X10: a High-Productivity Approach to High Performance Programming

Rajkishore Barik
Christopher Donawa
Matteo Frigo
Allan Kielstra
Vivek Sarkar

HPC Challenge Class 2 Award Submission

This work has been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004.
Motivation: Productivity Challenges caused by Future Hardware Trends

Challenge: Develop new language, compiler and tools technologies to support productive portable parallel abstractions for future hardware.
High-Productivity, High-Performance Programming with X10

**X10 Programming Model**

- 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 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

High-Productivity, High-Performance Programming with X10
Current Status: Multi-core SMP Implementation for X10

X10 source

X10 Grammar
DOMO Static Analyzer
Code Generation Templates

X10 Parser
Analysis passes
Java code emitter

Annotated AST
Target Java
Java compiler

Common components w/ SAFARI

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

X10 Front End

JCU thread pool

Place

Ready Activities
Executing Activities
Blocked Activities
Clock
Future

Inbound activities
Outbound activities
Inbound replies
Outbound replies

X10 Runtime

Java Concurrency Utilities (JCU)
STM library

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

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

Extern interface
Fortran, C/C++ DLL's

High-Productivity, High-Performance Programming with X10
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
7 High-Productivity, High-Performance Programming with X10

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 ateach(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

#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);
}

SLOC counts are comparable

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

High-Productivity, High-Performance Programming with X10
Performance Results for STREAM

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

High-Productivity, High-Performance Programming with X10
RandomAccess

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 @each(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);
    }
}
```
RandomAccess

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];
    }
}
```

SLOC counts are comparable

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);
    }
}
```

innerLoop() is a sequential C function

Restrict operator simplifies computation of local region

Inner parallel loop is a source of inefficiency in OpenMP version

Multi-place version designed to run unchanged on an SMP or a cluster
Performance Results for RandomAccess

Array size = 1.8GB
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);
    }
}
Performance Results for FFT
(w/ memoized sine/cosine twiddle factors)

\[ N = 2^{24} \quad (SQRTN = 2^{12}) \]
Summary

• X10 programming model provides core concurrency and distribution constructs for new era of parallel processing
• Results show competitive performance for Hybrid X10+C relative to OpenMP/C and Cilk
• Past studies have shown other productivity benefits of X10
• To find out more, come to the X10 exhibit in the Exotic Technologies area!
X10 context: PERCS Programming Model, Tools and Compilers

(PERCS = Productive Easy-to-use Reliable Computer System)

Eclipse platform

Productivity Measurements
Rational PurifyPlus
Java™ source code
(w/ threads & conc utils)
X10 source code
C/C++ source code
(w/ MPI, OpenMP, UPC)
Fortran source code
(w/ MPI, OpenMP)

Java Development Toolkit
Rational Team Platform
Java Compiler

X10 Development Toolkit
Remote System Explorer
X10 Compiler

C/C++ Development Toolkit
+ MPI & OpenMP extensions

C/C++ Compiler
w/ UPC extensions

Fortran Development Toolkit

Java Components
Java runtime

X10 components
Fast extern interface
X10 runtime

C/C++ components
C/C++ runtime

Fortran components
Fortran runtime

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

Productivity Measurements
Refactoring for Concurrency
Performance Explorer
Parallel Tools Platform (PTP)

Java Development Toolkit
C/C++ Development Toolkit
Fortran Development Toolkit

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

High-Productivity, High-Performance Programming with X10
X10 Eclipse Development Toolkit
X10 Eclipse Debugging Toolkit
20 High-Productivity, High-Performance Programming with X10

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
  - Clocks are a generalization of barriers and MPI communicators

- **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 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.