X10 at Scale

Olivier Tardieu and the X10 Team

http://x10-lang.org

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
X10 Overview
X10: Productivity and Performance at Scale

>8 years of R&D by IBM Research supported by DARPA (HPCS/PERCS)

Programming language

- Bring Java-like productivity to HPC
  - evolution of Java with input from Scala, ZPL, CCP, …
  - imperative OO language, garbage collected, type & memory safe
  - rich data types and type system
- Design for scale
  - multi-core, multi-processor, distributed, heterogeneous systems
  - few simple constructs for concurrency and distribution

Tool chain

- Open source compilers, runtime, IDE
- Debugger (*not open source*)
X10 in a Nutshell
Asynchronous Partitioned Global Address Space (APGAS)

Fine-grained concurrency
- `async S`
- `finish S`

Place-shifting operations
- `at(p) S`
- `at(p) e`

Atomicity within a place
- `when(c) S`
- `atomic S`

Distributed heap
- `GlobalRef[T]`
- `PlaceLocalHandle[T]`
APGAS Idioms

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

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

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

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

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

- **Data exchange**
  \[
  \text{// swap row i local and j remote}
  \text{val} \; h = \text{here};
  \text{val} \; \text{row}_i = \text{rows}()\{(i); \}
  \text{finish} \; \text{at}(p) \; \text{async} \; \{
    \text{val} \; \text{row}_j = \text{rows}()\{(j); \}
    \text{rows}()\{(j) = \text{row}_i; \}
    \text{at}(h) \; \text{async} \; \text{row}()\{(i) = \text{row}_j; \}
  \}
  \]
Scale
What Scale?

X10 has *places* and *asyncs* in each place

We want to

- Handle millions of asyncs (➔ billions)
- Handle tens of thousands of places (➔ millions)

We need to

- **Scale up**
  - shared memory parallelism (today: 32 cores per place)
  - schedule many asyncs with a few hardware threads
- **Scale out**
  - distributed memory parallelism (today: 50K places)
  - provide mechanisms for efficient distribution (data & control)
  - support distributed load balancing
Outline

Scale Out Work

- Experiments
  - DARPA PERCS prototype
  - benchmarks

- Deep Dive
  - distributed termination detection
  - high-performance interconnects
  - memory management
  - distributed load balancing

Scale Up Work

- Overview
  - Cilk-style scheduling for X10
Scale Out: Experiments
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

- **D-Link Optical Interface**: Connects to other Super Nodes
- **Memory DIMM’s (64x)**
- **Hub Module (8x)**
- **PCIe Interconnect**: Connects 4 Nodes to form Super Node
- **L-Link Optical Interface**: Connects to other Super Nodes
- **D-Link Optical Interface**: Connects to other Super Nodes
- **D-Link Optical Fiber**
- **Memory DIMM’s (64x)**
- **Water Connection**
- **P7 QCM (8x)**
- **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)

<table>
<thead>
<tr>
<th></th>
<th>cores</th>
<th>absolute performance at scale</th>
<th>parallel efficiency (weak scaling)</th>
<th>performance relative to best implementation available</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream</td>
<td>55,680</td>
<td>397 TB/s</td>
<td>98%</td>
<td>85% (lack of prefetching)</td>
</tr>
<tr>
<td>FFT</td>
<td>32,768</td>
<td>27 Tflops</td>
<td>93%</td>
<td>40% (no tuning of seq. code)</td>
</tr>
<tr>
<td>Linpack</td>
<td>32,768</td>
<td>589 Tflops</td>
<td>80%</td>
<td>80% (mix of limitations)</td>
</tr>
<tr>
<td>RandomAccess</td>
<td>32,768</td>
<td>843 Gups</td>
<td>100%</td>
<td>76% (network stack overhead)</td>
</tr>
<tr>
<td>KMeans</td>
<td>47,040</td>
<td>depends on parameters</td>
<td>97.8%</td>
<td>66% (vectorization issue)</td>
</tr>
<tr>
<td>SCCA1</td>
<td>47,040</td>
<td>depends on parameters</td>
<td>98.5%</td>
<td>100%</td>
</tr>
<tr>
<td>SCCA2</td>
<td>47,040</td>
<td>245 B edges/s</td>
<td>&gt; 75%</td>
<td>no comparison data</td>
</tr>
<tr>
<td>UTS (geometric)</td>
<td>55,680</td>
<td>596 B nodes/s</td>
<td>98%</td>
<td>reference code does not scale 4x to 16x faster than UPC code</td>
</tr>
</tbody>
</table>
HPCC Class 2 Competition: Best Performance Award

G-FFT

G-HPL

EP Stream (Triad)

G-RandomAccess

UTS
Scale Out: Deep Dive
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 Communication

*High-Performance Interconnects*

- **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(here.id);
  columnTeam.addReduce[columnRole, localMax, Team.MAX);
  ```

- bright future (MPI-3 and much more...)

  ```
  ```

  - good fit for APGAS
  - poor fit for APGAS today
  - good fit for APGAS
Row Swap in Linpack

// swap srcRow here with dstRow at place dst
def rowSwap(srcRow:Int, dstRow:Int, dst:Place) {
    val srcBuffer = buffers(); // per-place buffer (PlaceLocalHandle)
    val srcBufferRef = new RemoteRail(srcBuffer);
    val size = matrix().getRow(srcRow, srcBuffer);
    @Pragma(Pragma.FINISH_ASYNC_AND_BACK) finish {
        at (dst) async {
            @Pragma(Pragma.FINISH_ASYNC) finish {
                val dstBuffer = buffers();
                Array.asyncCopy[Double](srcBufferRef, 0, dstBuffer, 0, size);
            }
            matrix().swapRow(dstRow, dstBuffer);
            Array.asyncCopy[Double](dstBuffer, 0, srcBufferRef, 0, size);
        }
        matrix().setRow(srcRow, srcBuffer);
    }
}
Scalable Memory Management

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

- 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

not an issue in practice

issue is contained
Time-Out

Scale Out Work
- Experiments
  - DARPA PERCS prototype
  - benchmarks
- Deep Dive
  - distributed termination detection
  - high-performance interconnects
  - memory management
  - distributed load balancing

Scale Up Work
- Overview
  - Cilk-style scheduling for X10

Implement traditional codes
- Statically scheduled and distributed
- With less pain
  - type safe & memory safe
  - PGAS
  - higher-level constructs
    - for concurrency and distribution

Implement new kind of codes
Scalable Global Load Balancing

Unbalanced Tree Search

- Problem
  - count nodes in randomly generated tree
  - separable cryptographic random number generator
    \[
    \text{childCount} = f(\text{nodeId})
    \]
    \[
    \text{childId} = \text{SHA1}(\text{nodeId}, \text{childIndex})
    \]
  - 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

good model for state space exploration problems
Scalable Global Load Balancing

Unbalanced Tree Search

- Lifeline-based global work stealing [PPoPP’11]
  - $n$ random victims then $p$ lifelines (hypercube)
    - fixed graph with low degree and low diameter
  - synchronous (steal) then asynchronous (deal)
- Root finish accounts for
  - startup asyncs + lifeline asyncs
  - not random steal attempts
- Compact work queue (for shallow trees)
  - represent intervals of siblings
  - thief steals half of each work item
- Sparse communication graph
  - bounded list of potential random victims
  - finish trades contention for latency

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

def steal() {
    val h = here.id;
    for (i:Int in 0..w) {
        if (!empty()) break;
        finish at (Place(victims(rnd.nextInt(m)))) async request(h, false);
    }
    for (lifeline:Int in lifelines) {
        if (!empty()) break;
        if (!lifelinesActivated(lifeline)) {
            lifelinesActivated(lifeline) = true;
            finish at (Place(lifeline)) async request(h, true);
        }
    }
}
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();
    }
  }
}
From Theory to Practice

From 256 cores in January 2011 to 7,936 in March 2012 to 47,040 in July 2012
Delivered in August 2012

good abstractions
Scale Up
Local Load Balancing

- Problem: many more asyncs than execution units (hardware threads)

- Solution: cooperative work-stealing scheduling
  - pool of worker threads
  - per-worker deque (double-ended queue) of pending jobs
  - idle worker pops job from own deque or steals from other worker if empty

- Question 1: what is a job?
  - obvious candidate: async body (fork/join-style scheduler)
  - alternative: continuation (cilk-style scheduler)

- Question 2: what if a job has to wait (IO, RPC…)?
  - if possible: execute prereq job while waiting
  - if not either: wait (but spawn new thread to maintain core utilization)
  - or: save execution state and reuse thread
Fibonacci

static def fib(n: Int) {
    if (n<=1) return n;
    val x: Int;
    val y: Int;
    finish {
        async x = fib(n-2);
        y = fib(n-1);
    }
    return x+y;
}
Cilk-Style Scheduling for X10

- X10-source-to-X10-source program transformation
  - exposes the “compiler stack”
    - transforms methods into frame classes
    - transforms local variables into frame fields
    - rewrite method invocations to save the parent frame pointer and pc
  - rewrite methods to permit reentry (resume execution at specified instruction)
  - rewrite X10 concurrency constructs into invocations of the runtime scheduler

- X10 runtime scheduler in X10
  - can walk the stack, save the stack, restore the stack
  - can resume the execution at saved program point with restored stack

- Hacks
  - permit uninitialized fields
  - speculative stack allocation
Micro Benchmarks

Fib speedup
Integrate speedup
QuickSort speedup

execution time (s) with 16 threads
sequential overhead
X10 vs. Cilk++: PBBS Benchmarks

http://www.cs.cmu.edu/~guyb/pbbs/

- Compared running time (s) with 16 cores
X10 vs. Cilk++: PBBS Benchmarks

http://www.cs.cmu.edu/~guyb/pbbs/

- Compared sequential overhead
Wrap-Up
Conclusions

- X10 can scale to Petaflop systems
- X10 can ease the development of scalable codes
  - for traditional statically scheduled and distributed codes
  - for novel classes of codes (APGAS)

Future Work

- Tightly integrate the scale-out and scale-up work
- Continue and generalize work on distributed termination
- Systematize distributed load balancing
- Develop APGAS beyond X10
  - X10-runtime-as-a-library project
References

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

- “A Brief Introduction to X10 (for the HPC Programmer)”

- X10 2012 HPC challenge submission
  http://hpcchallenge.org

- Unbalanced Tree Search in X10
  http://dl.acm.org/citation.cfm?id=1941582

- Cilk-style scheduling for X10
  http://dl.acm.org/citation.cfm?id=2145850