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

www.research.ibm.com/x10

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
  - Igor Peshansky
  - Christoph von Praun
  - Vijay Saraswat
  - Vivek Sarkar
  - Tong Wen

• **X10 Tools**
  - Philippe Charles
  - Julian Dolby
  - Robert Fuhrer
  - Frank Tip
  - Mandana Vaziri

• **Emeritus**
  - Kemal Ebcioglu
  - Christian Grothoff

Recent Publications


Upcoming tutorials

• PACT 2006, OOPSLA 2006, SC 2006 (submitted)
Tokyo Research Laboratory (東京基礎研究所)

Business Service Research
- Business Solution (Marketing Tools)
- Industry Solution (Manufacturing Quality)

Information & Internet
- Technology for Contact Center
- Embedded technology for Automotive Industry
- Compliance

Programming Models & Tools
- XML Technology
- Server Managed User Experience

Systems
- zSeries Differentiation
- Commercial Scale-Out
- Future Commercial Workloads

Science & Technology
- Optical Interconnect
- Embedded System Development Tools
- BlueGene/L Application Tuning

Yamato Laboratory

Director
Hiroshi Maruyama

200 researchers
Established in 1982

IBM

PERCS

X10: An Object-Oriented Approach to Non-Uniform Cluster Computing
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
  - X10 (PERCS), XJ → DALI

- Development Tools
  - X10 tools (PERCS), Eclipse Parallel Tools Platform (PERCS), SAFE (Scalable And Flexible Error detection), Mastery (Legacy Code Transformation), Advanced Refactoring

- Deployment, Execution, Optimization
  - Continuous Program Optimization (PERCS), Jikes RVM, Metronome, Dynamic Optimization, PDS/Mirage

Past projects include: PTRAN automatic parallelization, ASTI optimizer for Fortran 90, IBM HPF compiler & runtime
High Productivity Computing Systems

Goals:
- Provide a new generation of economically viable high productivity computing systems for the national security and industrial user community (2007 – 2010)

Impact:
- **Performance** (efficiency): critical national security applications by a factor of 10X to 40X
- **Productivity** (time-to-solution)
- **Portability** (transparency): insulate research and operational application software from system
- **Robustness** (reliability): apply all known techniques to protect against outside attacks, hardware faults, & programming errors

Applications:
- Intelligence/surveillance, reconnaissance, cryptanalysis, airborne contaminant modeling and biotechnology

Fill the Critical Technology and Capability Gap
- **Today** (late 80’s HPC technology)…..to…..**Future** (Quantum/Bio Computing)
HPCS Projects and their Languages

- Cascade (led by Cray Inc.) -- Chapel
- PERCS (led by IBM) -- X10
- HERO (led by Sun Microsystems, Inc.) -- Fortress
PERCS Programming Model, Tools and Compilers (Productive Easy-to-use Reliable Computer System)

Eclipse platform

Productivity Measurements
Rational PurifyPlus
Refactoring for Concurrency
Rational Team Platform
Performance Explorer
Remote System Explorer
Parallel Tools Platform (PTP)

Text in blue identifies PERCS contributions in Phase II

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

X10 Components
Java components
Fast extern interface
C/C++ components
Fortran components

X10 runtime
Java runtime
C/C++ runtime
Fortran runtime

HPC Toolkit + pSigma + Performance Tuning Automation
Dynamic Compilation + Continuous Program Optimization
Integrated Parallel Runtime: MPI + LAPI + RDMA + OpenMP + threads
shared double a[N], b[N], c[N];
upc_forall (i = 0; i < VectorSize; i++; i) {
    a[i] = b[i] + alpha * c[i];
}

• Important compiler optimization: Identify shared array accesses that have affinity to the accessing thread, and transform them into local accesses

• Performance results on 64K-node BG/L machine:

<table>
<thead>
<tr>
<th>Language</th>
<th>Code Size</th>
<th>TB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline MPI</td>
<td>329</td>
<td>29.9</td>
</tr>
<tr>
<td>Hand-optimized MPI</td>
<td>??</td>
<td>160</td>
</tr>
<tr>
<td>UPC</td>
<td>105</td>
<td>95.7</td>
</tr>
</tbody>
</table>
shared u64Int Table[N];

u64Int ran = starts(NUPDATE/THREADS * MYTHREAD);

upc_forall (i = 0; i < NUPDATE; i++; i) {
    ran = (ran << 1) ^ (((s64Int) ran < 0) ? POLY : 0);
    Table[ran & (TableSize-1)] ^= ran;
}

• Important compiler optimization: Translate update operations into calls to one-sided updates in communication library

• Performance results on 64K-node BG/L machine:

<table>
<thead>
<tr>
<th>Language</th>
<th>Code Size</th>
<th>GUPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline MPI</td>
<td>938</td>
<td>0.28</td>
</tr>
<tr>
<td>Hand-optimized MPI</td>
<td>??</td>
<td>35</td>
</tr>
<tr>
<td>UPC</td>
<td>111</td>
<td>16.7</td>
</tr>
</tbody>
</table>
Outline

1. Motivation and X10 Programming Model
2. X10 Language
3. Productivity Analysis
4. Current Status and Future Work
Future System Trends: a new Era of Mainstream Parallel Processing

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

Multi-core chips  Heterogeneous Parallelism  Cluster Parallelism

Our response:
Use X10 as a new language for parallel hardware that builds on existing tools, compilers, runtimes, virtual machines and libraries
• Beyond current models: MPI, OpenMP, UPC, …
Overview of X10 Concurrency Model

• **Asynchronous activities**
  - Subsumes task parallelism, asynchronous data transfers, messaging
  - `async`, `future`, `foreach`, `ateach` constructs

• **Multithreaded Partitioned Global Address Space**
  - `place` = set of non-migrating activities and shared mutable objects/subarrays
    - Number of places is fixed at application launch time
  - References to shared data can be shared freely across places
  - Locality Rule: Any operation (read/write/invoke) on a shared mutable datum must be performed by an activity *at the same place* as the datum
    - A `BadPlaceException` is thrown when the Locality Rule is violated
  - Immutable data (final, value type instances) can be accessed from any place

• **Coordination of parallel control flow**
  - `finish` --- global termination of sub-activities (generalizes “join”)
  - `clock` --- coordination for dynamic activity group (generalizes “barrier”)

• **Coordination of data accesses**
  - intra-place atomic blocks, `force()` construct on futures

• **Deadlock safety**
  - Any program written with `async`, `foreach`, `ateach`, `finish`, `clock`, `atomic` constructs can *never deadlock*
All concurrency is expressed as asynchronous activities – subsumes threads, structured parallelism, messaging, DMA transfers, etc.

Places encapsulate binding of activities and globally addressable data

Atomic sections enforce mutual exclusion without user-controlled locks

Finish waits for all activities created in scope to terminate

Deadlock safety: any X10 program written with above constructs can never deadlock
Summary of X10 Memory Model

Memory model rules:
1. Accesses to *local mutable data* are performed *synchronously* using intra-place memory model.
2. Accesses to *remote mutable data* are performed by creating *remote activities*.
   - *Weaker ordering* guarantees for remote data accesses than for local data accesses.
3. Accesses to *immutable data* need no consistency management.
Activity Execution within a Place

**Place**

- **Inbound activities**: Ready Activities → Executing Activities → Outbound activities
- **Outbound replies**: Executing Activities → Blocked Activities → Executing Activities
- **Locality Rule**: Activity can only access its stack, place-local mutable data, or global immutable data
- **Atomic sections do not have blocking semantics**: Atomic sections do not have blocking semantics

**Place**

- **Completed Activities**: Ready Activities → Executing Activities
- **Future**: Executing Activities → Blocked Activities
- **Clock**: Executing Activities → Blocked Activities

**Activity Execution**

1. **Inbound activities**
   - Activities enter the place from the outside world.
   - They are ready to be executed.
2. **Executing activities**
   - Activities are currently executing inside the place.
   - They can access their stack, place-local mutable data, or global immutable data.
3. **Blocked activities**
   - Activities are temporarily stopped due to waiting for resources.
4. **Future**
   - Activities that are scheduled to run in the future.

**Local Rule**

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

**Atomic sections do not have blocking semantics**

- Atomic sections do not have blocking semantics, meaning they execute in a single, indivisible step.

---

16 X10: An Object-Oriented Approach to Non-Uniform Cluster Computing
Comparison with other Parallel Programming Models

- 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 (can be adopted as orthogonal extension to X10)
  - Fortress has a major focus on user-viewable program representations and module structures (complementary to X10)
  - Of the three languages, X10 is closest to leveraging skills of mainstream C/C++/Java community
Outline

1. Motivation and X10 Programming Model
2. X10 Language
3. Productivity Analysis
4. Current Status and Future Work
X10 realization in Java

- **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.
  
  \[ \text{region } r = [1:N,1:M]; \]

- **Distribution**
  - Mapping from region to places, e.g.
    
    \[ \text{dist } d = \text{block}(r); \]

- **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
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) // Asynchronous remote get and remote increment, a.x += b.y
    async (b) { final int v = b.y;
                 async (a) atomic a.x += v; }

20 X10: An Object-Oriented Approach to Non-Uniform Cluster Computing
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
// 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]);
   ```
X10 vs. Java™ languages

• X10 is an extension of a serial subset of the Java language
  • Base language = Java 1.4 language
    • Java 5 features (generics, metadata, etc.) will be supported in the future
  • Notable features removed from Java language
    • Concurrency --- threads, synchronized, etc.
    • Java arrays – replaced by X10 arrays
  • Notable features added to Java language
    • Concurrency – places, async, finish, atomic, future, force, foreach, ateach, clocks, distributions
    • X10 arrays --- points, regions, multidimensional distributed arrays, array reductions, array initializers
    • Serial constructs --- nullable, const, extern, value types
• X10-Java interoperability
  • Allow mix of X10 and Java objects, but Java objects must be place-local
JGF montecarlo example: Serial Java version

```java
initTasks() { tasks = new ToTask[nRunsMC]; ...

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());
    }
}
```
public void runThread() {
    results = new Vector(nRunsMC);
    Runnable thobjects[] = new Runnable [JGFMonteCarloBench.nthreads];
    Thread th[] = new Thread [JGFMonteCarloBench.nthreads];
    // Create (nthreads-1) to share work
    for(int i=1;i<JGFMonteCarloBench.nthreads;i++) {
        thobjects[i] = new AppDemoThread(i,nRunsMC);
        th[i] = new Thread(thobjects[i]);
        th[i].start();
    }
    // Parent thread acts as thread 0
    thobjects[0] = new AppDemoThread(0,nRunsMC);
    thobjects[0].run();
    // Wait for child threads
    for(int i=1;i<JGFMonteCarloBench.nthreads;i++) {
        try { th[i].join();} catch (InterruptedException e) {} 
    }
}
class AppDemoThread implements Runnable {
    ... // initialization code
    public void run() {
        PriceStock ps;
        int ilow, iupper, slice;
        slice = (nRunsMC+JGFMonteCarloBench.nthreads-1) / JGFMonteCarloBench.nthreads;
        ilow = id*slice;
        iupper = Math.min((id+1)*slice, nRunsMC);
        for( int iRun=ilow; iRun < iupper; iRun++ ) {
            ps = new PriceStock();
            ps.setInitAllTasks(AppDemo.initAllTasks);
            ps.setTask(AppDemo.tasks[iRun]);
            ps.run();
            AppDemo.results.addElement(ps.getResult());
        }
    } // run()
}
public void runDistributed() throws MPIException {
    int ilow,ihigh;
    if (JGFMonteCarloBench.rank==0)
        results = new Vector(nRunsMC);
    p_nRunsMC = (nRunsMC+JGFMonteCarloBench.nprocess-1) / JGFMonteCarloBench.nprocess;
    p_results[0] = new Vector(p_nRunsMC);
    ilow = JGFMonteCarloBench.rank*p_nRunsMC;
    ihigh = (JGFMonteCarloBench.rank+1)*p_nRunsMC;
    if (JGFMonteCarloBench.rank ==
        JGFMonteCarloBench.nprocess-1)
        ihigh = nRunsMC;
    // Now do the computation.
    PriceStock ps;
    for( int iRun=ilow; iRun < ihigh; iRun++ ) {
        ps = new PriceStock();
        ps.setInitAllTasks(initAllTasks);
        ps.setTask(tasks[iRun]);
        ps.run();
        p_results[0].addElement(ps.getResult());
    }
}
if (JGFMonteCarloBench.rank==0) {
    for(int i=0;i<p_results[0].size();i++)
        results.addElement((ToResult)
            p_results[0].elementAt(i));
    for(int j=1;j<JGFMonteCarloBench.nprocess;j++) {
        p_results[0].removeAllElements();
        MPI.COMM_WORLD.Recv(p_results,0,1,MPI.OBJECT,j,j);
        for(int i=0;i<p_results[0].size();i++)
            results.addElement((ToResult) p_results[0].elementAt(i));
    }
} else
    MPI.COMM_WORLD.Send(p_results,0,1,
        MPI.OBJECT,0,JGFMonteCarloBench.rank);

} // runDistributed()
initTasks() { tasks = new ToTask[dist.block([0:nRunsMC-1])]; ... }

public void runDistributed()
{
    results = new x10Vector(nRunsMC);
    // Now do the computation
    finish ateach ( 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);
    }
}
X10 vs. Java™ languages

• X10 is an extension of a serial subset of the Java language
  – Base language = Java 1.4 language
    • Java 5 features (generics, metadata, etc.) will be supported in the future
  – Notable features removed from Java language
    • Concurrency --- threads, synchronized, etc.
    • Java arrays – replaced by X10 arrays
  – Notable features added to Java language
    • Concurrency – places, async, finish, atomic, future, force, foreach, ateach, clocks, distributions
    • X10 arrays --- points, regions, multidimensional distributed arrays, array reductions, array initializers
    • Serial constructs --- nullable, const, extern, value types

• X10-Java interoperability
  – Allow mix of X10 and Java objects, but Java objects must be place-local
Outline

1. Motivation and X10 Programming Model
2. X10 Language
3. **Productivity Analysis**
4. Current Status and Future Work
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
   - Analysis of results in progress
   - Acknowledgments for productivity study
     - Pittsburgh Supercomputing Center: Nick Nystrom, John Urbanic, Deborah Weisser
     - IBM Research Social Computing Group: Catalina Danis, Christine Halverson, Wendy Kellogg
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>3,081</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>
</tbody>
</table>

total 7,741 11,165 9,626 3,183 52,683 42,467 53,659 180,524
Preliminary Results
(Analysis of results is still in progress)

- 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></th>
<th>MPI</th>
<th>UPC</th>
<th>X10</th>
</tr>
</thead>
<tbody>
<tr>
<td>obtained correct parallel output</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<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

1. Motivation
2. X10 Language and Programming Model
3. Productivity Analysis
4. **Current Status and Future Work**
X10 Implementation Status

Mostly complete w/ open source release planned in 2H2006:

- **Reference X10 implementation on a single SMP**
  - Shared CVS repository with university partners
  - Nightly regression tests (~ 281 unit tests)
  - X10 application set starting to grow
  - Basis for Optimized SMP Implementation

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

Just starting:

- **X10 Libraries for C/C++ users on large-scale machines**
- **Annotations for user-directed code transformations and specializations**
- **Static Analysis and Ahead-Of-Time Optimization**
  - Optimization of BadPlaceException checks and redundant async/finish ops
  - Use of static analysis to enhance X10-specific refactorings
**X10 Reference Implementation**

- **Foo.x10**
  - **x10c**
  - **Foo.java**
    - // #line pseudocomment in Foo.java specifies source line mapping in Foo.x10
  - **Foo.class**
  - **x10c**

**X10 source program** --- must contain a class named Foo with a “public static void main(String[] args) method

**X10 compiler** --- translates Foo.x10 to Foo.java, uses javac to generate Foo.class from Foo.java

**X10 program translated into Java** ---

**X10 Virtual Machine**
(JVM + J2SE libraries + X10 libraries + X10 Multithreaded Runtime)

**X10 Program Output**

**X10 Abstract Performance Metrics**
(event counts, distribution efficiency)

**External DLL’s**

**X10 extern interface**
Using X10 VM’s on Multiple Nodes in an HPC System: an early experiment with LAPI

Single program across many VMs: How a multi-node VM works

To run a program each VM:
- loads the same program and configuration data.
- Establishes LAPI connections
Place 0 runs main

Each VM knows about all other VMs and transfers data on demand
- Asyncs (remote data manipulation) marshal needed data in runtime and send via LAPI
- In many cases, 0-copy, non-serialized transfer may be used
- Low-level API (LAPI) communication library handles data transfers and synchronization issues

Performance: VM + Communication Library Is a Viable Option

Two versions of a parallel transpose used: C and Java, both based on the LAPI library. Run across four machines connected by a Federation switch.

Can co-operating Java processes on different machines exchange data fast enough to be used as a basis for an X10 implementation?

Tile size: 800x800 (640000) 8-byte doubles
System Model: IBM 9119-575 PowerPC, POWER5
Number Of Processors: 8 Clock Speed: 1.92 GHz
Memory Size: 15552 Mb

We think that there is a lot of promise.
Using Places for Intra-SMP Locality

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

→ Thus, places serve only as affinity hints for intra-SMP locality
→ Good motivation for using X10 as a development language multi-core processors
Overview of Single Node SMP
X10 Implementation

X10 source ➔ X10 Grammar ➔ X10 Parser ➔ Analysis passes ➔ Annotated AST ➔ Java code emitter ➔ Java compiler

Common components w/ SAFARI

DOMO Static Analyzer ➔ Analysis passes ➔ Java code emitter ➔ Java compiler

Code Generation Templates

Target Java

X10 classfiles
(Java classfiles with special annotations for X10 analysis info)

Place 0

Place 1

X10 libraries

Java Concurrency Utilities (JCU)

STM library

Place

Ready Activities

Executing Activities

Blocked Activities

Clock

Future

Inbound replies

Outbound activities

Atomic sections do not have blocking semantics

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

JCU thread pool

Inbound activities

Outbound replies

Place Ready Activities Completed Activities Blocked Activities Clock Future

JCU thread pool

Fortran, C/C++ DLL’s

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)

JCU thread pool

Fortran, C/C++ DLL’s

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)
Improving Single Activity Performance

- Optimization of runtime check and floating-point operations
- Specialized optimization of rectangular arrays and regions
- Light-weight allocation and initialization of X10 objects
- Specialized code optimization with dynamic compilation, and optimization of X10 value types.
- Aliasing
  - Successive rows/columns of X10 arrays are never aliased (and always assumed to be aliased in Java)
  - Many optimizations including data-flow based optimizations and scheduling are restricted by imprecise alias information.
- X10 multi-dimensional arrays are not “arrays of arrays”
  - Faster element access via a single base address rather than indirect access
  - Better opportunity for versioning and other bounds check removal
  - Less space wasted by object headers
- General Floating Point improvements
  - E.g., integer to double conversions on PowerPC
- Faster interaction with other languages
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.
X10 Implementation Research Challenges
(Incomplete List)

- Analysis and optimization of explicitly parallel programs
  - Proposed approach: extension of DOMO for analyzing parallel control flow
  - Initial research problem: optimized placement of finish operations

- Garbage collection across multiple places
  - Cannot assume stop-the-world across places
  - Also extend with dynamic change in number of places

- 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

- Efficient invocation of components in other languages
  - C, Fortran
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, ...)

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

- X10 programming model provides core concurrency and distribution constructs for future computer systems in a post-Moore’s Law world

- Future Work (see paper for details)
  - Type system enhancements, Determinacy, Array sublanguage
  - Combining implicit and explicit syntax
  - Static analysis
  - Refactoring tools
  - Scalable optimized implementation
  - . . .

- We’d welcome collaboration on X10
  - Applications to evaluate X10 in different domains
  - X10 subprojects for different hardware platforms
    - Multi-core, Co-processor, Clusters
  - Preparing for an open source release