X10 and APGAS at Petascale

Olivier Tardieu[1], Benjamin Herta[1], David Cunningham[2] *, David Grove[1], Prabhanjan Kambadur[1], Vijay Saraswat[1], Avraham Shinnar[1], Mikio Takeuchi[3], Mandana Vaziri[1]

[1]IBM T. J. Watson Research Center
[2]Google Inc.
[3]IBM Research - Tokyo

{tardieu,bherta,groved,pkambadu,vsaraswa,shinnar,mvaziri}@us.ibm.com, dcunnin@google.com, mtake@jp.ibm.com

Abstract

X10 is a high-performance, high-productivity programming language aimed at large-scale distributed and shared-memory parallel applications. It is based on the Asynchronous Partitioned Global Address Space (APGAS) programming model, supporting the same fine-grained concurrency mechanisms within and across shared-memory nodes.

We demonstrate that X10 delivers solid performance at petascale by running (weak scaling) eight application kernels on an IBM Power 775 supercomputer utilizing up to 55,680 Power7 cores (for 1.7 Pflop/s of theoretical peak performance). We detail our advances in distributed termination detection, distributed load balancing, and use of high-performance interconnects that enable X10 to scale out to tens of thousands of cores.

For the four HPC Class 2 Challenge benchmarks, X10 achieves 41% to 87% of the system's potential at scale (as measured by IBM's HPCC Class 1 optimized runs). We also implement K-Means, Smith-Waterman, Betweenness Centrality, and Unbalanced Tree Search (UTS) for geometric trees. Our UTS implementation is the first to scale to petaflop systems.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—distributed programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—concurrent programming structures, control structures

Keywords X10; APGAS; scalability; performance

* Work done while employed at IBM T. J. Watson Research Center.

PPoPP '14, February 15–19, 2014, Orlando, Florida, USA.

http://dx.doi.org/10.1145/10.1145/2555243.2555245

1. Overview

X10 is a high-performance, high-productivity programming language developed at IBM.[1] It is a class-based, strongly-typed, garbage-collected, object-oriented language [32, 33]. To support concurrency and distribution, X10 uses the Asynchronous Partitioned Global Address Space programming model (APGAS [31]). This model introduces two key concepts – places and asynchronous tasks – and a few mechanisms for coordination. With these, APGAS can express both regular and irregular parallelism, message-passing-style and active-message-style computations, fork-join and bulk-synchronous parallelism. In contrast to hybrid models like MPI+OpenMP, the same constructs underpin both intra- and inter-place concurrency.

Both its modern, type-safe sequential core and simple programming model for concurrency and distribution contribute to making X10 a high-productivity language in the HPC and Big Data spaces. User productivity is further enhanced by providing tools such as an Eclipse-based IDE (X10DT) and a source-level debugger.

In this paper, we focus on enabling X10 applications to run at very large scale. We demonstrate that X10 and APGAS deliver performance at petascale for regular and irregular kernels. We present experimental results for eight kernels.[2] We implement the four HPC Class 2 Challenge benchmarks: *HPL*, *FFT*, *RandomAccess*, and *Stream Triad* [17], as well as *Smith-Waterman* [37], *Betweenness Centrality* (BC) [5], *K-Means* [22], and *Unbalanced Tree Search* (UTS) [25]. These programs are compiled using X10's native backend,[3] and run on a large IBM Power 775 system with a theoretical peak performance of 1.7 Pflop/s.

[1] X10 was developed as part of the IBM "Productive, Easy-to-use, Reliable Computing System" project (PERCS [41]), supported by the DARPA High Productivity Computer Systems initiative (HPCS [11]).

[2] The X10 tool chain and the benchmark codes are publicly available at `http://x10-lang.org`.

[3] X10 is implemented with two backends. On the *managed* backend, X10 compiles into Java and runs on (a cluster of) JVMs. On the *native* backend, X10 compiles into C++ and generates a native binary for execution on scale-out systems.

For the four HPC Challenge benchmarks, X10 today achieves 41% to 87% of the system's potential at scale as reported by IBM's *optimized runs* entry to the HPC Class 1 Challenge in November 2012 [16]. Our K-Means and Smith-Waterman implementations scale linearly with the number of compute nodes (weak scaling). Our UTS implementation for geometric trees also scales linearly. To the best of our knowledge, it is the first implementation of UTS to scale to petaflop systems. Our BC implementation, which is being actively developed, is able to process 245 Billion edges per second using 47,040 cores.

All applications are written to run with minimal intra-place concurrency. We have separately done work on schedulers for intra-place concurrency [13, 40], but the results reported here do not reflect the integration of these schedulers with the scale-out stack. We leave this for future work.

All applications, except UTS, are implemented using a classic SPMD approach. For these applications, data is statically partitioned and computations are statically scheduled. For UTS, dynamic distributed load balancing is indispensable. We had to significantly revise and extend the lifeline-based load balancing algorithm from Saraswat et al. [35] to make it scale to tens of thousands of Power7 cores. While we recently added dynamic load balancing to our BC implementation, the results we report here predate this change.

Contributions. The contributions of this paper are:

- We demonstrate that the APGAS programming model delivers performance at petascale for both regular and irregular kernels.
- We show experimental results for eight kernel benchmarks implemented in X10 and running on a petaflop Power 775 system. We provide detailed performance analyses and, for the four HPC Challenge benchmarks, we compare X10 performance to the best implementations available.
- We describe our solutions to the distributed termination detection problem at scale – the implementation of X10's *finish* construct – and the effective use of high-performance interconnects in X10.
- We present a novel distributed load balancing algorithm for UTS derived from [35]. To the best of our knowledge, our implementation is the first that can effectively load balance UTS on geometric trees at petascale.

Outline. The next two sections review the core concepts of the X10 programming language (Section 2) and describe the key innovations necessary to allow them to perform at scale (Section 3). After describing the hardware and software configuration in Section 4, we discuss the implementation and performance results for each kernel in turn: we review the HPC Challenge benchmarks in Section 5, UTS in Section 6, and the remaining codes in Section 7. We discuss related work in Section 8 and conclude in Section 9.

2. The X10 Language

Like Java, X10 is a strongly-typed, garbage-collected, class-based, object-oriented programming language with single-class multiple-interface inheritance. To support concurrency and distribution, X10 introduces a few key concepts. We briefly review the core Asynchronous Partitioned Global Address Space programming model (APGAS [31]) that is at the heart of the X10 programming model.

2.1 APGAS Concepts

A *place* is a collection of data and worker threads operating on the data, typically realized as an operating system process. A single X10 computation typically runs over a large collection of places. The notion of places is reified in the language: if `S` is a statement, then `at(p) S` is a statement that shifts to place `p` to execute `S`. Similarly, `at(p) e` evaluates the expression `e` at place `p`. The expression `here` evaluates to the current place. The expression `GlobalRef(someObject)` computes a global reference to `someObject` that can be passed freely from place to place but only dereferenced at the `home` place of the reference, that is, the place of `someObject`.

Asynchrony is fundamental to the language: if `S` is a statement then `async S` is a statement which runs `S` as a separate, concurrent *activity*. Dually, the `finish S` statement executes `S` and waits for all activities transitively spawned during the execution of `S` to terminate before continuing.

Additional concurrency control is provided by the statement `when(c) S` which executes `S` in a single uninterrupted step when the condition `c` is true. An optimized unconditional form of `when`, `atomic S`, is also provided.

Other X10 features such as asynchronous memory transfers and dynamic barriers (*clocks*) can be viewed as particular patterns of use of these constructs.

An X10 program consists of at least one class definition with a `main` method. The number n of places available to a particular execution (0 to $n-1$) and the mapping from places to nodes is specified by the user at launch time using MPI-like controls. The execution starts with the execution of the `main` method at `Place(0)`. Other places are initially idle.

2.2 APGAS Idioms

The power of X10's core APGAS constructs lies in that, for the most part, they can be nested freely. Combinations of these constructs provide for MPI-like message passing, SPMD computation, active-message-style computation, bulk synchronous parallel computation, overlap between computation and communication, fork-join recursive parallel decomposition, etc. Formal semantics for these constructs have been developed in [8, 20, 33].

Remote evaluation is simply:

```
val v = at(p) evalThere(arg1, arg2); // blocking
```

We can combine `at` and `async` to obtain active messages:

```
at(p) async runThere(arg1, arg2); // non-blocking
```

We can combine `finish` and `async` to compute Fibonacci numbers with recursive parallel decomposition:

```
def fib(n:Int):Int {
  if (n < 2) return n;
  val f1:Int; val f2:Int;
  finish {
    // f1 and f2 are computed in parallel
    async f1 = fib(n-1);
    f2 = fib(n-2);
  }
  return f1+f2;
}
```

In the next example, one activity is spawned in each place to run some startup code. The `finish` construct works across places to ensures the initialization completes in each place before the main body runs:

```
class Foo {
  public static def main(args:Rail[String]) {
    finish for(p in Place.places()) {
      at(p) async {
        ... // startup code
    } }
    ... // main body
} }
```

Parallel, possibly distributed tasks can be synchronized with clocks. In this example, the clock ensures that loop iterations are synchronized across places:

```
clocked finish for (p in Place.places()) {
  at(p) clocked async for (val i in 0..9) {
    ... // loop body
    Clock.advanceAll(); // global barrier
} }
```

Below, we use a `GlobalRef` and `atomic` updates to compute the average system load across all places:

```
val acc = new Cell[Double](0.0);
val ref = GlobalRef[Cell[Double]](acc);
finish for(p in Place.places()) at(p) async {
  val load = MyUtils.systemLoad(); // at place p
  at(ref.home) async atomic ref() += load;
}
val averageLoad = acc()/Place.MAX_PLACES;
```

X10's compiler and runtime systems understand asynchrony and places and support them. For instance, the type checker tracks occurrences of `GlobalRef`s to ensure these are dereferenced in the proper places. In the last example, it verifies that `ref` is only accessed at place `ref.home`.

The compiler also analyzes the bodies of `at` statements and expressions to identify inter-place data dependencies and instructs the X10 runtime to serialize data accordingly.[4] In this example, the value of `load` is serialized from place `p` to place `ref.home` as part of the execution of `at(ref.home)`.

[4] Data reachable from the body of an `at` is implicitly copied from the source to the destination place of the `at`. `GlobalRef`s can be used to obtain remote references instead.

For bulk copies, such as array copies, X10's standard libraries offer dedicated APIs such as the `Array.asyncCopy` method, which minimize local memory transfers. An array `asyncCopy` is treated exactly as if it were an `async`. Its termination is simply tracked by the enclosing finish, making it easy to overlap communication and computation:

```
finish {
  // srcArray is local, dstArray is remote
  Array.asyncCopy(srcArray, 0, dstArray, 0, size);
  computeLocally(); // while sending the data
}
```

A great deal more information on X10 can be found online at `http://x10-lang.org` including the language specification [32], programmer's guide [34], and a collection of tutorials and sample programs.

2.3 Productivity

X10's productivity results from the combination of object-orientedness, strong typing, memory safety, a simple programming model for concurrency and distribution (APGAS), and tooling (Eclipse-based IDE and debugger).

The same `finish` construct is employed to track the termination of a large number of asynchronous tasks (local or remote) as well as the delivery of a single message or anything in-between. The same asynchronous tasks are used for distributing computations or data or both simultaneously across nodes. In this paper, we demonstrate that such an economy of means does not preclude scalable performance while providing a significant productivity boost. An in-depth productivity analysis however is beyond the scope of this paper. See Section 8 for references on this topic.

3. X10 at Scale

In this section, we discuss the key innovations and extensions that are needed to successfully scale X10 and the APGAS programming model to very large systems.

3.1 Scalable Finish

The X10 language places no restrictions on the ability of the programmer to combine and nest `at` and `async` statements within a `finish`. Implementing X10's `finish` construct therefore requires a distributed termination protocol that can handle arbitrary patterns of distributed task creation and termination.

Because networks can reorder control messages, fully general distributed termination detection algorithms become prohibitively expensive with scale. In particular, the default `finish` implementation in X10 uses $O(n^2)$ space where n is the number of places involved. It may also flood the network interface of the place of the activity waiting on the `finish` construct.

Collective communications such as barriers in large distributed systems are optimized in hardware and/or software. We need to do the same for `finish`.

Finish optimizations. We augment the X10 runtime to dynamically optimize `finish` by optimistically assuming that it is local (within a single place) and then dynamically switching to a more expensive distributed algorithm the first time an activity governed by the `finish` executes an `at`. Furthermore, the runtime automatically coalesces and compresses the control messages used by the termination detection algorithm.

In addition to these general dynamic optimizations, the runtime provides implementations of distributed `finish` that are specialized to common patterns of distributed concurrency for which there exist more efficient implementations. Currently we recognize five such patterns:

`FINISH_ASYNC` a finish governing a single activity, possibly remote. E.g.,

```
finish at(p) async S;
finish { async S1; S2 } // with S2 sequential
```

`FINISH_HERE` a finish governing a round trip. E.g.,

```
h=here; finish at(p) async {S1; at(h) async S2;}
```

`FINISH_LOCAL` a finish governing local activities. For instance, if `S` does not spawn remote activities,

```
finish for(i in 1..n) async S;
```

`FINISH_SPMD` a finish governing the execution of remote activities that do not spawn subactivities outside of a `finish`. E.g.,

```
finish for(p in places) at(p) async finish S;
```

`FINISH_DENSE` a finish governing activities with dense or irregular communication graphs. E.g.,

```
finish for(p in places) at(p) async {
  finish for(q in places) at(q) async S;
}
```

In this example, there is direct communication between any two places in `places`.

Each one of our benchmark programs uses a subset of these specialized distributed termination detection algorithms. The optimized implementations start to make a difference with hundreds of X10 places and become critical with thousands of places or more.

SPMD codes typically exploit three of these implementations: `FINISH_SPMD` for the "root" finish governing the parallel execution of the "main" activity in each place, `FINISH_ASYNC` for "puts" and `FINISH_HERE` for "gets". Codes exploiting intra-place concurrency can benefit from `FINISH_LOCAL` (even if the default algorithm already does a pretty good job in this case). Codes with lots of peer-to-peer communications use `FINISH_DENSE` in place of `FINISH_HERE`. This is the case for instance of the distributed load balancer in the Unbalanced Tree Search code (see below), as it permits an idle place to directly request work from randomly selected places in a fully decentralized manner.

For the first four patterns, the termination detection algorithms are actual specializations of the default algorithm. For instance, for `FINISH_SPMD`, the runtime knows it needs to wait for exactly n termination messages if n remote activities were spawned. In contrast to the default case, the order, source place, or content of each message is irrelevant.

Dense, irregular communication graphs are always a challenge. Network stacks of supercomputers tend to be optimized for traditional, regular workloads and are of course very concerned with latency. For instance the Power 775 network stack favors communication graphs with low outdegree. Detecting the termination of a large number of irregular activities across a large number of places is therefore not something these stacks are designed to do well out of the box. First, there is no regularity to exploit. Second, optimizing for latency is just wrong as the latency of the *last* control message is the only one that matters. But the network stack has no way of knowing this.

This is where the `FINISH_DENSE` implementation comes to the rescue. It uses software routing techniques to shape the traffic of control messages into something more natural, more idiomatic to the network stack. For now, it simply compensates for the fact that we run multiple X10 places per compute node by routing all communications through one master place in each node. A finish control message going from place p to q is routed via places $p - p\%b$ then $q - q\%b$ then q where b is the number of places per node (up to 32).

In the future, it will be interesting to see if such a latency-trading traffic-shaping approach is beneficial to other architectures as well, and whether more sophisticated routing strategies can further improve performance.

Implementation selection. We have prototyped a fully automatic compiler analysis that is capable of detecting many of the situations where these patterns are applicable. For instance, it correctly classifies the various occurrences of `finish` in our HPL code into instances of `FINISH_SPMD`, `FINISH_ASYNC`, and `FINISH_HERE`.

Unfortunately, we have not been able to turn this prototype into a finished product yet. Therefore, in our current system, opportunities to apply these specialized `finish` implementations are still guided by programmer supplied annotations – pragmas – such as:

```
@Pragma(Pragma.FINISH_ASYNC) finish at(p) async S;
```

As future work, we intend to further "harden" this analysis to the point where it can be robustly applied as part of the X10 compiler's standard optimization package. Ultimately, we do not want the end-user to worry about `finish` implementation selection at all.

We also intend to make the specialization mechanism extensible and let expert users develop efficient implementations tailored to particular concurrency patterns and/or specific network topologies.

3.2 Scalable Broadcast

Iterating sequentially over many places to send identical or similar messages (as we naively do in the examples of Section 2) can waste valuable time and flood the network. We developed a `PlaceGroup` library for efficiently managing large groups of places. In particular, we support efficient broadcast over place groups using spawning trees in order to parallelize and distribute the task creation overhead. It also efficiently handles the detection of the broadcast completion by means of nested `FINISH_SPMD` blocks.

3.3 High-Performance Interconnects

As expected from supercomputers, the Power 775 *Torrent* interconnect supports additional communication mechanisms beyond basic point-to-point "fifo" primitives.

To make it possible to adapt to a wide range of interconnects, the X10 runtime has a layered structure. At the lower level, the X10 Runtime Transport (X10RT) API provides a common interface to transports such as IBM's PAMI transport, MPI, and TCP/IP sockets.

We add support for RDMAs and collectives to X10RT (see below). An implementation of X10RT however is only required to provide basic point-to-point primitives. We provide an emulation layer to handle the more advanced APIs when not natively supported. We surface them in the X10's standard libraries as follows.

Collectives. X10 teams – the `x10.util.Team` class – offer capabilities similar to HPC collectives, such as Barrier, All-Reduce, Broadcast, All-To-All, etc. Some networks support these multi-way communication patterns in hardware including some simple calculations on the data. When the X10 runtime is configured for these systems, the X10 team operations map directly to the hardware implementations available, offering performance that cannot be matched by point-to-point messages. When unavailable, our emulation layer kicks in.

RDMAs. RDMA (Remote Direct Memory Access) hardware, such as the Torrent or InfiniBand, enables the transfer of segments of memory from one machine to another without local copies and without the involvement of the CPU or operating system. This technology significantly reduces the latency of data transfers, and frees the CPU to do other work while the transfer is taking place.

We modify the `Array.asyncCopy` implementation to take advantage of RDMAs. We also surface the "GUPS" RDMA feature of the Torrent, which enables atomic remote memory updates (e.g., XOR a memory location with an argument data word).

Congruent Memory Allocator. To use RDMA or collectives, the application needs to register the memory segments eligible for transfer with the network hardware. Moreover, the task initiating the communication must typically know the effective address of each memory segment involved (both source and destination). We implement a "congruent" memory allocator to allocate arrays backed by registered memory segments (outside of the control of the garbage collector). When using the same allocation sequence in every place, this allocator can be configured for symmetric allocation in order return the same sequence of addresses everywhere.

The Torrent, even more than the CPU, is very sensitive to TLB misses. It is therefore important (essential for RandomAccess) that these memory segments are backed by large pages so as to minimize the number of TLB entries. Our congruent memory allocator makes use of large pages if supported and enabled on the system.

Importantly for productivity, the allocation, garbage collection, or use of regular data is not affected at all. Congruent arrays do not behave differently from regular arrays after their initial allocation except of course for supporting extra communication primitives. Ultimately, we only want the end-user to designate arrays for congruent allocation. Everything else should then be handled automatically by the runtime system.

3.4 Scalable Load Balancing

Because irregular workloads are becoming the norm rather than the exception, there is a demand for dynamic distributed load balancing techniques. By applying dynamic distributed load balancing, a runtime system can effectively dynamically distribute or redistribute computationally-intensive tasks across CPU cores within and across shared-memory nodes to maximize utilization.

Saraswat et al. have developed a global load balancing library in X10 (GLB) based on insights from the Unbalanced Tree Search scheduler they described in [35]. GLB takes care of distributed rebalancing by permitting idle places to "steal" work from other places. The choice of the victim sequence is key to the scalability of distributed work stealing: who to try steal from, when, how often, when to back off, etc. GLB handles this.

In Section 6, we describe generic improvements to the load balancer as well as UTS-specific optimizations that make it possible to scale this highly-unbalanced workload to petascale systems for the first time. We have recently integrated the generic elements of this new scheduler to the GLB library and implemented UTS and BC using GLB [43].

4. The Power 775 System

We gathered our performance results on a Power7-based Power 775 supercomputer named Hurcules [29, 30].

The smallest building block of the machine is called an *octant* or simply a host. An octant is composed of a quad-chip module containing four eight-core 3.84 Ghz Power7 processors, one optical connect controller chip (codenamed *Torrent*), and 128 GB of memory. The peak bi-directional bandwidth between any two chips is 96 GB/s (direct link).

A single octant has a peak performance of 982 Gflop/s, a peak memory bandwidth of 512 GB/s, and a peak bi-directional interconnect bandwidth of 192 GB/s. Each octant forms a single SMP node running an operating system image. One physical *drawer* consists of eight octants. Four drawers are connected together to form a *supernode*.

The full machine we used for our measurements contains 56 supernodes, with 1740 octants (55,680 cores) available for computation. This gives the system a theoretical peak of 1.7 Pflop/s. For some benchmarks, we only used 32,768 cores because our implementations require the number of cores to be a power of two. For others (Section 7), we only had access to 47,040 cores due to operational constraints.

Interconnect. The Power 775 system is organized into a two-level direct-connect topology [2] with "L" links connecting every pair of octants within a supernode and "D" links connecting every pair of supernodes in the system.

1. Every octant in a drawer is connected to every other octant of this drawer using a "L" Local link (LL) with a peak bandwidth of 24 GB/s in each direction.
2. Every octant in a supernode is connected to every other octant in the other three drawers of this supernode using a "L" Remote link (LR) with a peak bandwidth of 5 GB/s in each direction.
3. Every supernode is connected to every other supernode using "D" links – eight of them in the current configuration for a combined peak bandwidth of 80 GB/s in each direction.

As a result, any two octants are at most three hops away (L-D-L). We configure the routing protocol for "direct striped" routes with `MP_RDMA_ROUTE_MODE=hw_direct_striped`. Intra-supernode messages only use a single L link (LL or LR). Inter-supernode messages only use the direct D links between the two supernodes (in addition to L links within these supernodes if needed) but are permitted to spread across all eight parallel D links.

In order to understand the bandwidth characteristics of a system partition of a given size, one has to account for two factors: the number and peak bandwidth of the various links (LL, LR, and D) as well as the peak interconnect bandwidth of each octant. Please refer to [38] for a thorough bandwidth analysis. In short, as we scale from one octant to a drawer to a supernode to the full system, we will observe three performance modes:

- With one supernode or less, the cross-section bandwidth is limited by the peak interconnect bandwidth of each individual octant.
- With a few supernodes, the cross-section bandwidth is limited by the aggregated D link bandwidth.
- With many supernodes, the cross-section bandwidth is again limited by the per-octant interconnect bandwidth.

In particular, there is a sharp drop in All-To-All bandwidth per octant when going from one supernode to two supernodes, followed by a slow recovery when further increasing the number of supernodes, followed by a plateau.

Software Configuration. Each of the octants runs RedHat Enterprise Linux 6.1 and uses the IBM Parallel Active Messaging Interface (PAMI) for network communication.

We compiled the benchmark programs using native X10 version 2.2.3 with and compiled the resulting C++ files with xlC version 11 with the `-qinline -qhot -O3 -q64 -qarch=auto -qtune=auto` compiler options. For the FFT and HPL kernels we used native implementations of key numerical routines from FFTE and IBM ESSL respectively. Our UTS code calls a native C routine to compute SHA1 hashes.

We executed the programs in a mode in which each X10 place contained a single worker thread (`X10_NTHREADS=1`) on which the X10 runtime scheduler dispatched the activities for that place. Moreover each core in the system supported exactly one X10 place. To minimize OS jitter, each X10 place was bound to a specific core (by setting `X10RT_CPUMAP`).

In the remainder of the paper, we use the terms place and core interchangeably to measure scale. A node or octant or host has 32 cores and runs 32 places.

While we always rely on PAMI to communicate among places even if they belong to the same octant, PAMI itself leverages shared memory to optimize intra-node communications.

We report results with all optimizations turned on for all benchmarks. Unfortunately, we did not have sufficient time allocated on this supercomputer to assess the benefits of our techniques on a per-optimization or per-benchmark basis. For some of these optimizations however (including `FINISH_DENSE` in UTS, scalable broadcast in the HPCC benchmarks, RDMA-based All-To-All collective in FFT), we observed that the runs at scale do not terminate (in any reasonable amount of time) without the optimization.

5. HPC Challenge Benchmarks

The HPC Challenge benchmark collection was designed not only to better characterize the capabilities of supercomputers than a single measurement can (Class 1 competition), but also to provide an opportunity to compare programming languages and models for concurrency and distribution (Class 2 competition) [17]. In the Class 2 competition, entries are judged for both performance and elegance.

In 2012, IBM entered both the Class 1 and Class 2 competition for this Power 775 system. IBM's Class 1 implementations are intended to demonstrate the highest performance achievable by this system [30]. They are written in a mix of C and assembly code. They are specifically tailored for the Power 775 architecture and carefully hand- and auto-tuned. They interface directly with the hardware de-

vice drivers bypassing the entire network stack. They rely on ad-hoc benchmark-specific communication protocols. They are intrinsically non-modular, i.e., cannot be composed into larger application codes. In contrast, the X10 implementations we entered into the Class 2 competition are built upon a common network stack and a few constructs (finish, async, at) that can be composed arbitrarily. The X10 code has been tuned for Power 775 to a much lesser extent, and runs unchanged on commodity clusters.

The implementations and performance results we describe in this section are essentially those obtained for our winning 2012 Class 2 entry to the HPC Challenge [39] with minor updates. We first discuss our implementations of the four benchmarks, then analyze and compare X10 performance with IBM's optimized runs as reported in IBM's 2012 Class 1 entry [15, 16].

5.1 Implementation

Our SPMD-style implementations of the four benchmarks mimic the main attributes of the reference implementations. We discuss each benchmark in turn.

Global HPL. Global HPL measures the floating point rate of execution for solving a linear system of equations. Performance is measured in Gflop/s.

Our implementation features a two-dimensional block-cyclic data distribution, a right-looking variant of the LU factorization with row-partial pivoting, and a recursive panel factorization. It lacks however various refinements of the reference implementation such as configurable look-ahead, configurable recursion depth in the panel factorization, and configurable collective communications. We simply use default PAMI collectives (via X10's Team class).

Our implementation uses a collection of idioms for communication: asynchronous array copies for row fetch or swap and teams for barriers, row and column broadcast, and pivot search. We take advantage of `finish` pragmas to make sure the compiler and runtime recognize that a row swap is a simple message exchange for instance.

Global RandomAccess. Global RandomAccess measures the system's ability to update random memory locations in a table distributed across the system, by performing XOR operations at the chosen locations with random values. Because the memory is spread across all the places, any update to a random location is likely to be an update to memory located at a remote place, not the local place. Performance is measured by how many Gup/s (Giga Updates Per Second) the system can sustain.

Our implementation takes advantage of congruent memory allocation to obtain a distributed array backed by large pages where the per-place array fragment is at the same address in each place. It then uses the Torrent's "GUPS" RDMA for the remote updates.

Global FFT. Global FFT performs a 1D discrete Fourier transform on an array of double-precision complex values. The source and destination arrays are evenly distributed across the system. FFT stresses the floating point units, network, and memory subsystems. Performance is measured in Gflop/s.

Our implementation alternates non-overlapping phases of computation and communication on the array viewed as a 2D matrix: global transpose, per-row FFTs, global transpose, multiplication with twiddle factors, per-row FFTs, and global transpose. The global transposition is implemented with local data shuffling, followed by an All-To-All collective, and then finally another round of local data shuffling.

EP Stream. EP Stream (Triad) measures sustainable local memory bandwidth. It performs a scaled vector sum with two source vectors and one destination vector. Performance is measured in GB/s.

Our implementation of this benchmark follows a straightforward SPMD style of programming. The main activity launches an activity at every place using a `PlaceGroup` broadcast. These activities then allocate and initialize the local arrays, perform the computation, and verify the results. The backing storage for the arrays is allocated using huge pages to enable efficient usage of TLB entries.

5.2 Performance Results

For all runs, places are mapped to hosts in groups of 32. In particular, runs with 32 places or less use a single host. We run with up to 32,768 places in power-of-two increments. For Stream, we also run with the full system: 55,680 places. We use a constant per-place amount of memory (weak scaling). The exact amount is chosen in accordance with the HPC Challenge guidelines.

For every benchmark, we the same X10 implementation for all runs. In particular, our single-core "sequential" runs still use the full, distributed, parallel X10 code.

We plot the resulting performance curves in Figure 1. For each kernel we plot both the aggregated performance as well as the per-core performance (per-host for RandomAccess). We'll discuss the later four kernels in Sections 6 and 7.

Class 1 Comparison. In Table 1, we compare our performance results with IBM's HPC Class 1 optimized runs on the system [16].

Unfortunately, the comparison at scale is only indirect since the reference runs were obtained with larger core counts, which were not available to us. In addition, our FFT, HPL, and RandomAccess implementations require the number of cores to be powers of two. We compare the per-core performance for the largest X10 and Class 1 runs. Due to the nature of the benchmarks (for HPL and Stream) and the architecture of the network (for RandomAccess and FFT), we believe the Class 1 per-core performance results should be relatively stable between 32K and 64K cores.[5]

[5] An earlier Class 1 submission for a smaller 1470-host Power 775 system [15] show results within 7% of the more recent 1989-host submission [16] we use for comparison purposes.

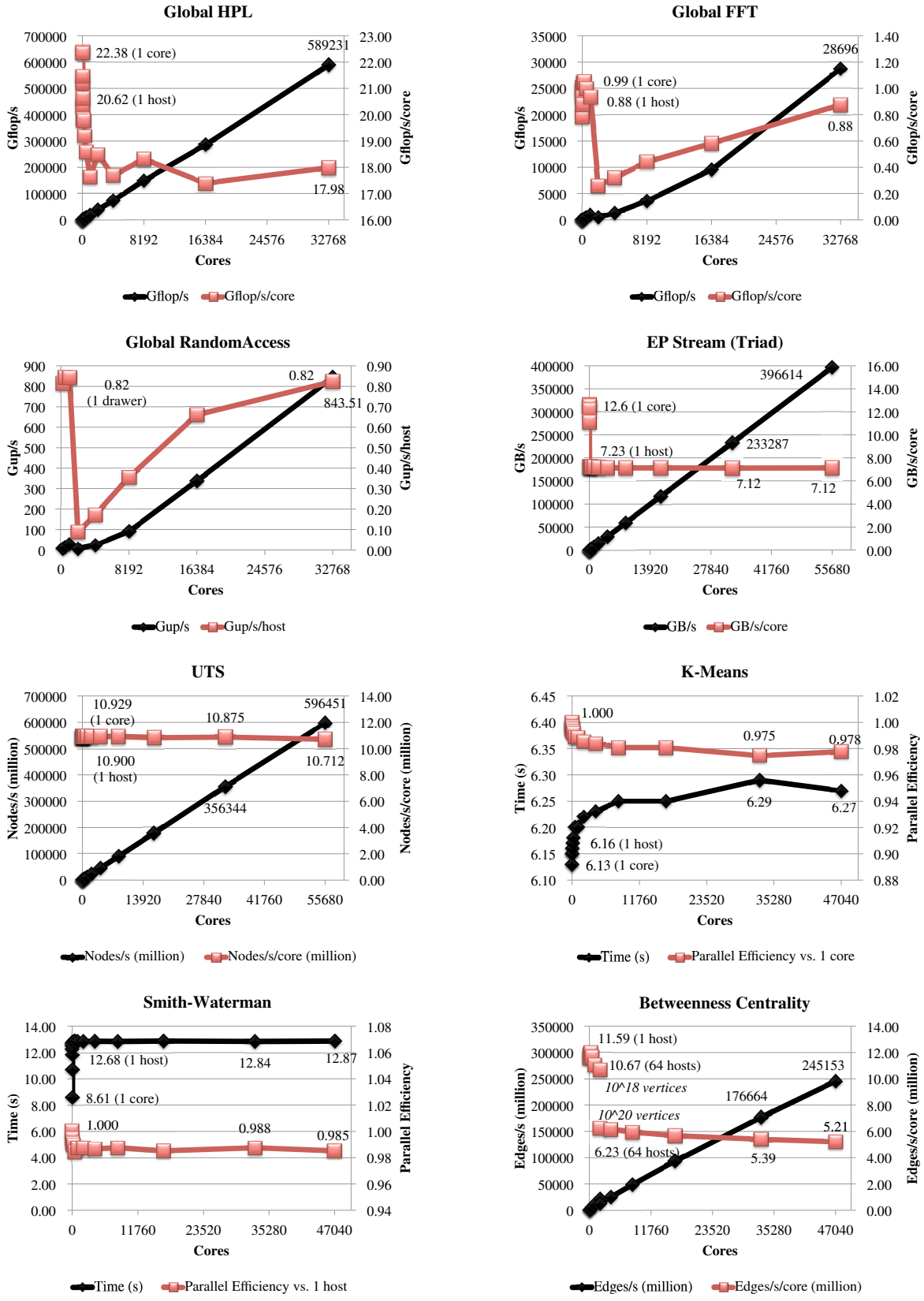

Figure 1. X10 Performance Results

Global HPL. We use about 55% of the memory of each host and a block size of 360. With 32,768 places, we measure an aggregated performance of about 589 Tflop/s, that is, about 60% of the theoretical peak of 1,024 hosts, which amounts to about 70% of the effective `DGEMM` peak performance (ESSL).

The single-core performance is 22.38 Gflop/s. The single-host performance is 20.62 Gflop/s/core. The performance at scale is 17.98 Gflop/s/core. Hence, the X10 code achieves a relative efficiency of 80% at scale. The efficiency drops primarily when scaling from 1 to 1,024 cores. Above 1,024 cores, the efficiency curve flattens. The seesaw is an artifact of the switch from a $n*n$ to a $2n*n$ block cyclic distribution for even and odd powers of two.

X10 achieves about 85% of the performance of the Class 1 run at scale (Gflop/s/core), which divides a matrix of approximately the same size across 1,989 hosts instead of 1,024.

Global RandomAccess. The per-place table size is fixed at 2 GB for a total of 64 GB per host, that is, half of the available memory. We plot results for eight hosts (one drawer) and above.

We measure 0.82 Gup/s/host at both end of the spectrum, that is, with 8 hosts and with 1,024 hosts. In either case, the per-host interconnect bandwidth is the bottleneck, hence the "perfect" relative efficiency. In-between the per-host Gup/s number is significantly lower because the cross-section bandwidth becomes a bottleneck as discussed in Section 4. For smaller runs (4 hosts or less, not plotted), other network bottlenecks come into play (switching, beyond the scope of this paper), and the performance per host is lower. These phases are intrinsic to the interconnect and can be observed also with the Class 1 or UPC code.

Overall, the performance of the X10 code matches the performance of the UPC code for the same benchmark, as well as the performance of a direct C implementation against PAMI. In comparison, the reference code (which bypasses PAMI) achieves 1.02 Gup/s/host at scale.

Global FFT. For even powers, each place uses 2 GB of memory. For odd powers, only 1 GB is used. Hence either half or a quarter of the memory of the host.

With one place, we measure 0.99 Gflop/s. At scale, the rate reduces to 0.88 Gflop/s per core. In-between, as we observed for RandomAccess, the per-core performance is significantly hindered by the relatively low cross-section bandwidth.

We only achieve about 41% of the per-core Class 1 performance at scale. The primary bottleneck lies in our sequential code. On Power 775, the All-To-All time represents only a small fraction of FFT's total execution time. Unfortunately, we did not have sufficient time on the system to hand- or auto-tune our sequential code (data shuffle and 1D FFT) or to experiment with computation communication overlap.

EP Stream. We use 1.5 GB of memory per place.

The per-place memory bandwidth decreases as the number of places per host increases. It drops from 12.6 GB/s with one place to 7.23 GB/s with 32 places due to the QCM architecture for a total of 231.5 GB/s for a single host. Our single-host measurements match the performance of the reference OpenMP EP Stream (Triad) implementation.

With 32 places and above, the per-place memory bandwidth is essentially constant. The total system bandwidth at 55,680 places (1,740 hosts) is about 397 TB/s, which exceeds 98% of 1,740 times the single-host bandwidth. We attribute the 2%-loss to jitter and synchronization overheads.

Our single-host number represents about 87% of the Class 1 result, which takes advantage of Power7 prefetching instructions that we do not. It may be possible to use xlC to generate assist threads for data prefetching for this benchmark and others, but we have not yet experimented with doing so.

Summary. As shown in Table 1, our relative performance compared to the Class 1 runs varies from 41% for FFT to 87% for Stream. As discussed earlier, due to configuration discrepancies these numbers are only estimates. Nevertheless, we believe that they demonstrate that X10 can deliver solid performance at scale even compared to the best codes tuned by IBM's best experts.

In Table 2, we compute that the per-host performance at scale of the X10 code never drops below 87% of the single-host performance for any of these four benchmarks.

We select this particular comparison as our main scalability metric for two reasons. First, the performance scaling from one core to one host is not expected to be linear in general as the memory bandwidth does not scale linearly due to bus contention. Second, the performance per host for partitions of intermediate size is gated by the cross-section bandwidth bottleneck for communication-bound benchmarks.

6. Unbalanced Tree Search

The workload in any of the four HPC Class 2 Challenge benchmarks can be partitioned statically across a distributed system effectively. We now consider a benchmark that is not amenable to such static partitioning but requires advanced dynamic distributed load balancing techniques to scale to large distributed systems.

The Unbalanced Tree Search benchmark measures the rate of traversal of a tree generated on the fly using a splittable random number generator. For this work, we used a *fixed geometric law* for the random number generator with branching factor $b_0 = 4$ and seed $r = 19$, and tree depth d varying from 14 with one place to 22 at scale (weak scaling).

> The nodes in a geometric tree have a branching factor that follows a geometric distribution with an expected value that is specified by the parameter $b_0 > 1$. The parameter d specifies its maximum depth cut-off,

Benchmark	X10 Implementation		HPC Class 1 Optimized Runs		Relative Performance
	Cores	Performance at Scale	Cores	Performance at Scale	
Global HPL	32,768	589.231 Tflop/s	63,648	1343.67 Tflop/s	85%
Global RandomAccess	32,768	843.58 Gup/s	63,648	2020.77 Gup/s	81%
Global FFT	32,768	28,696 Gflop/s	62,208	132,658 Gflop/s	41%
EP Stream (Triad)	32	231.481 GB/s	32	264.156 GB/s	87%

Table 1. Performance Comparisons for the HPC Class 2 Challenge Benchmarks

Benchmark	Performance of the X10 Implementation		Host Count	Relative Efficiency
	With One Host	At Scale	At Scale	
Global HPL	20.62 Gflop/s/core	17.98 Gflop/s/core	1,024	87%
Global RandomAccess	0.82 Gup/s/host	0.82 Gup/s/host	1,024	100%
Global FFT	0.88 Gflop/s/core	0.88 Gflop/s/core	1,024	100%
EP Stream (Triad)	7.23 GB/s/core	7.12 GB/s/core	1,740	98%
UTS	10.900 M nodes/s/core	10.712 M nodes/s/core	1,740	98%
K-Means	6.16s run time	6.27s run time	1,470	98%
Smith-Waterman	12.68s run time	12.87s run time	1,470	98%
Betweenness Centrality	11.59 M edges/s/core	5.21 M edges/s/core	1,470	45%

Table 2. Relative Efficiency: Performance at Scale versus Single-Host Performance (for the Same X10 Implementation)

> beyond which the tree is not allowed to grow ... The expected size of these trees is $(b_0)^d$, but since the geometric distribution has a long tail, some nodes will have significantly more than b_0 children, yielding unbalanced trees. [25]

The depth cut-off makes it possible to estimate the size of the trees and guess parameters for a target execution time, but should not be used to predict subtree sizes. In other words, all nodes are to be treated equally for load balancing purposes, irrespective of the current depth.

6.1 Implementation

Our implementation follows from [35]. It uses global work-stealing with random steals followed by lifeline steals and hyper-cubes for the lifeline graphs.

Lifeline-based Global Work Stealing. Every worker (one per place) maintains a list of pending nodes to process (nodes not counted yet). Each worker primarily processes its own list. If the list becomes empty, the idle worker attempts to steal nodes from another worker. Steal attempts are first random – a victim worker is randomly selected – and synchronous – the thief wait for the attempt to complete either successfully – the victim list was not empty – or unsuccessfully – the victim was also idle. Past a few random attempts, the thief falls back to a fixed precomputed list of victims called lifelines, sends requests to these and dies. Lifelines have memory. If a request for work cannot be served immediately because the lifeline itself is idle, then if the lifeline later manages to obtain new nodes to process, it will split these nodes between itself and others as requested, resuscitating dead workers in the process.

Intuitively, random attempts are very effective at distributing work when most workers are busy, whereas lifelines are effective at propagating work quickly when many workers are idle. Lifeline "edges" are organized in graphs with both low diameters and low degree such as hyper-cubes to co-minimize the distance between any two workers and the number of lifeline requests in flight.

Because workers die when unsuccessful at random steals the overall termination of the counting can be implemented with a single `finish` construct. Each worker is implemented with one `async` task. Resuscitation is also one `async` task.

Work is initially distributed from the root worker in one tree-shaped wave. See [35] for more details.

Refinements. We first improve the scalability beyond [35] by (i) further reducing the overhead of termination detection, (ii) shaping the network traffic, and (iii) improving the work-queue implementation.

We use the `FINISH_DENSE` algorithm for the root `finish` responsible for detecting the termination of the tree traversal. We use an algorithm akin to `FINISH_HERE` to detect the termination of steal attempts – outgoing request followed by incoming response.[6] The root `finish` only accounts for the initial work distribution as well as the redistribution along lifelines but is oblivious to rebalancing operations resulting from successful random steal attempts.

We precompute for each place a set of potential victims with no more than 1,024 elements to bound the out-degree of the communication graph. We observe a severe degradation of the network performance at scale without such a bound.

[6] We could use the standard `FINISH_HERE` algorithm instead, but we have not yet rewritten our UTS code to confirm that the performance is the same.

We adopt a more compact representation of the nodes remaining to be processed in a place, by directly representing intervals of siblings in the tree as intervals (lower, upper bounds) instead of using expanded lists of nodes.

Finally, to counteract the bias introduced by the depth cutoff, a thief steals fragments of every interval in the work list. There are few of them since we traverse the tree depth first.

The last two changes are tailored for UTS for shallow trees and make a tremendous difference for these, but are not likely to help as much for deep and narrow trees. The earlier improvements however are applicable to distributed work stealing in general. We include them in the GLB library (see Section 3.4).

6.2 Performance Results

We run from one to 32,768 places in power-of-two increments and then at scale with 55,680 places. Figure 1 shows the total number of nodes processed per second as well as the per-place processing rate. We increase the depth of the tree from 14 with one place to 22 with 55,680 places so as to obtain runs of duration ranging from 90 to 200 seconds.

The per-place processing rate varies from 10.929 million nodes per second for one place to 10.712 million nodes per second at scale. The single-place performance is identical to the performance of the sequential implementation (no parallelism, distribution, or load balancing). The distributed implementation at scale achieves 98% parallel efficiency.

At scale, we traverse a tree of 69,312,400,211,958 nodes in 116s, that is, about 596 billion nodes/s. As part of this traversal, we compute 17,328,102,175,815 SHA1 hashes (for random number generation). In contrast, the algorithm from [35] achieves its peak performance with a few thousand cores and slows down to a crawl beyond that due to overwhelming termination detection overheads and network contention. We tried traversing the same tree at scale with the original algorithm and had to kill the run after an hour.

7. Other Benchmarks

In this section, we briefly review the three remaining kernels: K-Means, Smith-Waterman, and Betweenness Centrality.

K-Means. K-Means clustering aims to partition n points in a d-dimensional space into k clusters so as to minimize the average distance to cluster centroids. We implement Lloyd's algorithm [21]. Given arbitrary initial centroids, it iteratively refines the clusters and centroids.

We partition the points across p places. In parallel at each place, we classify the points by nearest centroid and compute the average positions of the per-place points in each cluster. Then we use two All-Reduce collectives to compute the averages across all places. They provide updated centroids for the next iteration.

We run with $40000p$ points with p places and 4096 clusters (dimension 12). We measure the running time for 5 iterations. It varies from 6.13s with one place to 6.27s with 47,040 places. Efficiency versus a single place never drops below 97%.

Smith-Waterman. The Smith-Waterman algorithm is a dynamic programming algorithm to compute the best possible alignment (partial match) of a short DNA sequence against a long DNA sequence.

We parallelize the computation by splitting the long sequence into overlapping fragments and computing in parallel the best match of the short sequence against each fragment. The best overall match is the best of the best matches.

We run with a short string of 4000 elements and a long string with $40000p$ elements. We report the running time for 5 iterations of the benchmark. We measure 8.61s for one place, 12.68s for one host (32 places), and 12.87s at scale with 47,040 places. The running time increases from 1 to 32 places because of memory bus contention. Scaling out from one host to 1,470 hosts we only lose 2% efficiency.

Betweenness Centrality. Betweenness centrality measures the "centrality" of a node in a graph, i.e., the number of shortest paths from all vertices to all others that pass through that node. We compute this measure for each node in an undirected R-MAT graph [6] using Brandes' algorithm [5].

Since even a small graph incurs a significant amount of computation, we replicate the graph in every place. We randomly partition the vertices across places. Each place is responsible for computing the centrality measure for all its vertices. These computations are local and independent.

We run with one host up to 1,470 hosts. Because BC is not amenable to perfect weak scaling, we consider two different problem instances. With 2,048 places or less, we consider a graph with 2^{18} vertices and 2^{21} edges. With 2,048 places or more, we consider a graph with 2^{20} vertices and 2^{23} edges. At scale we traverse 245 Billion edges per second.

There is a significant performance drop at 2,048 places when we increase the problem size, due – we speculate – to the increased footprint of the graph. With a fixed-size graph between 32 and 2,048 places, the number of traversed edges per place and per second reduces from 11.59 million to 10.67 million. With the larger graph, we measure 6.23 million edges/s/place with 2,048 places and 5.21 million with 47,040 places. Therefore, while the measured relative efficiency is only 45% at scale (Table 2), the "corrected" efficiency is 77% if we discount the performance drop due to the switch to a larger graph.

The remaining 23% primarily results from increasing unbalance. The computation of the centrality measure takes a variable amount of time depending on the vertex position in the graph. By randomizing the partition, we can mitigate this unbalance, but only to a degree. The smaller the parts, the higher the imbalance.

Since we collected these results, we have implemented BC on top of the GLB library to dynamically distribute the load across all places [43]. The resulting code has better efficiency.

8. Related Work

We structure the discussion of related work around the three main contributions of this paper: scalability of APGAS constructs, performance evaluation of high-productivity programming models, and algorithmic advances for dynamic distributed load balancing in the context of the Unbalanced Tree Search problem.

APGAS scalability. Our work on scalable `finish` is related to prior work in the Charm parallel programming system to develop a scalable algorithm for quiescence detection [36]. As in X10, Charm's core execution model combines asynchronous distributed tasks with the ability to determine when a collection of tasks has completed. Charm++ supports the ability to detect completion of a set of tasks or global quiescence of the entire computation [19]. Algorithmically, Charm's *Counting* algorithm and the *Task Balancing* algorithm [4] used by X10 as the most general implementation of `finish` make different time/space trade-offs. The Counting algorithm uses asymptotically fewer counters, but requires two phases to detect termination. Each phase involves communicating with all the tasks in scope. This is ill-suited for X10 where termination detection scopes are numerous, usually nested, and overlap places. Additionally, we have developed specialized algorithms for commonly occurring usage patterns `finish`.

The Co-Array Fortran 2.0 `finish` construct also ensures the global completion of a set of asynchronous tasks [23]. Because CAF is SPMD-centric, its `finish` is designed for this special case, specified as a collective operation, and implemented with rounds of termination detection as in Charm++. X10's `finish` is not limited to SPMD patterns, but we provide a specialized implementation for the SPMD case (`FINISH_SPMD` pragma).

Chapel also incorporates distributed asynchronous tasks and finish-like synchronization with its `begin`, `on` and `sync` constructs [9]. The techniques we have developed for scaling `finish` should be applicable to Chapel.

Productivity and performance. As illustrated by the submissions to the HPC Challenge Class 2 competitions [14], there has been significant prior work on improving the performance and scalability of high-productivity programming models. Performance results on machines ranging from hundreds to tens of thousands of nodes have been reported for PGAS languages such as Co-Array Fortran [23], UPC [1], Chapel [7], and X10 [39] and for programming models such as Charm++ [18] and XcalableMP [24]. The results we report in this paper represent significantly higher levels of absolute performance and/or scalability than was achieved in previous systems.

X10's productivity has been modeled as well as empirically studied by IBM as part of the HPCS PERCS project [12] and independently by X10 users [27]. While type or memory safety have the same productivity benefits at small and large scale, prior productivity analyses have not confirmed or refuted the value of APGAS constructs at very large scale. With this work, we demonstrate that these constructs do perform at very large scale, hence continue to deliver productivity to the programmer.

Our congruent memory allocator is similar to many symmetric memory allocators. It permits data structures to be at predictable but not necessary identical addresses in each place. This helps maximizing TLB utilization when multiple places are running on the same node. The IBM UPC implementation studied approaches for scalable congruent memory allocation and developed the Shared Variable Directory as one such scalable technique [3]. Our approach is more restricted, but does demonstrate how custom memory allocation can be integrated into a type safe and mostly garbage-collected programming language.

UTS. UTS, an excellent example of an irregular application, was first described by Prins et al. [28]. Olivier and Prins [26] provided the first scalable implementation of UTS on clusters that provided up to 80% efficiency on 1,024 nodes. To this end, they employed a custom application-level load balancer along with optimizations such as one-sided communications and novel termination detection techniques. Dinan et al. [10] provide a framework for global load balancing, which was used to achieve speedups on 8196 processors. Global load balancing and termination detection facilities were provided to express irregular applications. By reserving one core per compute node on the cluster exclusively for lock and unlock operations, this framework allowed threads to steal work asynchronously without disrupting the victim threads. However, the cost paid was a static allocation (one core out of every eight) for communication. This results in lower throughput because the thread is not available for user-level computations. Saraswat et al. [35] introduced lifeline-based global load balancing and showed 87% efficiency on 2048 nodes. An implementation of the life-line algorithm in Co-Array Fortran achieved 58% efficiency at 8192 nodes [23]. A more recent UTS code using CAF 2.0 `finish` construct achieves a 74% parallel efficiency on 32,768 Jaguar cores [42]. In comparison, our code reaches 98% parallel efficiency with 55,680 Power7 cores.

Work-stealing schedulers have been developed for X10 both in the context of intra- and inter-node load balancing. For the intra-node case, pure runtime techniques have been developed [13] as well as compiler-supported techniques [40]. For the inter-node case, the GLB library based on the scheduler developed for UTS in [35] enables the automated distributed load balancing of locality-insensitive tasks using global work stealing.

We have recently incorporated our improvements to the UTS scheduler to the GLB library and implemented Betweenness Centrality on top of GLB [43]. We were able to confirm that GLB can effectively balance BC and improve its efficiency at scale.

9. Summary

We implemented eight application kernels in X10 and ran them at scale on a petaflop Power 775 supercomputer. Excluding Betweenness Centrality, we measure an efficiency at scale (for 1,024 hosts or more) consistently above 87% of the single-host efficiency. For the four HPC Class 2 Challenge benchmarks, X10 achieves 41% to 87% of the top performance numbers reported for this system.

We show that X10's `finish` construct for distributed termination detection delivers performance at scale. We use `finish` to block for the delivery of a single asynchronous message, wait for the termination of a regular SPMD-style distributed computation, or control a distributed load balancing kernel, among other things. We believe there is a great productivity benefit in having a unique, universal, yet scalable mechanism for termination detection.

We show that hardware-accelerated communication primitives like RDMAs and collectives can be integrated into X10 via libraries and that accounting for low-level memory requirements can be achieved without crippling X10's automated memory management and safety.

To the best of our knowledge, our UTS implementation for geometric trees is the first to scale to petaflop systems. Asynchrony and distributed termination detection are essential to any scalable UTS implementation. We believe that the X10 language and tools gave us the ability to experiment with the UTS code like no other programming model would, ultimately giving us the keys to performance at scale.

We focus on scale out: we want as many places as possible to stress our `finish` implementations, etc. Therefore, we run with one place per core and implement the benchmark codes with minimal intra-place concurrency. A more natural APGAS implementation however would take advantage of intra-place concurrency, run with only one or a few places per host, and probably perform marginally better.

While collecting our performance results, we observed many times the practical benefits of the asynchronous programming style advocated for by X10. If a single core is not performing optimally, a statically scheduled code like HPL suffers greatly. With UTS however, there is no measurable impact as the load is dynamically pulled from the bad core.

Acknowledgments

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

Earlier versions of the code presented here were worked on by Sreedhar Kodali, Ganesh Bikshandi, Pradeep Varma, Krishna Venkat and other colleagues.

We would like to thank all our colleagues on the PERCS project who made the collection of the performance results possible. We would especially like to thank Kevin Gildea, Vickie Robinson, Pat Esquivel, Pat Clarke, George Almasi, Gabriel Tanase, and Ram Rajamony.

References

[1] G. Almási, B. Dalton, L. L. Hu, F. Franchetti, Y. Liu, A. Sidelnik, T. Spelce, I. G. Tanase, E. Tiotto, Y. Voronenko, and X. Xue. 2010 IBM HPC Challenge Class II Submission, Nov. 2010.

[2] B. Arimilli, R. Arimilli, V. Chung, S. Clark, W. Denzel, B. Drerup, T. Hoefler, J. Joyner, J. Lewis, J. Li, N. Ni, and R. Rajamony. The PERCS high-performance interconnect. In *Proceedings of the 2010 18th IEEE Symposium on High Performance Interconnects*, HOTI '10, pages 75–82, Washington, DC, USA, 2010. IEEE Computer Society.

[3] C. Barton, C. Casçaval, G. Almási, Y. Zheng, M. Farreras, S. Chatterje, and J. N. Amaral. Shared memory programming for large scale machines. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 108–117, New York, NY, USA, 2006. ACM.

[4] S. M. Blackburn, R. L. Hudson, R. Morrison, J. E. B. Moss, D. S. Munro, and J. Zigman. Starting with termination: a methodology for building distributed garbage collection algorithms. In *Proceedings of the 24th Australasian conference on Computer science*, ACSC '01, pages 20–28, Washington, DC, USA, 2001. IEEE Computer Society.

[5] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001.

[6] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *In SDM*, 2004.

[7] B. Chamberlain, S.-E. Choi, M. Dumler, T. Hildebrandt, D. Iten, V. Litvinov, G. Titus, C. BaAaglino, R. Sobel, B. Holt, and J. Keasler. Chapel HPC Challenge Entry: 2012, Nov. 2012.

[8] S. Crafa, D. Cunningham, V. Saraswat, A. Shinnar, and O. Tardieu. Semantics of (Resilient) X10. `http://arxiv.org/abs/1312.3739`, Dec. 2013.

[9] Cray. Chapel language specification version 0.93. Apr. 2013.

[10] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

[11] J. Dongarra, R. Graybill, W. Harrod, R. Lucas, E. Lusk, P. Luszczek, J. Mcmahon, A. Snavely, J. Vetter, K. Yelick, S. Alam, R. Campbell, L. Carrington, T.-Y. Chen, O. Khalili, J. Meredith, and M. Tikir. DARPA's HPCS Program: History, Models, Tools, Languages. In M. V. Zelkowitz, editor, *Advances in COMPUTERS High Performance Computing*, volume 72 of *Advances in Computers*, pages 1 – 100. Elsevier, 2008.

[12] K. Ebcioglu, V. Sarkar, T. El-Ghazawi, and J. Urbanic. An experiment in measuring the productivity of three parallel programming languages. In *P-PHEC workshop, held in conjunction with HPCA*, February 2006.

[13] D. Grove, O. Tardieu, D. Cunningham, B. Herta, I. Peshansky, and V. Saraswat. A performance model for X10 applications: what's going on under the hood? In *Proceedings of the 2011*

ACM SIGPLAN X10 Workshop, X10 '11, pages 1:1–1:8, New York, NY, USA, 2011. ACM.

[14] HPC Challenge Awards Competition. `http://www.hpcchallenge.org/`.

[15] HPC Challenge Benchmark Record 482. `http://icl.cs.utk.edu/hpcc/hpcc_record.cgi?id=482`, July 2012.

[16] HPC Challenge Benchmark Record 495. `http://icl.cs.utk.edu/hpcc/hpcc_record.cgi?id=495`, Nov. 2012.

[17] HPC Challenge Benchmarks. `http://icl.cs.utk.edu/hpcc/`.

[18] L. V. Kalez, A. Arya, A. Bhatele, A. Gupta, N. Jain, P. Jetley, J. Lifflander, P. Miller, Y. Sun, R. Venkataramanz, L. Wesolowski, and G. Zheng. Charm++ for Productivity and Performance, Nov. 2011.

[19] P. P. Laboratory. The Charm++ Parallel Programming System Manual. Technical Report Version 6.4, Department of Computer Science, University of Illinois, Urbana-Champaign, 2013.

[20] J. K. Lee and J. Palsberg. Featherweight X10: a core calculus for async-finish parallelism. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 25–36, New York, NY, USA, 2010. ACM.

[21] S. Lloyd. Least squares quantization in PCM. *IEEE Trans. Inf. Theor.*, 28(2):129–137, Sept. 2006.

[22] J. MacQueen. Some methods for classification and analysis of multivariate observations. Proc. 5th Berkeley Symp. Math. Stat. Probab., Univ. Calif. 1965/66, 1, 281-297 (1967)., 1967.

[23] J. Mellor-Crummey, L. Adhianto, G. Jin, M. Krentel, K. Murthy, W. Scherer, and C. Yang. Class II Submission to the HPC Challenge Award Competition Coarray Fortran 2.0, Nov. 2011.

[24] M. Nakao, H. Murai, T. Shimosaka, and M. Sato. XcalableMP 2012 HPC Challenge Class II Submission, Nov. 2012.

[25] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng. UTS: an unbalanced tree search benchmark. In *Proceedings of the 19th international conference on Languages and compilers for parallel computing*, LCPC'06, pages 235–250, Berlin, Heidelberg, 2007. Springer-Verlag.

[26] S. Olivier and J. Prins. Scalable dynamic load balancing using UPC. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 123–131, Washington, DC, USA, 2008. IEEE Computer Society.

[27] J. Paudel and J. N. Amaral. Using the Cowichan problems to investigate the programmability of X10 programming system. In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, X10 '11, pages 4:1–4:10, New York, NY, USA, 2011. ACM.

[28] J. Prins, J. Huan, B. Pugh, C.-W. Tseng, and P. Sadayappan. UPC Implementation of an Unbalanced Tree Search Benchmark. Technical Report 03-034, Univ. of North Carolina at Chapel Hill, October 2003.

[29] D. Quintero, K. Bosworth, P. Chaudhary, R. G. da Silva, B. Ha, J. Higino, M.-E. Kahle, T. Kamenoue, J. Pearson, M. Perez, F. Pizzano, R. Simon, and K. Sun. *IBM Power Systems 775 for AIX and Linux HPC Solution*. IBM, 2012.

[30] R. Rajamony, M. W. Stephenson, and W. E. Speight. The Power 775 Architecture at Scale. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 183–192, New York, NY, USA, 2013. ACM.

[31] V. Saraswat, G. Almasi, G. Bikshandi, C. Cascaval, D. Cunningham, D. Grove, S. Kodali, I. Peshansky, and O. Tardieu. The Asynchronous Partitioned Global Address Space Model. In *AMP'10: Proceedings of The First Workshop on Advances in Message Passing*, June 2010.

[32] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. The X10 language specification, v2.2.3. Aug. 2012.

[33] V. Saraswat and R. Jagadeesan. Concurrent clustered programming. In *Concur'05*, pages 353–367, 2005.

[34] V. Saraswat, O. Tardieu, D. Grove, D. Cunningham, M. Takeuchi, and B. Herta. A brief introduction to X10 (for the high performance programmer). `http://x10.sourceforge.net/documentation/intro/latest/html/`, Feb. 2013.

[35] V. A. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy. Lifeline-based global load balancing. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 201–212, 2011.

[36] A. B. Sinha, L. V. Kale, and B. Ramkumar. A dynamic and adaptive quiescence detection algorithm. Technical Report 93-11, Parallel Programming Laboratory, Department of Computer Science , University of Illinois, Urbana-Champaign, 1993.

[37] T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195 – 197, 1981.

[38] G. Tanase, G. Almási, E. Tiotto, M. Alvanos, A. Ly, and B. Dalton. Performance analysis of the IBM XL UPC on the PERCS architecture. Technical Report RC25360, IBM Research, Mar. 2013.

[39] O. Tardieu, D. Grove, B. Bloom, D. Cunningham, B. Herta, P. Kambadur, V. A. Saraswat, A. Shinnar, M. Takeuchi, and M. Vaziri. X10 for Productivity and Performance at Scale, Nov. 2012.

[40] O. Tardieu, H. Wang, and H. Lin. A work-stealing scheduler for X10's task parallelism with suspension. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 267–276, 2012.

[41] Wikipedia. PERCS. `http://en.wikipedia.org/w/index.php?title=PERCS`, 2011.

[42] C. Yang, K. Murthy, and J. Mellor-Crummey. Managing Asynchronous Operations in Coarray Fortran 2.0. In *IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1321–1332, 2013.

[43] W. Zhang, O. Tardieu, D. Grove, B. Herta, T. Kamada, V. Saraswat, and M. Takeuchi. GLB: Lifeline-based Global Load Balancing Library in X10. `http://arxiv.org`, Dec. 2013.