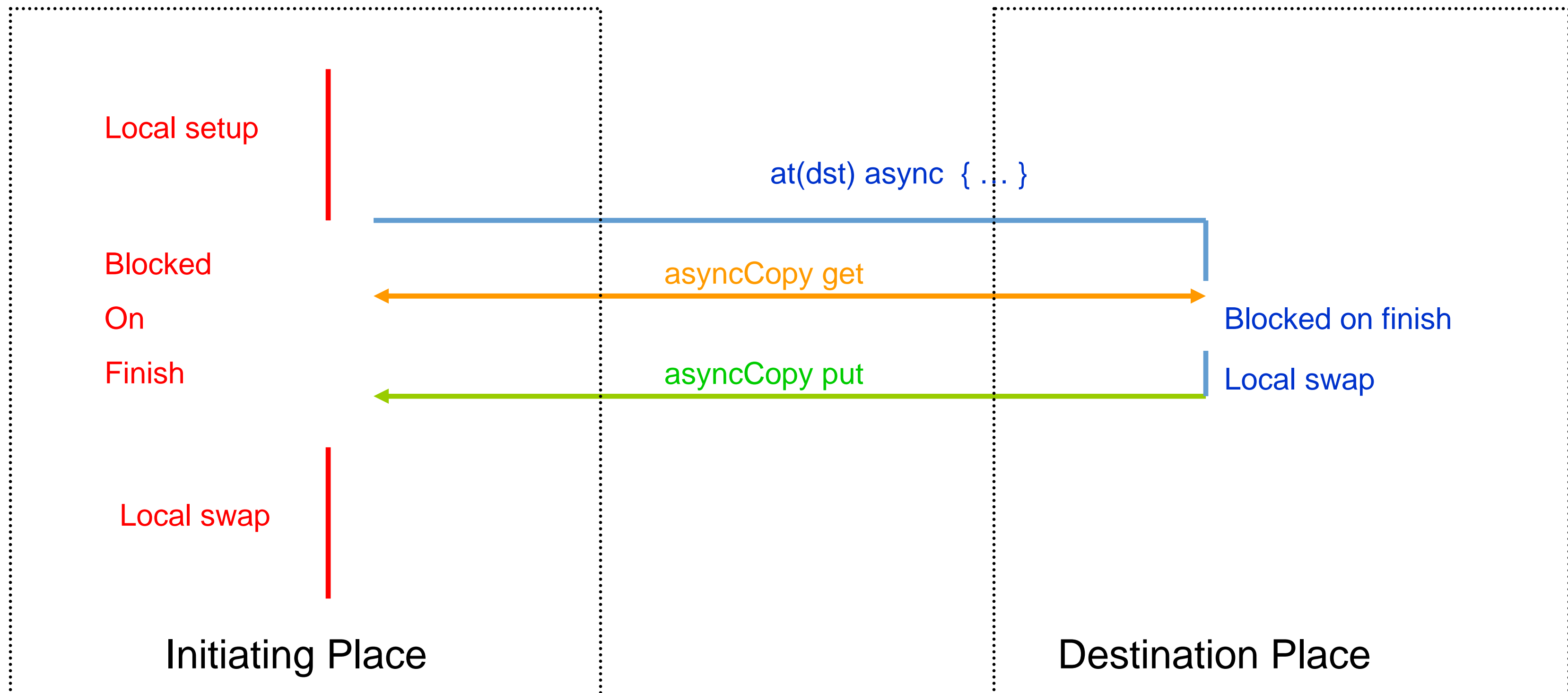
- Collectives
  - multi-point coordination and communication
  - all kinds of restrictions today

```
Team.WORLD.barrier();
val columnTeam = Team.WORLD.split(here.id%p, here.id/p);
columnTeam.allReduce(localMax, Team.MAX);
```

- bright future (MPI-3 and beyond)

Row Swap from Linpack Benchmark

- Programming problem
  - Efficiently exchange rows in distributed matrix with another Place
  - Exploit network capabilities



Row Swap from Linpack Benchmark

```
// swap row with index srcRow located here with row dstRow located at place dst
// val matrix:PlaceLocalHandle[Matrix[Double]];
// val buffers:PlaceLocalHandle[Rail[Double]];

def rowSwap(srcRow:Int, dstRow:Int, dst:Place) {
  val srcBuffer = buffers();
  val srcBufferRef = GlobalRail(srcBuffer);
  val size = matrix().getRow(srcRow, srcBuffer);
  finish {
    at(dst) async {
      val dstBuffer = buffers();
      finish {
        Array.asyncCopy[Double](srcBufferRef, 0, dstBuffer, 0, size);
      }
      matrix().swapRow(dstRow, dstBuffer);
      Array.asyncCopy[Double](dstBuffer, 0, srcBufferRef, 0, size);
    }
  }
  matrix().setRow(srcRow, srcBuffer);
}
```

Unbalanced Computations at Scale

Unbalanced Tree Search

- Problem
  - count nodes in randomly generated tree
  - separable cryptographic random number generator

    childCount = f(nodeId)

    childId = SHA1(nodeId, childIndex)

  ➔ highly unbalanced trees

  ➔ unpredictable

  ➔ tree traversal can be relocated (no data dependencies, no locality)

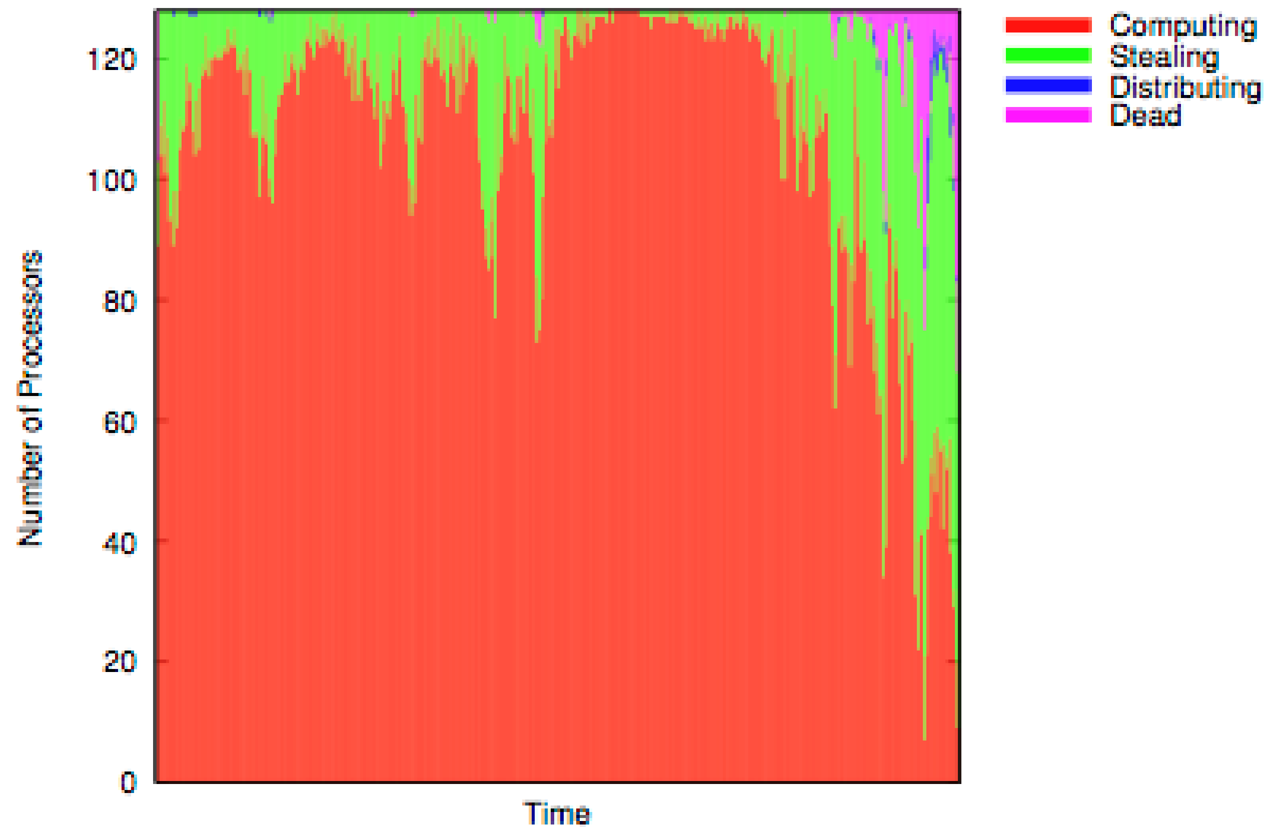
- Strategy
  - dynamic distributed load balancing
    - effectively move work (node ids) from busy nodes to idle nodes
    - deal? steal? startup?
  - effectively detect termination

Lifeline-based Global Work Stealing

- One task in each place
  - maintains a list of pending nodes and processes list until empty
  - then tries to steal nodes from other places
- Random steal attempts then lifelines
  - first try to synchronously steal from randomly selected victims a few times
  - if fail then steal from preselected lifelines
  - if fail then die but lifelines remember requests
    - lifeline “resurrects” task if more nodes become available
    - lifeline distribute nodes to resurrected task
- Lifeline graph needs low diameter, low degree ➔ hypercubes

- finish just works!
  - if done then all tasks die and root finish returns
  - root finish only accounts for initial tasks and resurrected tasks
  - communications for random steal attempts need not be tracked

Life Cycle



Main Loop (Sketch)

```
def process() {
  alive = true;
  while(!empty()) {
    while(!empty()) { processAtMostN(); Runtime.probe(); deal(); }
    steal();
  }
  alive = false;
}

def steal() {
  val h = here.id;
  for(1..w) {
    if(!empty()) break;
    finish at(Place(victims(rnd.nextInt(m)))) async request(h, false);
  }
  for(lifeline in lifelines) {
    if(!empty()) break;
    if(!lifelinesActivated(lifeline)) {
      lifelinesActivated(lifeline) = true;
      finish at(Place(lifeline)) async request(h, true);
} } }
```

Handling Thieves (Sketch)

```
def request(thief:Int, lifeline:Boolean) {
 val nodes = take(); // grab nodes from the local queue
 if(nodes == null) {
  if(lifeline) lifelineThieves.push(thief);
  return;
 }
 at(Place(thief)) async {
  if(lifeline) lifelineActivated(thief) = false;
  enqueue(nodes); // add nodes to the local queue
} }

def deal() {
 while(!lifelineThieves.empty()) {
  val nodes = take(); // grab nodes from the local queue
  if(nodes == null) return;
  val thief = lifelineThieves.pop();
  at(Place(thief)) async {
   lifelineActivated(thief) = false;
   enqueue(nodes); // add nodes to the local queue
   if(!alive) process();
} } }
```

Miscellaneous

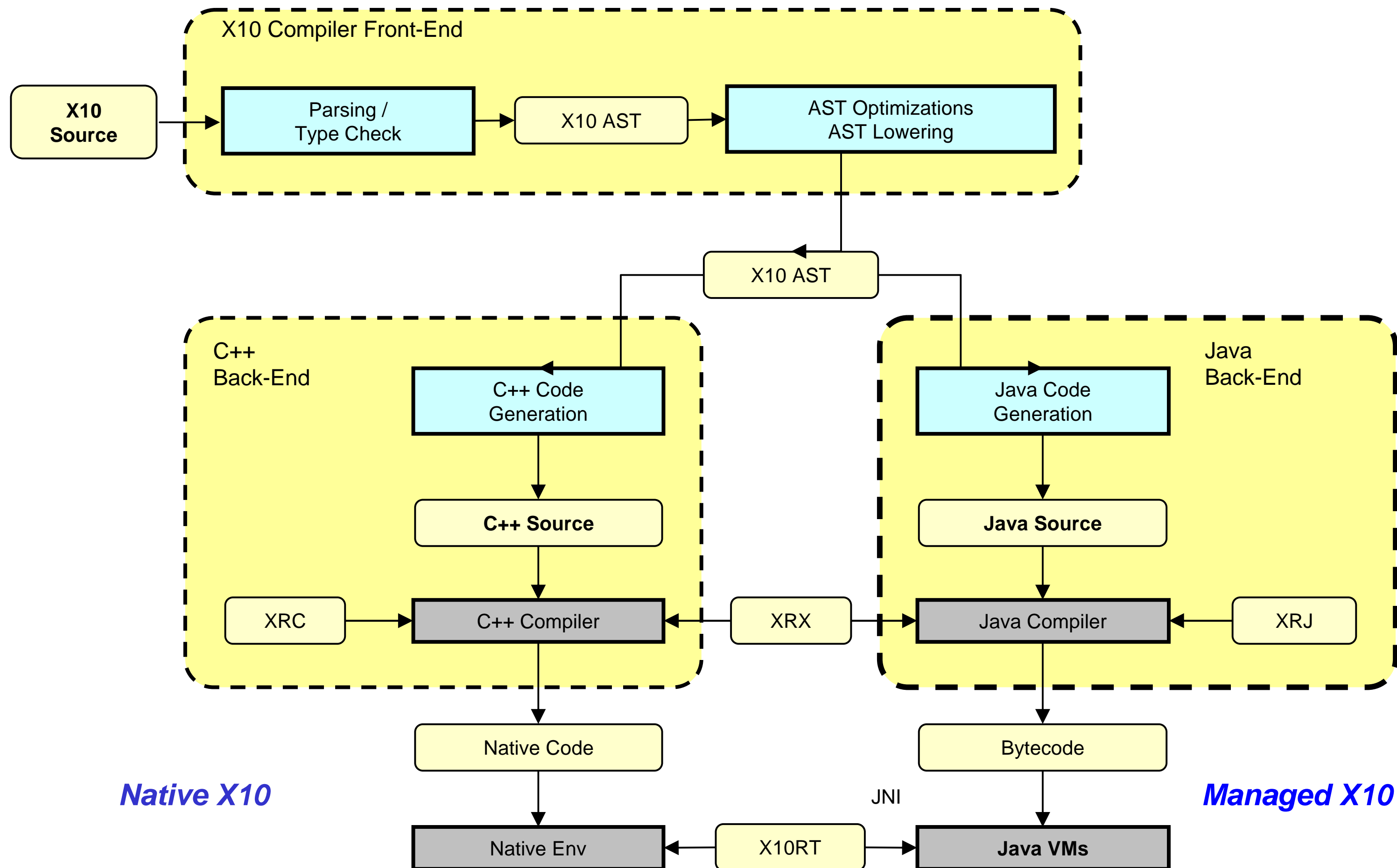
IBM

Java vs. C++ as Implementation Substrate

- Java
  - just-in-time compilation (blessing & curse)
  - sophisticated optimizations and runtime services for OO language features
  - straying too far from Java semantics can be quite painful
  - implementing a language runtime in vanilla Java is limiting
    - no benefits from structs for instance
- C++
  - ahead-of-time compilation (blessing & curse)
  - minimal optimization of OO language features
  - implementing language runtime layer
    - Ability to write low-level/unsafe code (flexibility)
    - Much fewer built-in services to leverage (blessing & curse)

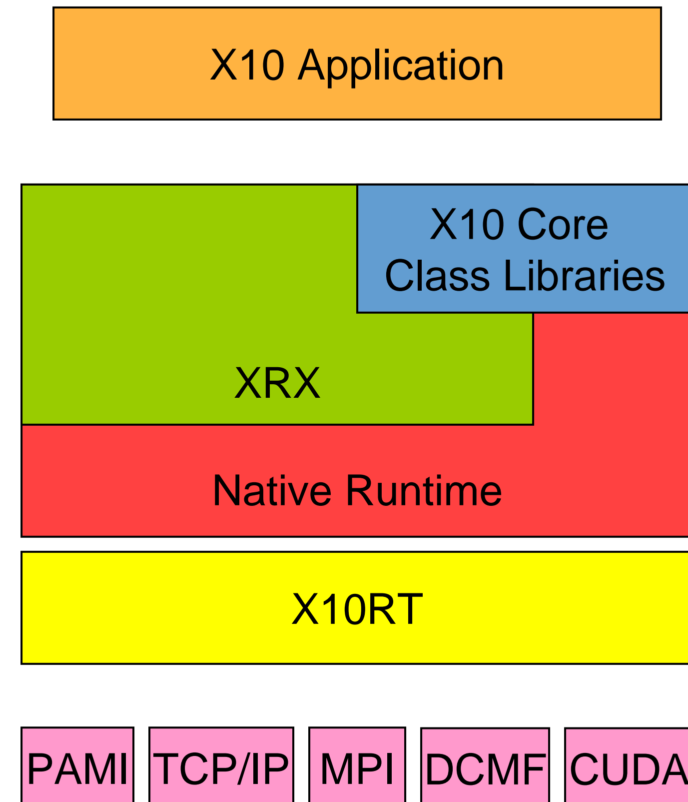
Dual path increases effort and constrains language design but also widens applicability and creates interesting opportunities

X10 Compilation



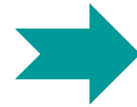
X10 Runtime

- X10RT (X10 runtime transport)
  - active messages, collectives, RDMAs
  - implemented in C; emulation layer
    (cf. pure Java version is under development)
- Native runtime
  - processes, threads, atomic operations
  - object model (layout, rtt, serialization)
  - two versions: C++ and Java
- XRX (X10 runtime in X10)
  - implements APGAS: async, finish, at
  - X10 code compiled to C++ or Java
- Core X10 libraries
  - x10.array, io, util, util.concurrent



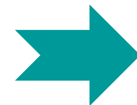
Memory Management

- Garbage collector
  - problem 1: distributed heap
  - solution: segregate local/remote refs
    - GC for local refs; distributed GC experiment (cf. Java backend supports distributed GC)
  - problem 2: risk of overhead and jitter
  - solution: maximize memory reuse…



not an issue in practice

- Congruent memory allocator
  - problem: not all pages are created equal
    - large pages required to minimize TLB misses
    - registered pages required for RDMAs
    - congruent addresses required for RDMAs at scale
  - solution: dedicated memory allocator
    - configurable congruent registered memory region
      - backed by large pages if available
      - only used for performance critical arrays



issue is contained

C Bindings

```
// essl_natives.h
void blockMulSub(double*, double*, double*, int);

// essl_natives.cc
void blockMulSub(double* me, double* left, double* upper, signed int B)
{
  double alpha = -1.0;
  double beta = 1.0;
  dgemm_("N", "N", &B, &B, &B, &alpha, upper, &B, left, &B, &beta, me, &B);
}

// LU.x10
import x10.compiler.*;

@NativeCPPInclude("essl_natives.h")
@NativeCPPCompilationUnit("essl_natives.cc")
class LU {
  @NativeCPPExtern
  native static def blockMulSub(me:Rail[Double], left:Rail[Double], upper:Rail[Double], B:Int):void;
  ...

  // Use of blockMulSub
  blockMulSub(block, left, upper, B);
  ...
```

Java Bindings

- The same annotation-based mechanisms work for Java backend
  - @Native for methods, fields, and statements
  - @NativeRep for types

- In addition, Java backend supports compiler-supported external Java linkage mechanism based on X10-Java integrated type system
  - Normal Java statements can be mixed in X10 code

```
//  X10 program that accesses relational database with JDBC (Java Database Connectivity) API
val c = java.sql.DriverManager.getConnection("jdbc:derby:test");
val s = c.createStatement();
val rs = s.executeQuery("SELECT num, addr FROM location");
while (rs.next()) {
    val num = rs.getInt(1);
    val addr = rs.getString(2);
    Console.OUT.println("num=" + num + ", addr=" + addr);
}
c.commit();
```

Wrap Up

Final Thoughts

- X10 Approach
  - Augment full-fledged modern language with core APGAS constructs
  - Enable programmer to evolve code from prototype to scalable solution

  - Problem selection: do a few key things well, defer many others
  - Mostly a pragmatic/conservative language design (except when it is not…)

- X10 2.4 (today) is not the end of the story
  - A base language in which to build higher-level frameworks
    (Global Matrix Library, Main-Memory Map Reduce, ScaleGraph)
  - A target language for compilers (MatLab, stencil DSLs)

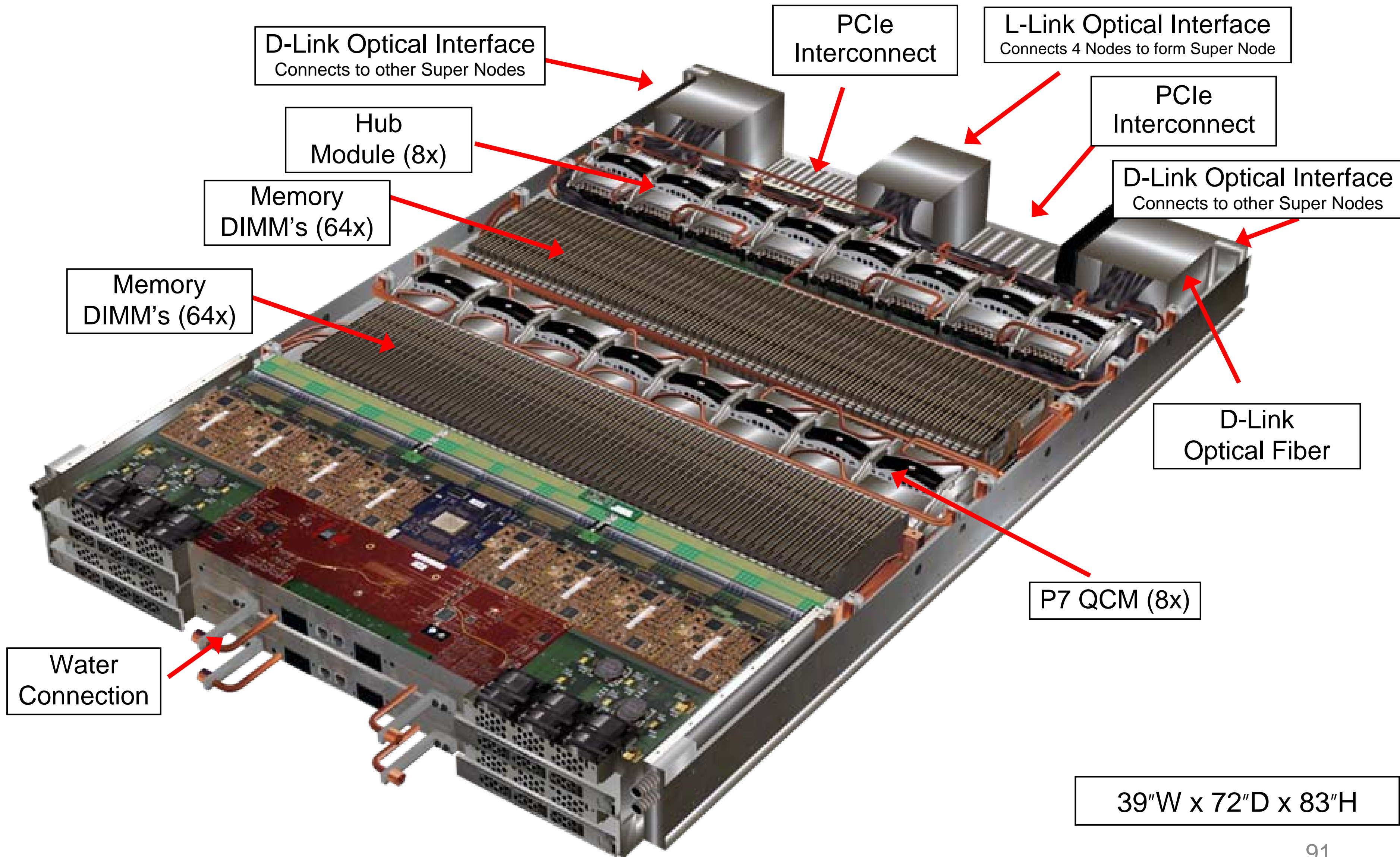
  - APGAS runtime: X10 runtime as Java and C++ libraries
  - APGAS programming model in other languages

Benchmarks

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

Power 775 Drawer



39″W x 72″D x 83″H

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

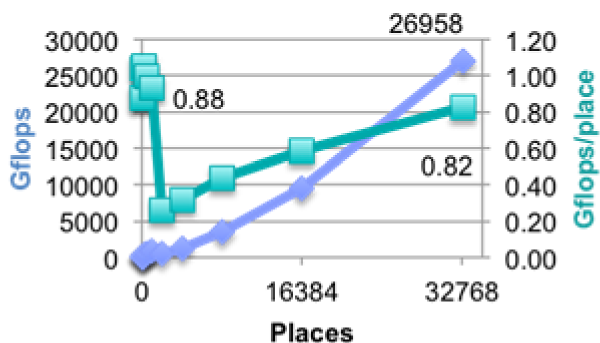
- *One core = one place = one main async*
- *Native libraries for sequential math kernels: ESSL, FFTW, SHA1*

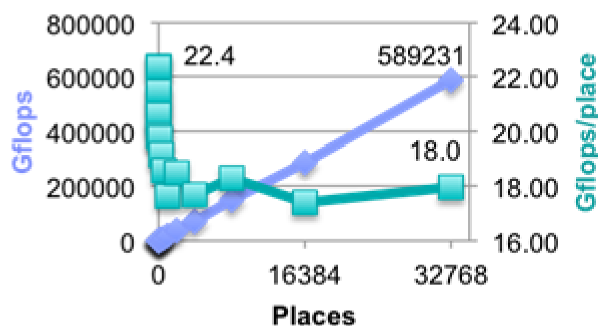
Performance at Scale (Weak Scaling)

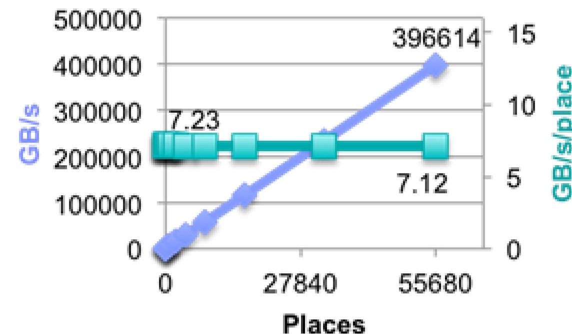
	cores	absolute performance at scale	parallel efficiency (weak scaling)	performance relative to best implementation available
Stream	55,680	397 TB/s	98%	85% (lack of prefetching)
FFT	32,768	27 Tflops	93%	40% (no tuning of seq. code)
Linpack	32,768	589 Tflops	80%	80% (mix of limitations)
RandomAccess	32,768	843 Gups	100%	76% (network stack overhead)
KMeans	47,040	depends on parameters	97.8%	66% (vectorization issue)
SSCA1	47,040	depends on parameters	98.5%	100%
SSCA2	47,040	245 B edges/s	> 75%	no comparison data
UTS (geometric)	55,680	596 B nodes/s	98%	*reference code does not scale* *4x to 16x faster than UPC code*

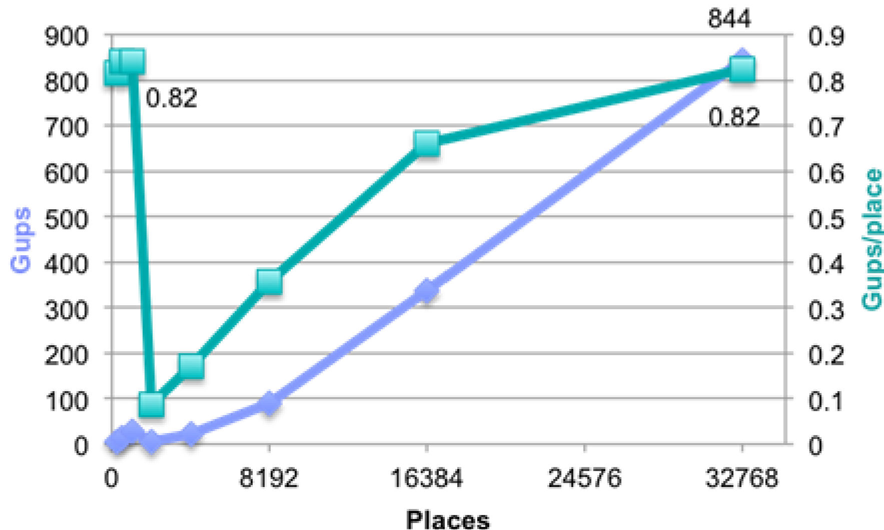
IBM

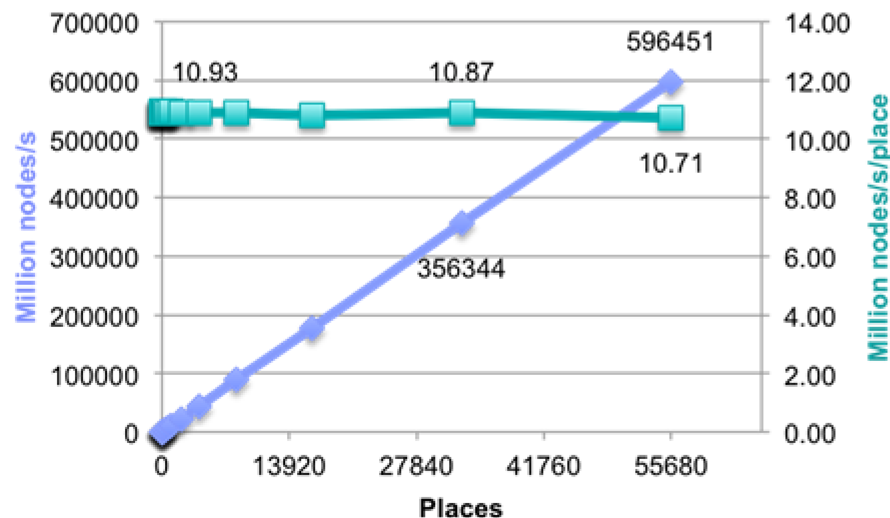
HPCC Class 2 Competition: Best Performance Award











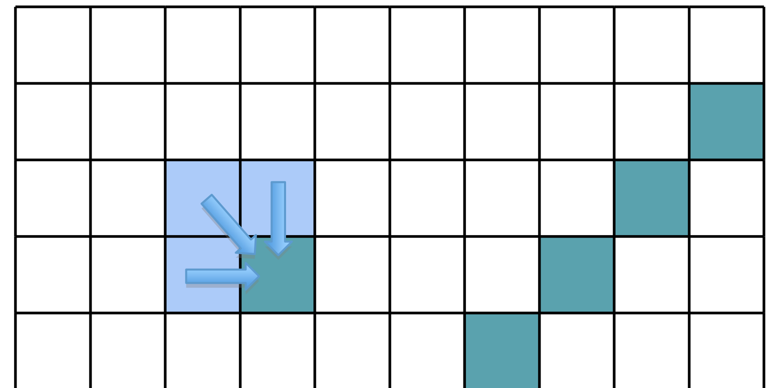
IBM

Exercise

Smith-Waterman

- Problem
  - local sequence alignment of DNA sequences
  - find similar regions in two strings
    - match, mismatch, insertion, deletion
- Strategy
  - sequential algorithm
    - dynamic programming
  - parallelization and distribution
    - if short string size << long string size
      - split long string with overlap
      - easy to distribute
    - if comparable length
      - parallel wave front
      - distribution?

Scoring matrix up to (p, q)



$$s(p,q) = f(s(p-1,q),s(p,q-1),s(p-1,q-1))$$