X10 and APGAS at Petascale

Olivier Tardieu¹, Benjamin Herta¹, David Cunningham², David Grove¹, Prabhanjan Kambadur¹, Vijay Saraswat¹, Avraham Shinnar¹, Mikio Takeuchi³, Mandana Vaziri¹

¹IBM T.J. Watson Research Center
²Google Inc.
³IBM Research – Tokyo

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

- X10 tackles the challenge of programming at **scale**
  - HPC, cluster, cloud
  - scale out: run across many distributed nodes ➔ this talk & PPAA talk
  - scale up: exploit multi-core and accelerators ➔ CGO tutorial
  - resilience and elasticity ➔ next talk

- X10 is
  - a programming language
    - imperative object-oriented strongly-typed garbage-collected (like Java)
    - concurrent and distributed: Asynchronous Partitioned Global Address Space model
  - an open-source tool chain developed at IBM Research ➔ X10 2.4.2 just released
  - a growing community
    - X10 workshop at PLDI’14 ➔ CFP at http://x10-lang.org

- Double goal: **productivity** and **performance**
Outline

- X10
  - programming model: Asynchronous Partitioned Global Address Space

- Optimizations for scale out
  - distributed termination detection
  - high-performance networks
  - memory management

- Performance results
  - Power 775 architecture
  - benchmarks

- Global load balancing
  - Unbalanced Tree Search at scale
X10 Overview
APGAS Places and Tasks

Task parallelism
- **async** $S$
- **finish** $S$

Place-shifting operations
- **at**($p$) $S$
- **at**($p$) $e$

Concurrency control
- **when**(c) $S$
- **atomic** $S$

Distributed heap
- **GlobalRef**[$T$]
- **PlaceLocalHandle**[$T$]
APGAS Idioms

- Remote procedure call
  \[ v = \text{at}(p) \ f(\text{arg1}, \text{arg2}); \]

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

- SPMD
  \[ \text{finish} \ \text{for}(p \ \text{in} \ \text{Place.\places()}) \ { \]
  \[ \text{at}(p) \ \text{async} \ { \]
  \[ \text{for}(i \ \text{in} \ 1..n) \ { \]
  \[ \text{async} \ \text{doWork}(i); \]
  \[ } \]
  \[ } \]

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

- Divide-and-conquer parallelism
  \[ \text{def} \ \text{fib}(n:\text{Long}):\text{Long} \ { \]
  \[ \text{if}(n < 2) \ \text{return} \ n; \]
  \[ \text{val} \ x:\text{Long}; \]
  \[ \text{val} \ y:\text{Long}; \]
  \[ \text{finish} \ { \]
  \[ \text{async} \ x = \text{fib}(n-1); \]
  \[ y = \text{fib}(n-2); \]
  \[ } \]
  \[ \text{return} \ x + y; \]
  \[ } \]
public class BlockDistRail[T] {
    protected val sz:Long;  // block size
    protected val raw:PlaceLocalHandle[Rail[T]];

    public def this(sz:Long, places:Long){T haszero} {
        this.sz = sz;
        raw = PlaceLocalHandle.make[Rail[T]](PlaceGroup.make(places), ()=>new Rail[T](sz));
    }
    public operator this(i:Long) = (v:T) { at(Place(i/sz)) raw()(i%sz) = v; }
    public operator this(i:Long) = at(Place(i/sz)) raw()(i%sz);

    public static def main(Rail[String]) {
        val rail = new BlockDistRail[Long](5, 4);
        rail(7) = 8;
        Console.OUT.println(rail(7));
    }
}
Optimizations for Scale Out
Distributed Termination Detection

- Local finish is easy
  - synchronized counter: increment when task is spawned, decrement when task ends

- Distributed finish is non-trivial
  - network can reorder increment and decrement messages

- X10 algorithm: disambiguation in space
  - one row of $n$ counters per place with $n$ places
  - when place $p$ spawns task at place $q$ increment counter $q$ at place $p$
  - when task terminates at place $p$ decrement counter $p$ at place $p$
  - finish triggered when sum of each column is zero

- Charm++ algorithm: disambiguation in time
  - successive non-overlapping waves of termination detections
Optimized Distributed Termination Detection

- Source optimizations
  - aggregate messages at source
  - compress

- Software routing
  - aggregate messages at intermediate nodes

- Pattern-based specialization
  - “put”: a finish governing a single task ➔ wait for one ack
  - “get”: a finish governing round trip ➔ wait for return task
  - local finish: a finish with no remote tasks ➔ single counter
  - SPMD finish: a finish with no nested remote task ➔ single counter
  - irregular/dense finish: a finish with lots of links ➔ software routing

- Runtime optimizations + static analysis + pragmas ➔ scalable finish
High-Performance Networks

- **RDMAs**
  - efficient remote memory operations
  - asynchronous semantics
    - just another task
  ➔ good fit for APGAS

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

- **Collectives**
  - multi-point coordination and communication
  - networks/APIs biased towards SPMD today
  ➔ poor fit for APGAS today

  ```java
  Team.WORLD.barrier(here.id);
  columnTeam.addReduce(columnRole, localMax, Team.MAX);
  ```

- **future: MPI-3 and beyond**
  - one-sided collectives, endpoints, etc.
  ➔ good fit for APGAS
Memory Management

- Garbage collector
  - problem: distributed heap
    - distributed garbage collection is impractical
  - solution: segregate local/remote refs
    - only local refs are automatically collected  ➔ issue is contained

- Congruent memory allocator
  - problem: low-level requirements
    - large pages required to minimize TLB misses
    - registered pages required for RDMAs
    - congruent addresses required for RDMAs at scale
  - solution: dedicated memory allocator
    - congruent registered pages
    - large pages if available
    - only used for performance-critical arrays
    - only impacts allocation & deallocation  ➔ issue is contained
Performance Results
DARPA HPCS/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
DARPA HPCS/PERCS Benchmarks

- HPC Challenge benchmarks
  - Linpack
  - Stream Triad
  - Random Access
  - Fast Fourier Transform
  - TOP500 (flops)
  - local memory bandwidth
  - distributed memory bandwidth
  - mix

- Machine learning kernels
  - KMeans
  - SSCA1
  - SSCA2
  - UTS
  - graph clustering
  - pattern matching
  - irregular graph traversal
  - unbalanced tree traversal

- Implemented in X10 as pure scale out tests
  - one core = one place = one main task
  - native libraries for sequential math kernels: ESSL, FFTE, SHA1
### Performance at Scale (Weak Scaling)

<table>
<thead>
<tr>
<th></th>
<th>number of cores at scale</th>
<th>absolute performance at scale</th>
<th>relative efficiency compared to single host (weak scaling)</th>
<th>performance at scale 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>87%</td>
</tr>
<tr>
<td>FFT</td>
<td>32,768</td>
<td>28.7 Tflop/s</td>
<td>100%</td>
<td>41% (no tuning)</td>
</tr>
<tr>
<td>Linpack</td>
<td>32,768</td>
<td>589 Tflop/s</td>
<td>87%</td>
<td>85%</td>
</tr>
<tr>
<td>RandomAccess</td>
<td>32,768</td>
<td>843 Gup/s</td>
<td>100%</td>
<td>81%</td>
</tr>
<tr>
<td>KMeans</td>
<td>47,040</td>
<td></td>
<td>98%</td>
<td>?</td>
</tr>
<tr>
<td>SSCA1</td>
<td>47,040</td>
<td></td>
<td>98%</td>
<td>?</td>
</tr>
<tr>
<td>SSCA2</td>
<td>47,040</td>
<td>245 B edges/s</td>
<td>see paper for details</td>
<td>?</td>
</tr>
<tr>
<td>UTS (geometric)</td>
<td>55,680</td>
<td>596 B nodes/s</td>
<td>98%</td>
<td>prior impl. do not scale</td>
</tr>
</tbody>
</table>
HPCC Class 2 Competition 2012: Best Performance Award

- **G-FFT**
  - Gflops/Place: 26958
  - Gflops: 0.88

- **G-HPL**
  - Gflops/Place: 589231
  - Gflops: 22.4

- **EP Stream (Triad)**
  - GB/s/Place: 396614
  - GB/s: 7.23

- **G-RandomAccess**
  - Gups/Place: 844
  - Gups: 0.82

- **UTS**
  - Million nodes/s/Place: 596451
  - Million nodes/s: 10.93
Global Load Balancing
Unbalanced Tree Search at Scale

- **Problem**
  - count nodes in randomly generated tree → unbalanced & unpredictable
  - separable random number generator → no locality constraint

- Lifeline-based global work stealing [PPoPP’11]
  - $n$ random victims then $p$ lifelines (hypercube)
  - steal (synchronous) then deal (asynchronous)

- **Novel optimizations**
  - use of nested finish scopes → scalable finish
    - use of “dense” finish pattern for root finish
    - use of “get” finish pattern for random steal attempt
  - pseudo random steals → software routing
  - compact work queue encoding (for shallow trees) → less state, smaller messages
    - lazy expansion of intervals of nodes (siblings)
Conclusions

- **Performance**
  - X10 and APGAS can scale to Petaflop systems

- **Productivity**
  - X10 and APGAS can implement legacy algorithms
    - such as statically scheduled and distributed codes
  - X10 and APGAS can ease the development of novel scalable codes
    - including irregular and unbalanced workloads

- APGAS constructs deliver productivity and performance gains at scale

- Follow-up work presented PPAA’14
  - APGAS global load balancing library derived from UTS
  - application to SSCA2