X10: A High-Productivity Approach to Programming Multi-Core Systems

x10.sf.net

Vivek Sarkar
vsarkar@us.ibm.com
Senior Manager, Programming Technologies
IBM T.J. Watson Research Center

This work has been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004.
Acknowledgments

**X10 Core Team**
- Rajkishore Barik
- Chris Donawa
- Allan Kielstra
- Nate Nystrom
- Igor Peshansky
- Christoph von Praun
- Vijay Saraswat
- Vivek Sarkar
- Tong Wen

**X10 Tools**
- Philippe Charles
- Robert Fuhrer
- Stan Sutton

**Emeritus**
- Kemal Ebcioglu
- Christian Grothoff

**X10 Publications**


**X10 tutorials**
Programming Technologies Research at IBM

Goal: Focus our research on core technologies for development, deployment, and execution of programs and related software assets

Focus Research Areas and Current Projects:

• Programming Models and Programming Language Design
  - Collage, DALI/XJ, X10

• Development Tools
  - CSQ (includes SAFE, Security Analysis, Scripting Analysis), Parallel Tools, SAFARI

• Deployment, Execution, Optimization
  - Dynamic Optimization, Jikes RVM, Metronome, PDS/Mirage,
PERCS Programming Model, Tools and Compilers (Productive Easy-to-use Reliable Computer System)

**Eclipse platform**

- X10 source code
- Java™ source code (w/ threads & conc util)
- C/C++ source code (w/ MPI, OpenMP, UPC)
- Fortran source code (w/ MPI, OpenMP)

<table>
<thead>
<tr>
<th>Productivity Measurements</th>
<th>Rational PurifyPlus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Refactoring for Concurrency</td>
<td>Rational Development Toolkit</td>
</tr>
<tr>
<td>Performance Explorer</td>
<td>Remote System Explorer</td>
</tr>
<tr>
<td>Parallel Tools Platform (PTP)</td>
<td>X10 Compiler</td>
</tr>
</tbody>
</table>

- X10 Components
- Java components
- C/C++ components
- Fortran components

<table>
<thead>
<tr>
<th>X10 runtime</th>
<th>Java runtime</th>
<th>C/C++ runtime</th>
<th>Fortran runtime</th>
</tr>
</thead>
</table>

- Fast extern interface

- HPC Toolkit + pSigma + Performance Tuning Automation
- Dynamic Compilation + Continuous Program Optimization
- Integrated Parallel Runtime: MPI + LAPI + RDMA + OpenMP + threads

Text in blue identifies PERCS contributions in Phase II

**PERCS**

4 X10: An Object-Oriented Approach to Non-Uniform Cluster Computing
Outline

• Software challenges for multi-core systems
• X10 Programming Model and Language
• X10 Productivity Analysis
• X10 Implementation
• Conclusions
Future Multi-Core Systems: a new Era of Mainstream Parallel Processing

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

Homogeneous Multi-core

Heterogeneous Multi-core

Multi-Core Cluster

Our response:
Use X10 as a new language for parallel hardware that builds on existing tools, compilers, runtimes, virtual machines and libraries
Current state of the art in Automatic Parallelization and Explicit Parallelism

• Automatic parallelization technologies have been successful in SIMDization, Instruction scheduling, Data privatization, …

... but whole-program automatic parallelization still eludes us

• Explicit parallel programming models have either focused on uniform shared-memory parallelism or message-passing parallelism, neither of which is appropriate for multi-core processors with a non-uniform shared memory model

  – Threaded models like OpenMP and Java have a uniform view of memory

  – Bulk-synchronous MPI model is SPMD (one process per node) with infrequent coarse-grained barriers and collective/two-sided communications

  – PGAS model (CAF, UPC, Titanium) is SPMD (one process per node) with coarse-grained barriers and fine-grained one-sided communications
Cell Programming Model
(Heterogeneous Multi-Core Parallelism)

64-bit Power Architecture with VMX

EIB (up to 96B/cycle)
Disconnect between Uniform Memory Access model and Heterogeneous Multi-Core Parallelism

DMA Commands

- **Put**: Transfer from Local Store to EA space
- **Puts**: Transfer and Start SPU execution
- **Putr**: Put Result - (Arch. Scarf into L2)
- **Putl**: Put using DMA List in Local Store
- **Get**: Transfer from EA Space to Local Store
- **Gets**: Transfer and Start SPU execution
- **Getl**: Get using DMA List in Local Store
- **Sndsig**: Send Signal to SPU

**Command Modifiers:** `<f,b>`
- **f**: Embedded Tag Specific Fence
  - Command will not start until all previous commands in same tag group have completed
- **b**: Embedded Tag Specific Barrier
  - Command and all subsequent commands in same tag group will not start until previous commands in same tag group have completed

SL1 Cache Management Commands

- **sdcrt**: Data cache region touch (DMA Get hint)
- **sdcrtst**: Data cache region touch for store (DMA Put hint)
- **sdcrz**: Data cache region zero
- **sdcrs**: Data cache region store
- **sdcrf**: Data cache region flush

Command Parameters

- **LSA**: Local Store Address (32 bit)
- **EA**: Effective Address (32 or 64 bit)
- **TS**: Transfer Size (16 bytes to 16K bytes)
- **LS**: DMA List Size (8 bytes to 16 K bytes)
- **TG**: Tag Group(5 bit)
- **CL**: Cache Management / Bandwidth Class

Synchronization Commands

**Lockline** (Atomic Update) Commands:
- **getllar**: DMA 128 bytes from EA to LS and set Reservation
- **putllc**: Conditionally DMA 128 bytes from LS to EA
- **putlluc**: Unconditionally DMA 128 bytes from LS to EA

**Barrier** - all previous commands complete before subsequent commands are started

- **mfcsync**: Results of all previous commands in Tag group are remotely visible
- **mfceieio**: Results of all preceding Puts commands in same group visible with respect to succeeding Get commands
What’s Involved in Programming for Cell?

- Partition application into PPE and SPE portions
- Map application data onto Local Store for SPE parts
  - this typically requires both temporal and spatial concerns for data that needs to be streamed in and out of Local Store
- Code SPE functions to exploit SIMD processing capabilities

```c
vector float a,b,c,d;
a=(vector float){2.3,6.0,0.0,5.1};
d = spu_madd(a,b,c); // d=a*b+c
```

- Orchestrate the data streaming using MFC commands

```c
vector float a[8];
spu_mfcdma32(a,p,128,tagnum,READ);
spu_mfcstat(ALL);
```

- Parallelize across multiple SPEs and 2 threads on the PPE
What’s Involved in Programming for Cell? (contd.)

- SIMDize if at all possible
  - reorder data, array-of-structures vs structure-of-arrays
  - either use intrinsics, or allow the compiler to do it automatically
  - Align SIMD data on 16 byte boundaries – go to any lengths!

- Try to have data in Local Store before it is needed
  - DMA latency is in the hundreds of cycles
  - SPEs are single threaded processors

- It's better to do DMA from the SPE side
  - more channels / less trouble synchronizing

- Try to reduce the amount of “branchy” code

- Using an SPE to run Scalar code is OK
  - it’s really another – application specific - processor
Outline

- Software challenges for multi-core systems
- **X10 Programming Model and Language**
- X10 Productivity Analysis
- X10 Implementation
- Conclusions
X10 Approach

- Unified abstractions of asynchrony and concurrency for use in
  - Multi-core SMP Parallelism
  - Messaging and Cluster Parallelism

- Productivity
  - High Level Language designed for portability and safety
  - X10 Development Toolkit for Eclipse

- Performance
  - Extend VM+JIT model for high performance
  - Performance transparency – don’t lock out the performance expert!
    - expert programmer should have controls to tune optimizations and tailor
data, distributions & communications to actual deployment

- Build on sequential subset of Java language
  - Retain core values of Java --- productivity, ubiquity, maturity, security
  - Target adoption by mainstream developers with Java/C/C++ skills
X10 Programming Model

Storage classes:
- Activity-local
- Place-local
- Partitioned global
- Immutable

Dynamic parallelism with a Partitioned Global Address Space
Places encapsulate binding of activities and globally addressable data
All concurrency is expressed as asynchronous activities – subsumes threads, structured parallelism, messaging, DMA transfers (beyond SPMD)
Atomic sections enforce mutual exclusion of co-located data
  - No place-remote accesses permitted in atomic section
  - Immutable data offers opportunity for single-assignment parallelism

Deadlock safety: any X10 program written with async, atomic, finish, foreach, ateach, and clocks can never deadlock
X10 Language

- **async** [(Place)] [clocked(c...)] Stm
  - Run Stm asynchronously at Place

- **finish** Stm
  - Execute s, wait for all asyncs to terminate (generalizes join)

- **foreach** (point P : Reg) Stm
  - Run Stm asynchronously for each point in region

- **ateach** (point P : Dist) Stm
  - Run Stm asynchronously for each point in dist, in its place.

- **atomic** Stm
  - Execute Stm atomically

- **new T**
  - Allocate object at this place (here)

- **new T[d] / new T value [d]**
  - Array of base type T and distribution d

- **Region**
  - Collection of index points, e.g.
    region r = [1:N,1:M];

- **Distribution**
  - Mapping from region to places, e.g.
    - dist d = block(r);

- **next**
  - suspend till all clocks that the current activity is registered with can advance

- **future** [(Place)] [clocked(c...)] Expr
  - Compute Expr asynchronously at Place

- **F. force()**
  - Block until future F has been computed

- **extern**
  - Lightweight interface to native code

**Deadlock safety:** any X10 program written with above constructs excluding future can never deadlock
- Can be extended to restricted cases of using future

15  X10: An Object-Oriented Approach to Non-Uniform Cluster Computing
X10 Arrays, Regions, Distributions

ArrayExpr:

- new ArrayType ( Formal ) { Stm }
- Distribution Expr
  -- Lifting
- ArrayExpr [ Region ]
  -- Section
- ArrayExpr | Distribution
  -- Restriction
- ArrayExpr || ArrayExpr
  -- Union
- ArrayExpr.overlay(ArrayExpr)
  -- Update
- ArrayExpr.scan( [fun [, ArgList]] )
- ArrayExpr.reduce( [fun [, ArgList]] )
- ArrayExpr.lift( [fun [, ArgList]] )

ArrayType:

- Type [Kind] [ ]
- Type [Kind] [ region(N) ]
- Type [Kind] [ Region ]
- Type [Kind] [ Distribution ]

Region:

- Expr : Expr
  -- 1-D region
- [ Range, ..., Range ]
  -- Multidimensional Region
- Region & Region
  -- Intersection
- Region | Region
  -- Union
- Region – Region
  -- Set difference
- BuiltinRegion

Dist:

- Region -> Place
  -- Constant distribution
- Distribution | Place
  -- Restriction
- Distribution | Region
  -- Restriction
- Distribution || Distribution
  -- Union
- Distribution – Distribution
  -- Set difference
- Distribution.overlay(Distribution)
- BuiltinDistribution

Language supports type safety, memory safety, place safety, clock safety.
X10 Language Constructs:
Examples

1) `finish` { // Intra-place parallelism
    final int x = ... , y = ... ;
    async a.foo(x); // Initiate two activities at same
    async b.bar(y); // place as parent activity
} // Wait for both activities to complete

2) `finish` { // Inter-place parallelism
    final int x = ... , y = ... ;
    async (a) a.foo(x); // Execute at a’s place
    async (b) b.bar(y); // Execute at b’s place
}

3) // Implicit and explicit versions of remote fetch-and-add
   a) a.x += b.y ;
   b) async (b) {final int v = b.y; async (a) atomic a.x += v; }

X10: An Object-Oriented Approach to Non-Uniform Cluster Computing
X10 Dynamic Activity Invocation Tree

// X10 pseudo code
main(){ // implicit finish
    Activity A0 (Part 1);
    async {A1; async A2;}
    try {
        finish {
            Activity A0 (Part 2);
            async A3;
            async A4;
        }
        catch (...) { ... }
    }
    Activity A0 (Part 3);
4) `future<int> F = future(a) { a.baz() };` // returns immediately
   
   . . .
   int i = F.force(); // block until return value is obtained

5) // A is a local 1-D array, B is a distributed 2-D array
   int[,] A = new int[[0:N-1]];  
   int[,] B = new int[dist.blockRows([[0:M-1,0:N-1]])];
   
   . . .
   // serial pointwise for loop
   . . .
   // intra-place pointwise parallel loop
   . . .
   // inter-place pointwise parallel loop
   ateach (point[i,j] : B.distribution) B[i,j] = h(B[i,j]);
Explicit vs. Implicit Syntax for Places

- **Explicit syntax** – target place specified explicitly for remote activity
  - `async (a) { a.z = expr ; a.foo(x); . . . }`
  - `BadPlaceException` thrown if operation is performed on remote reference

- **Implicit syntax** – freely access global address space, and let compiler insert the target places
  - `{ a.z = expr ; . . . }
  - `{ final int T = expr; `finish async(a) a.z = T; … }`
  - More convenient to write code, but harder to control and debug performance

- **X10 approach**
  - Allow combination of implicit and explicit syntax
  - Extend type system with dependent types to statically identify local operations
Example: Monte Carlo Calculations

Explicit language concurrency greatly simplifies threading related constructs.

**Multi-Threaded Java**

```java
public void runDistributed() {
    results = new x10Vector(nRunsMC);
    // Now do the computation
    finish at each (point[iRun] : tasks.distribution) {
        PriceStock ps = new PriceStock();
        ps.setInitAllTasks((ToInitAllTasks) initAllTasks);
        ps.setTask(tasks[iRun]);
        ps.run();
        final ToResult r = ps.getResult(); // ToResult is a value type
        async(results) atomic results.v.addElement(r);
    }
}
```

```java
class AppDemoThread implements Runnable {
    ... // initialization code
    public void run() {
        PriceStock ps;
        int ilow, iupper, slice;
        slice = (nRunsMC+JGFMonteCarloBench.nthreads-1) / JGFMonteCarloBench.nthreads;
        ilow = id*sacle;
        iupper = Math.min((id+1)*slice, nRunsMC);
        for( int iRun=ilow; iRun < iupper; iRun++ ) {
            ps = new PriceStock();
            ps.setInitAllTasks(this.initAllTasks);
            ps.setTask(tasks[iRun]);
            ps.run();
            AppDemo.results.addElement(ps.getResult());
        }
        // run()
    }
}
```

**Single-Threaded Java**

```java
public void runSerial() {
    results = new Vector(nRunsMC);
    // Now do the computation.
    PriceStock ps;
    for( int iRun=0; iRun < nRunsMC; iRun++ ) {
        ps = new PriceStock();
        ps.setInitAllTasks(initAllTasks);
        ps.setTask(tasks[iRun]);
        ps.run();
        results.addElement(ps.getResult());
    }
}
```

**Multi-Threaded X10**

```
public void runDistributed() {
    results = new ToTask[dist.block([0:nRunsMC-1])];
    // Now do the computation
    for( ToTask task : tasks.distribution ) {
        PriceStock ps = new PriceStock();
        ps.setInitAllTasks((ToInitAllTasks) initAllTasks);
        ps.setTask(task);
        ps.run();
        results.addElement(ps.getResult());
    }
}
```

```
initTasks() { tasks = new ToTask[nRunsMC]; … }
```
X10 Deployment

X10 language defines mapping from X10 objects & activities to X10 places

X10 deployment defines mapping from virtual X10 places to physical processing elements

---

**Homogeneous Multi-core**  
**Heterogeneous Accelerators**  
**Clusters**

---

22 X10: An Object-Oriented Approach to Non-Uniform Cluster Computing
X10 Deployment on an SMP

- Basic Approach -- partition X10 heap into multiple place-local heaps
- Each X10 object is allocated in a designated place
- Each X10 activity is created (and pinned) at a designated place
- Allow an X10 activity to synchronously access data at remote places outside of atomic sections (implicit syntax)

Thus, places serve as affinity hints for intra-SMP locality
Possible X10 Deployment for Cell (under discussion)

- Basic Approach:
  - Map 9 places on to PPE + eight SPEs
  - Use finish & async’s as high-level representation of DMAs

- Opportunities:
  - Exploit dynamic compilation and code specialization

- Challenges:
  - Weak PPE
  - SIMDization is critical
  - Lack of hardware support for coherence
  - Limited memory on SPE's
  - Limited performance of code with frequent conditional or indirect branches
  - Different ISA’s for PPE and SPE.
Outline

• Software challenges for multi-core systems
• X10 Programming Model and Language
• X10 Productivity Analysis
• X10 Implementation
• Conclusions
X10 Productivity Analysis

Scenario: parallelization of serial X10/Java code

Two sets of results

1. **Code size comparison** of serial, multi-threaded, and distributed versions of Java Grande benchmarks in Java and X10
   - Details in OOPSLA paper, “X10: An Object-Oriented Approach to Non-Uniform Cluster Computing”, (Section 5)

2. **Human productivity study** comparing *time to first correct parallel version* for C+MPI, UPC, and X10
   - Summary in P-PHEC 2006 workshop paper, "An Experiment in Measuring the Productivity of Three Parallel Programming Languages.
   - Acknowledgments for productivity study
     - Pittsburgh Supercomputing Center: Nick Nystrom, John Urbanic, Deborah Weisser
     - IBM Research Social Computing Group: Catalina Danis, Christine Halverson, Wendy Kellogg
## Code Size Comparison: Java Grande Benchmarks

<table>
<thead>
<tr>
<th>Language</th>
<th>Code size metric</th>
<th>Serial version</th>
<th>Multithreaded version</th>
<th>Distributed version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java + threads + MPI</td>
<td>Classes/SLOC</td>
<td>50/3254</td>
<td>64/4028</td>
<td>50/3973</td>
</tr>
<tr>
<td></td>
<td>SLOC ratio</td>
<td></td>
<td>1.24</td>
<td>1.22</td>
</tr>
<tr>
<td></td>
<td>SSC</td>
<td>2592</td>
<td>3212</td>
<td>3133</td>
</tr>
<tr>
<td></td>
<td>SSC ratio</td>
<td></td>
<td>1.24</td>
<td>1.21</td>
</tr>
<tr>
<td></td>
<td># stmts changed</td>
<td></td>
<td>1108</td>
<td>781</td>
</tr>
<tr>
<td></td>
<td>Change ratio</td>
<td></td>
<td>0.43</td>
<td>0.30</td>
</tr>
<tr>
<td>X10</td>
<td>Classes/SLOC</td>
<td>49/3107</td>
<td>49/3212</td>
<td>49/3243</td>
</tr>
<tr>
<td></td>
<td>SLOC ratio</td>
<td></td>
<td>1.03</td>
<td>1.04</td>
</tr>
<tr>
<td></td>
<td>SSC</td>
<td>2592</td>
<td>2649</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SSC ratio</td>
<td></td>
<td>1.04</td>
<td>1.07</td>
</tr>
<tr>
<td></td>
<td># stmts changed</td>
<td></td>
<td>363</td>
<td>510</td>
</tr>
<tr>
<td></td>
<td>Change ratio</td>
<td></td>
<td>0.15</td>
<td>0.21</td>
</tr>
</tbody>
</table>
Benchmark SLOC

Sequential Java, Java w/ Threads, Java w/ MPI
Sequential X10, 1-place X10, N-place X10

OOPSLA 05 Onwards!

X10: An Object-Oriented Approach to Non-Uniform Cluster Computing
Human Productivity Study
(Comparison of MPI, UPC, X10)

• Goals
  − Contrast productivity of X10, UPC, and MPI for a statistically significant
    subject sample on a programming task relevant to HPCS Mission Partners
  − Validate the PERCS Productivity Methodology to obtain quantitative results
    that, given specific populations and computational domains, will be of
    immediate and direct relevance to HPCS.

• Overview
  − 4.5 days: May 23-27, 2005 at the Pittsburgh Supercomputing Center (PSC)
  − Pool of 27 comparable student subjects
  − Programming task: Parallelizing the alignment portion of Smith-Waterman
    algorithm (SSCA#1)
  − 3 language programming model combinations (X10, UPC, or C + MPI)
  − Equal environment as near as possible (e.g. pick of 3 editors, simple println
    stmts for debugging)
  − Provided expert training and support for each language
Data Summary

- 180,524 source, source diff, compiler, batch, shell, web, and window events were recorded for the 27 subjects.
- Each event contains detailed information for subsequent contextual and temporal analysis.
- Example: compiler component
  - experiment and subject IDs
  - timestamp
  - compiler name
  - command line
  - number of errors and warnings
  - compiler output
  - links to source and batch records

<table>
<thead>
<tr>
<th>user_id</th>
<th>source</th>
<th>src_diff</th>
<th>compiler</th>
<th>batch</th>
<th>shell</th>
<th>web</th>
<th>window</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>132</td>
<td>353</td>
<td>179</td>
<td>168</td>
<td>1,305</td>
<td>9,775</td>
<td>2,450</td>
<td>14,362</td>
</tr>
<tr>
<td>M2</td>
<td>155</td>
<td>366</td>
<td>151</td>
<td>463</td>
<td>1,975</td>
<td>346</td>
<td>2,259</td>
<td>5,715</td>
</tr>
<tr>
<td>M3</td>
<td>237</td>
<td>465</td>
<td>330</td>
<td>107</td>
<td>1,386</td>
<td>2,736</td>
<td>2,483</td>
<td>7,744</td>
</tr>
<tr>
<td>M4</td>
<td>69</td>
<td>134</td>
<td>81</td>
<td>30</td>
<td>432</td>
<td>3,801</td>
<td>1,664</td>
<td>6,211</td>
</tr>
<tr>
<td>M5</td>
<td>119</td>
<td>271</td>
<td>139</td>
<td>24</td>
<td>608</td>
<td>299</td>
<td>1,458</td>
<td>2,918</td>
</tr>
<tr>
<td>M6</td>
<td>327</td>
<td>313</td>
<td>287</td>
<td>79</td>
<td>1,235</td>
<td>366</td>
<td>2,658</td>
<td>5,265</td>
</tr>
<tr>
<td>M7</td>
<td>409</td>
<td>1,067</td>
<td>427</td>
<td>77</td>
<td>1,920</td>
<td>1,300</td>
<td>3,691</td>
<td>8,891</td>
</tr>
<tr>
<td>M8</td>
<td>258</td>
<td>766</td>
<td>342</td>
<td>73</td>
<td>1,355</td>
<td>3,250</td>
<td>2,504</td>
<td>8,548</td>
</tr>
<tr>
<td>M9</td>
<td>116</td>
<td>247</td>
<td>160</td>
<td>59</td>
<td>875</td>
<td>913</td>
<td>1,224</td>
<td>3,594</td>
</tr>
<tr>
<td>U1</td>
<td>129</td>
<td>145</td>
<td>254</td>
<td>138</td>
<td>1,485</td>
<td>20</td>
<td>416</td>
<td>2,587</td>
</tr>
<tr>
<td>U2</td>
<td>224</td>
<td>525</td>
<td>256</td>
<td>240</td>
<td>1,669</td>
<td>733</td>
<td>2,222</td>
<td>5,869</td>
</tr>
<tr>
<td>U3</td>
<td>236</td>
<td>449</td>
<td>268</td>
<td>427</td>
<td>8,053</td>
<td>1,014</td>
<td>4,204</td>
<td>14,651</td>
</tr>
<tr>
<td>U4</td>
<td>316</td>
<td>420</td>
<td>162</td>
<td>298</td>
<td>1,301</td>
<td>580</td>
<td>1,478</td>
<td>4,555</td>
</tr>
<tr>
<td>U5</td>
<td>82</td>
<td>63</td>
<td>20</td>
<td>24</td>
<td>274</td>
<td>486</td>
<td>653</td>
<td>1,602</td>
</tr>
<tr>
<td>U6</td>
<td>297</td>
<td>388</td>
<td>179</td>
<td>227</td>
<td>1,479</td>
<td>389</td>
<td>1,691</td>
<td>4,650</td>
</tr>
<tr>
<td>U7</td>
<td>207</td>
<td>661</td>
<td>419</td>
<td>104</td>
<td>1,625</td>
<td>7,396</td>
<td>1,550</td>
<td>11,962</td>
</tr>
<tr>
<td>U8</td>
<td>244</td>
<td>500</td>
<td>238</td>
<td>303</td>
<td>7,529</td>
<td>645</td>
<td>1,563</td>
<td>11,022</td>
</tr>
<tr>
<td>U9</td>
<td>422</td>
<td>847</td>
<td>402</td>
<td>342</td>
<td>2,492</td>
<td>1,793</td>
<td>2,268</td>
<td>8,566</td>
</tr>
<tr>
<td>X1</td>
<td>767</td>
<td>354</td>
<td>645</td>
<td>0</td>
<td>2,104</td>
<td>987</td>
<td>2,056</td>
<td>6,913</td>
</tr>
<tr>
<td>X2</td>
<td>162</td>
<td>228</td>
<td>404</td>
<td>0</td>
<td>1,109</td>
<td>30</td>
<td>687</td>
<td>2,620</td>
</tr>
<tr>
<td>X3</td>
<td>766</td>
<td>329</td>
<td>432</td>
<td>0</td>
<td>1,421</td>
<td>91</td>
<td>1,419</td>
<td>4,458</td>
</tr>
<tr>
<td>X4</td>
<td>236</td>
<td>420</td>
<td>455</td>
<td>0</td>
<td>1,341</td>
<td>1,007</td>
<td>3,518</td>
<td>6,977</td>
</tr>
<tr>
<td>X5</td>
<td>680</td>
<td>251</td>
<td>669</td>
<td>0</td>
<td>1,809</td>
<td>844</td>
<td>1,753</td>
<td>6,006</td>
</tr>
<tr>
<td>X6</td>
<td>291</td>
<td>348</td>
<td>663</td>
<td>0</td>
<td>1,809</td>
<td>1,446</td>
<td>1,661</td>
<td>6,218</td>
</tr>
<tr>
<td>X7</td>
<td>238</td>
<td>484</td>
<td>595</td>
<td>0</td>
<td>1,731</td>
<td>307</td>
<td>1,396</td>
<td>4,751</td>
</tr>
<tr>
<td>X8</td>
<td>405</td>
<td>452</td>
<td>582</td>
<td>0</td>
<td>1,727</td>
<td>888</td>
<td>2,465</td>
<td>6,519</td>
</tr>
<tr>
<td>X9</td>
<td>217</td>
<td>319</td>
<td>887</td>
<td>0</td>
<td>2,634</td>
<td>1,025</td>
<td>2,268</td>
<td>7,350</td>
</tr>
<tr>
<td>total</td>
<td>7,741</td>
<td>11,165</td>
<td>9,626</td>
<td>3,183</td>
<td>52,683</td>
<td>42,467</td>
<td>53,659</td>
<td>180,524</td>
</tr>
</tbody>
</table>

Instrumented Raw Events for Full Experiment
Data Summary (contd.)

- Each thin vertical bar depicts 5 minutes of development time, colored by the distribution of activities within the interval.

- Development milestones bound intervals for statistical analysis:
  - begin/end task
  - begin/end development
  - first correct parallel output

<table>
<thead>
<tr>
<th>obtained correct parallel output</th>
<th>MPI</th>
<th>UPC</th>
<th>X10</th>
</tr>
</thead>
<tbody>
<tr>
<td>did not obtain correct parallel output</td>
<td>5</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>dropped out</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
Comparing average development times between languages, several observations are clear:

- Average development time for subjects using X10 was significantly lower than that for subjects using UPC and MPI.
- The relative time debugging was approximately the same for all languages.
- X10 programmers spent relatively more time executing code and relatively less time authoring and tidying code.
- Subjects using MPI spent more time accessing documentation (tutorials were online; more documentation is available).
- A batch environment was used in this study --- use of an interactive environment like Eclipse will probably have a significant impact on development time results.
Outline

• Software challenges for multi-core systems
• X10 Programming Model and Language
• X10 Productivity Analysis
• X10 Implementation
• Conclusions
X10 Implementation Status

Open source project on sourceforge (x10.sf.net)

- **Reference X10 implementation on a single SMP**
  - CVS repository hosted on
  - Nightly regression tests (~ 600 unit tests)
  - X10 application set starting to grow
  - Basis for Optimized SMP Implementation

- **X10 Development Toolkit (X10DT)**
  - Eclipse tools with basic X10 language support
  - X10-specific refactorings in progress
    - Extract Async
    - Introduce atomic sections
  - Built on meta-tooling framework (SAFARI)

Efforts under way this year:

- **X10 Libraries for C/C++ users on cluster of SMP nodes**
- **C/C++ code generation from X10**
- **Static Analysis and Ahead-Of-Time Optimization**
  - Concurrency analysis
  - Optimization of BadPlaceException checks and redundant async/finish ops
  - Use of static analysis to enhance X10-specific refactorings
X10 Eclipse Development Toolkit
Current Status: Multi-core SMP Implementation for X10

X10 source

X10 Grammar

X10 Parser

AST

Analysis passes

DOMO Static Analyzer

Annotated AST

Java code emitter

Code Generation Templates

Target Java

Java compiler

Common components w/ SAFARI

Place

Ready Activities

Executing Activities

Blocked Activities

Clock

Future

Inbound activities

Outbound activities

Inbound replies

Outbound replies

Place 0

Place 1

X10 libraries

Java Concurrency Utilities (JCU)

STM library

JCU thread pool

High Performance JRE
(IBM J9 VM + Testarossa JIT Compiler modified for X10 on PPC/AIX)

Portable Standard Java 5 Runtime Environment
(Runs on multiple Platforms)

Extern interface

Fortran, C/C++ DLL’s

X10 classfiles

(Java classfiles with special annotations for X10 analysis info)

Atomic sections do not have blocking semantics

Activity can only access its stack, place-local mutable data, or global immutable data

...
System Configuration used for Performance Results

- **Hardware**
  - STREAM (C/OpenMP & X10), RandomAccess (C/OpenMP & X10), FFT (X10)
    - 64-core POWER5+, p595+, 2.3 GHz, 512 GB (r28n01.pbm.ihost.com)
  - FFT (Cilk version)
    - 16-core POWER5+, p570, 1.9 GHz
  - All runs performed with page size = 4KB and SMT turned off
- **Operating System**
  - AIX v5.3
- **Compiler**
  - xlc v7.0.0.5 w/ -O3 option (also qsmp=omp for OpenMP compilation)
- **X10**
  - Dynamic compilation options: -J-Xjit:count=0,optLevel=veryHot
  - X10 activities use serial libraries written in C and linked with X10 runtime
  - Data size limitation: current X10 runtime is limited to a max heap size of 2GB
- All results reported are for runs that passed validation
  - Caveat: these results should not be treated as official benchmark measurements of the above systems
Performance Results for STREAM

Array size = $2^{26}$ elements
Combined memory for 3 arrays = 1.5GB
Performance Results for RandomAccess

Array size = 1.8GB

<table>
<thead>
<tr>
<th>#threads/places</th>
<th>OpenMP/C</th>
<th>Hybrid X10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.3E-02</td>
<td>5.3E-02</td>
</tr>
<tr>
<td>2</td>
<td>4.7E-03</td>
<td>7.5E-03</td>
</tr>
<tr>
<td>4</td>
<td>3.7E-03</td>
<td>6.7E-03</td>
</tr>
<tr>
<td>8</td>
<td>4.2E-02</td>
<td>1.6E-02</td>
</tr>
<tr>
<td>16</td>
<td>8.1E-03</td>
<td>1.1E-02</td>
</tr>
<tr>
<td>32</td>
<td>1.6E-02</td>
<td>1.6E-02</td>
</tr>
<tr>
<td>64</td>
<td>3.0E-02</td>
<td>1.5E-02</td>
</tr>
</tbody>
</table>
Performance Results for FFT (w/ memoized sine/cosine twiddle factors)

\[ N = 2^{24} (\text{SQRTN} = 2^{12}) \]
Studying the Java vs. C performance gap (1.1GHz Power4 system, xlc v7.0.0.1 xlc)

Experimental SciMark2 Results

Details on “Compiler flags”
- Force higher optimization level on first execution of methods
- Enable generation of Fused Multiply-Add machine instructions
- Simulate X10 Static Analyzer’s ability to remove most null- and bounds-checks by removing all such checks from selected methods.
- Also remove extra code and register allocation restrictions related to memory management (GC) in selected methods

41 X10: An Object-Oriented Approach to Non-Uniform Cluster Computing
Some Challenges in Optimization of X10 programs

- Improving Single Activity Performance
- Analysis and optimization of explicitly parallel programs
  - Proposed approach: use Parallel Program Graph (PPG) representation
- Analysis and optimization of remote data accesses
  - Proposed approach: perform data access aggregation and elimination using heap-based SSA framework
- Optimized implementation of Atomic Sections
  - Simple cases that can be supported by hardware
  - Analyzable atomic sections
  - General case
- Load-balancing
  - Dynamic, adaptive migration of places across nodes in deployment
- Efficient invocation of components in other languages
  - C, Fortran
- Garbage collection across multiple places
Outline

- Software challenges for multi-core systems
- X10 Programming Model and Language
- X10 Productivity Analysis
- X10 Implementation
- Conclusions
Related Work

- Single Program Multiple Data (SPMD) languages with Partition Global Address Space (PGAS)
  - Unified Parallel C, Co-Array Fortran, Titanium
  - X10 generalizes PGAS to a “threaded-PGAS” model (beyond SPMD)
- Hierarchical fork-join parallelism
  - Cilk (ultra-lightweight threads, work-stealing scheduling, …)
  - X10 generalizes Cilk by adding places, distributions, futures, …
- X10 has similarities with other languages in DARPA HPCS program --- Chapel (Cray) and Fortress (Sun) --- but there are also key differences
  - Chapel allows object migration and data redistribution, which could makes it harder to use for scalable parallelism (compared to X10)
  - Fortress has a major focus on new type system and user-viewable program representations (single-threaded advances that are complementary to X10)
### Relating optimizations for past programming paradigms to X10 optimizations

<table>
<thead>
<tr>
<th>Programming paradigm</th>
<th>Activities</th>
<th>Storage classes</th>
<th>Important optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message-passing e.g., MPI</td>
<td>Single activity per place</td>
<td>Place local</td>
<td>Message aggregation, optimization of barriers &amp; reductions</td>
</tr>
<tr>
<td>Data parallel e.g., HPF</td>
<td>Single global program</td>
<td>Partitioned global</td>
<td>SPMDization, synchronization &amp; communication optimizations</td>
</tr>
<tr>
<td>PGAS e.g., Titanium, UPC</td>
<td>Single activity per place</td>
<td>Partitioned global, place local</td>
<td>Localization, SPMDization, synchronization &amp; communication optimizations</td>
</tr>
<tr>
<td>DSM e.g., TreadMarks</td>
<td>Multiple</td>
<td>Partitioned global, activity local</td>
<td>Data layout optimizations, page locality optimizations</td>
</tr>
<tr>
<td>NUMA</td>
<td>Single activity per place</td>
<td>Partitioned global, activity local</td>
<td>Data distribution, synchronization &amp; communication optimizations</td>
</tr>
<tr>
<td>Co-processor e.g., STI Cell</td>
<td>Single activity per place</td>
<td>Partitioned-global, place-local</td>
<td>Data communication, consistency, &amp; synchronization optimizations</td>
</tr>
<tr>
<td>Futures / active messages</td>
<td>Multiple</td>
<td>Place-local, activity local</td>
<td>Message aggregation, synchronization optimization</td>
</tr>
<tr>
<td>Full X10</td>
<td>Multiple activities in multiple places</td>
<td>Partitioned-global, place-local, activity-local</td>
<td>All of the above</td>
</tr>
</tbody>
</table>
Summary: Advantages of X10 Programming Model

- Any program written with atomic, async, finish, foreach, ateach, and clock parallel constructs *will never deadlock*
  - future-force needs disciplined usage to guarantee absence of deadlock e.g., force should be performed by activity that created future (or an ancestor of that activity)
- Inter-node and intra-node parallelism integrated in a single model
- Remote activity invocation subsumes one-sided data transfer, remote atomic operations, active messages, ...
- Finish subsumes point-to-point and team synchronization
- All remote data accesses are performed as activities ➔ rules for ordering of remote accesses follows simply from concurrency model
- Can be easily mapped to multiple levels of parallel hardware (SIMD, SMT, coprocessors, cache prefetch, SMP, clusters, ...)

X10: An Object-Oriented Approach to Non-Uniform Cluster Computing
Conclusions and Future Work

• X10 programming model provides core concurrency and distribution constructs for new era of mainstream parallel processing

• Two primary opportunities for X10 adoption:
  – DARPA High Programming Language Systems
  – Mainstream language for multi-core

• We’d welcome collaboration on X10
  – Applications to evaluate X10 in different domains
  – X10 subprojects for different hardware platforms
    • Multi-core, Clusters, Co-processors, Grid
  – Participation in productivity experiments
  – Participation in X10 open source project on SourceForge (x10.sf.net)
### Application Frameworks
- Domain-specific prog models & langs:
  - Middleware in support of domain-specific programming models & frameworks e.g., J2EE containers, relational databases, Async Beans

### Middleware
- Tools for improved functional/performance quality of concurrent software

### Programming Tools
- Infrastructure languages that simplify concurrency for programming for C/C++/Java/C# programmers e.g., MS Concur, IBM X10, OpenMP

### General-purpose prog languages
- Dynamic code specialization, speculative parallelism, assist threads, SIMDization, code partitioning

### Dynamic compilers, VMs, Lang Runtime
- Speculative parallelism, assist threads, SIMDization, code partitioning

### Static Compilers
- Libraries that encapsulate concurrency e.g., Java Concurrency Utilities, Transactional Memory, Cell libs, TI OMAP

### System Libraries
- Dynamic scalable management of heterogeneous resources per core (frequency, power)

### OS and Hypervisors
- Application frameworks and libraries e.g., ESSL, graphics libraries, imaging libraries, security libraries

---

**FOCUS OF THIS TALK**

- Need to look holistically at entire Software Stack for Multi-Core Enablement
X10 Eclipse Debugging Toolkit
STREAM

OpenMP / C version

#pragma omp parallel for
for (j=0; j<N; j++) {
    b[j] = scalar*c[j];
}

Hybrid X10 + Serial C version

finish at each(point p : dist.factory.unique()) {
    final region myR = (D | here).region;
    scale(b, scalar, c, myR.rank(0).low(), myR.rank(0).high()+1);
}
STREAM

OpenMP / C version

```c
#pragma omp parallel for
for (j=0; j<N; j++) {
    b[j] = scalar*c[j];
}
```

Hybrid X10 + Serial C version

```c
finish at each(point p : dist.factory.unique()) {
    final region myR = (D | here).region;
    scale(b, scalar, c, myR.rank(0).low(), myR.rank(0).high()+1);
}
```

- `Traversing array region can be error-prone`
- `Implicitly assumes Uniform Memory Access model (no distributed arrays)`
- `Multi-place version designed to run unchanged on an SMP or a cluster`
- `Restrict operator simplifies computation of local region`
- `scale() is a sequential C function`
- `SLOC counts are comparable`
RandomAccess

OpenMP / C version

#define NUPDATE (4 * TableSize)
for (i=0; i<NUPDATE/128; i++) {
#pragma omp parallel for
  for (j=0; j<128; j++) {
    ran[j] = (ran[j] << 1) ^ ((s64Int) ran[j] < 0 ? POLY : 0);
    Table[ran[j] & (TableSize-1)] ^= ran[j];
  }
}

Hybrid X10 + Serial C version

finish ateach(point p : dist.factory.unique()) {
  final region myR = (D | here).region;
  for (int i=0; i<(4 * TableSize)/W; i++) {
    innerLoop(Table,TableSize,ran,myR.rank(0).low(),myR.rank(0).high()+1);
  }
}
OpenMP / C version

```c
#define NUPDATE (4 * TableSize)
for (i=0; i<NUPDATE/128; i++) {
    #pragma omp parallel for
    for (j=0; j<128; j++) {
        ran[j] = (ran[j] << 1) ^ ((s64Int) ran[j] < 0 ? POLY : 0);
        Table[ran[j] & (TableSize-1)] ^= ran[j];
    }
}
```

Hybrid X10 + Serial C version

```c
finish ateach(point p : dist.factory.unique()) {
    final region myR = (D | here).region;
    for (int i=0; i<(4 * TableSize)/W; i++) {
        innerLoop(Table,TableSize,ran,myR.rank(0).low(),myR.rank(0).high()+1);
    }
}
```

- Inner parallel loop is a source of inefficiency in OpenMP version
- Multi-place version designed to run unchanged on an SMP or a cluster
- SLOC counts are comparable
- innerLoop() is a sequential C function
- Restrict operator simplifies computation of local region
FFT: Transpose example

Cilk / C version (Recursive version)
#define SUB(A, i, j) (A)[(i)*SQRTN+(j)]
cilk void transpose(fftw_complex *A, int n)
{
    if (n > 1) {
        int n2 = n/2;
        spawn transpose(A, n2);
        spawn transpose(&SUB(A, n2, n2), n-n2);
        spawn transpose_and_swap(A, 0, n2, n2, n);
    } else { /* 1x1 transpose is a NOP */
    }
}

Hybrid X10 + Serial C version (Non-recursive version)
int nBlocks = SQRTN / bSize;
int p = 0;
finish for (int r = 0; r < nBlocks; ++r) {
    for (int c = r; c < nBlocks; ++c) { // Triangular loop
        final int topLefta_r = (bSize * r);
        final int topLefta_c = (bSize * c);
        final int topLeftb_r = (bSize * c);
        final int topLeftb_c = (bSize * r);
        async (place.factory.place(p++))
            transpose_and_swap(A, topLefta_r, topLefta_c, topLeftb_r, topLeftb_c, bSize);
    }
}