RC25334 (WAT1210-081) October 25, 2012
Computer Science

IBM Research Report

X10 for Productivity and Performance at Scale

A Submission to the 2012 HPC Class II Challange

Olivier Tardieu, David Grove, Bard Bloom, David Cunningham, Benjamin Herta, Prabhanjan Kambadur, Vijay A. Saraswat, Avraham Shinnar, Mikio Takeuchi*, Mandana Vaziri
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

*Tokyo Research Lab.

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

X10 for Productivity and Performance at Scale

A Submission to the 2012 HPC Class II Challenge

Olivier Tardieu, David Grove, Bard Bloom, David Cunningham, Benjamin Herta,
Prabhanjan Kambadur, Vijay A. Saraswat, Avraham Shinnar, Mikio Takeuchi, Mandana Vaziri

IBM

October 23, 2012

We implement all four HPC Class I benchmarks in X10: Global HPL, Global RandomAccess, EP Stream (Triad), and Global FFT. We also implement the Unbalanced Tree Search benchmark (UTS). We show performance results for these benchmarks running on an IBM Power 775 Supercomputer utilizing up to 47,040 Power7 cores. We believe that our UTS implementation demonstrates that X10 can deliver unprecedented productivity and performance at scale for unbalanced workloads.

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

1 Overview

X10 is a high-performance, high productivity programming language developed in the IBM "Productive, Easy-to-use, Reliable Computing System" project (PERCS [12]), supported by the DARPA High Productivity Computer Systems initiative (HPCS [3]). X10 is a class-based, strongly typed, explicitly concurrent, garbage-collected, multi-place object-oriented programming language [8, 7].

Since our last submission to the HPC Class II Challenge in 2009, we have made significant progress towards the development of X10 as a language and programming environment that simultaneously supports high performance and high productivity. With the X10 2.2 release in June 2011, we reached a level of language and implementation maturity that made it practical to begin building large-scale X10 applications. Since then, our work has been driven by the experiences gained by both the core X10 team at IBM and the external X10 community in building out a wide range of X10 frameworks and applications including a Global Load Balancing Framework [9], a framework for defining parallel portfolio-based SAT solvers [1], a Global Matrix Library [13], a Hadoop API-compatible Main Memory Map Reduce engine [10], the ANUChem suite of computational chemistry algorithms [5], the ScaleGraph distributed graph algorithm library [2] and the XAXIS large scale agent simulation framework [11]. Driven by this application-based focus, we have made significant refinements in the X10 language, the tool chain implementing it, and the underlying core class libraries and runtime system.

To bring modern development tooling to the X10 programmer, we have also invested in the development of X10DT, an Eclipse-based X10 Integrated Development Environment, and a source-level debugger for X10. X10DT provides X10 programmers the expected set of core IDE functionality for X10 application development in conjunction with the capability to manage the remote build and execution of X10 programs being developed on their laptop on multiple HPC systems. The IBM Parallel Debugger for X10 Programming integrates with X10DT and extends the production IBM Parallel Debugger with support for source-level debugging of X10 programs on selected platforms.

In this submission, we focus on the progress we have made on enabling X10 applications to run at very large scale. We therefore only present experimental results for five kernels, compiled using only one of the two X10 compiler backends, and running on a single large Power 775 system. However, X10 continues to be available and actively developed on a wide variety of platforms that span from laptops, to clusters, to supercomputers; Linux, AIX, Windows, and MacOS; x86 and Power.

1.1 Why UTS?

The workload in any of the four HPC Class II Challenge benchmarks can be partitioned statically across a distributed system effectively. For this submission, we wanted to consider an important class of problems for which such a static partitioning is not typically feasible: *state-space search* problems, such as the Unbalanced Tree Search benchmark.

The overall task of a state-space search problem is to calculate some metric over the set of all good configurations – e.g. the number of such configurations, the "best" configuration etc.

State-space search problems typically satisfy a few properties:

- Each configuration can be compactly represented.
- Usually, some non-trivial amount of computation is necessary to determine the next configurations from a good configuration.
- The number of configurations generated from a good configuration may be very difficult to predict statically.

The challenge at hand, therefore, is to parallelize the computation across a potentially very large number of hosts while achieving high parallel efficiency. The computation is typically initiated at a single host with the *root* configuration. A good solution must look to quickly divide up the work across all available hosts. It must ensure that once a host runs out of work it is able to quickly find work, if in fact work exists at any host in the system. That is, a good solution must solve a *global load-balancing* problem.

While we could have chosen other graph traversal kernels for this submission, in our experience, widely used kernels such as

SSCA2, while irregular, are much more balanced than UTS. In practice, SSCA2 can still be implemented with reasonable efficiency even on large scale systems by means of static partitioning – we have – provided the input data is randomly shuffled prior to distribution.

Outline The next two sections briefly review the core concepts of the X10 programming model and describe the key innovations and extensions necessary to allow them to perform at scale. After describing the hardware and software configuration in Section 4, we discuss the implementation and performance results for each kernel in turn: Global HPL, Global RandomAccess, Global FFT, EP Stream, and UTS. Section 10 concludes by summarizing the lines of code for each benchmark and key relative and absolute performance metrics.

2 Programming Model

We begin by briefly reviewing the core Asynchronous Partitioned Global Address Space (APGAS) programming model [6] that is at the heart of the X10 language design.

To support concurrency and distribution, X10 introduces a few key constructs. 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*. 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`. Concurrency control is provided by the statement `when (c) S` which (when started in a state in which `c` is true) can in a single uninterrupted step execute `S`. An optimized unconditional form of `when`, `atomic S`, is also provided. Finally, the last (and most important) control construct of X10 is `finish S`: it executes `S` and waits for all activities spawned during the execution of `S` to terminate before continuing.

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

The simplicity of these fundamental concurrency and distribution constructs is borne out by the fact that formal semantics have been developed [8, 4].

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 messaging style computation, bulk synchronous parallel computation, overlap between computation and communication, global view programming etc.

3 Idioms for Performance at Scale

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

3.1 Scaling Finish

In general, implementing `finish` requires a distributed termination protocol that can handle arbitrary patterns of distributed task creation and termination. The X10 language places no restrictions on the ability of the programmer to combine and nest `at` and `async` statements within a `finish`. The X10 runtime dynamically optimizes `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 controlled by the `finish` executes an `at`. Furthermore, the runtime automatically optimizes distributed finishes by applying message coalescing and compression to reduce the number and size of the control messages used by the termination protocol.

In addition to these general dynamic optimizations, the runtime provides implementations of distributed `finish` that are specialized to commonly occurring patterns of distributed concurrency that admit more efficient implementations. In our current system, opportunities to apply these specialized implementations are guided by programmer supplied annotations. For example,

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

Thanks to this information the runtime system will implement the distributed termination detection more efficiently.

Currently, the runtime system supports five finish pragmas:

FINISH_ASYNC A finish for a unique async possibly remote.

FINISH_LOCAL A finish with no remote activity.

FINISH_SPMD A finish with no nested remote activities in remote activities. The remote activities must wrap nested remote activities if any in nested finish blocks.

FINISH_HERE A finish which does not monitor activity starting or finishing in remote places. Useful for instance in a ping pong scenario where a remote activity is first created whose last action is to fork back an activity at the place of origin. The runtime will simply match the creation of the "ping" activity with the termination of the "pong" activity, ignoring both the termination of "ping" and creation of "pong" at the remote place.

FINISH_DENSE A scalable finish implementation for large place counts using indirect routes for control messages so as to reduce network contention at the expense of latency.

We have prototyped a fully automatic compiler analysis that is capable of detecting some of the situations where these patterns are applicable. However, it is not currently included in the distributed version of X10. As future work, we intend to further extend and "harden" this analysis to the point where it can be robustly applied as part of the X10 compiler's standard optimization package.

3.2 Optimizing Communication and Collective Operations

Enabling RDMA and Asynchronous Copy RDMA (Remote Direct Memory Access) hardware, such as InfiniBand, enables the transfer of segments of memory from one machine to another without the involvement of the CPU or operating system. This technology significantly reduces latency of data transfers, and frees the CPU to do other work while the transfer is taking place. To use RDMA hardware, the application needs to register the memory segments eligible for transfer with the network hardware, and issue transfer requests as background

tasks with a completion handler to signal when the transfer is complete. In X10, the main mechanism to do this is via the `Array.asyncCopy()` method, which performs these operations for you, if RDMA hardware is available. From the perspective of the programming model, an `asyncCopy` is treated exactly as if it were an `async`; just like `async` its termination is tracked by its dynamically enclosing `finish`.

Customized Memory Allocation The RDMA data transfers take place from a memory segment of one system to a segment at a (usually) remote system. When a data transfer is initiated, the caller needs to know the address of the memory segments both locally and remotely. The local pointer is easy, but the remote pointer must be determined. In a simple program, this usually involves some form of pre-RDMA messaging to get the remote pointer to the initiator of the RDMA call. There are many improvements that can be made on this. Within X10, we use a congruent memory allocator, which allocates and registers a memory buffer at the same address in every place (via `mmap()` or `shmget()`). This eliminates the need for the remote-pointer transfer, any form of remote pointer lookup table, or the need to calculate remote addresses at runtime.

Teams and other hardware optimizations X10 teams offer capabilities similar to HPC collectives, such as Barrier, All-Reduce, Broadcast, All-To-All, etc. Some networks support these well-known communication patterns in hardware, and some simple calculations on the data is supported as well. When the X10 runtime is configured for these systems, the X10 team operations will map directly to the hardware implementations available, offering performance that can not be matched with simple point-to-point messaging.

4 Hardware and Software Platform

We gathered performance results for our submission on a Power7-based Power 775 supercomputer. The smallest building block of the machine is called an octant. 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. A single octant has a peak performance of 982 GFLOPS; 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. The next logical building block of the system is a supernode (32 octants, 1024 cores). A supernode is built from 4 drawers, each drawer contains 8 octants. The full machine we used for our measurements contains 48 supernodes, with 1470 octants (47,040 cores) available for computation.[1] This gives the system a theoretical peak of 1443 TFLOPS.

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[2] version 2.2.3 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.

We executed all the programs in a mode in which each X10 place contained a single worker thread 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 level scheduling, the worker thread in each place was bound to a specific core.

[1] 66 octants are used either as service nodes or are reserved capacity to support the "fail-in-place" capabilities of the system.

[2] Native X10 compiles X10 to C++.

5 Global HPL

The Global HPL benchmark measures the floating point rate of execution for solving a linear system of equations. Performance is measured in Gflops.

5.1 Implementation

Our SPMD-style implementation of the benchmark mimics the main attributes of the reference HPL implementation. It 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, and uses default PAMI collectives.

For the local, sequential computations, our implementation makes use of IBM's ESSL library. It provides a BLAS-like API as a C-style header file and library. ESSL functions are imported in X10 as static methods by means of `@NativeCPPExtern` method annotations on `native` method declarations. The X10 compiler implements the mapping from X10 array to C-style arrays (wrapping/unwrapping).

```
@NativeCPPExtern native static def blockTriSolve(
  me:Rail[Double], diag:Rail[Double], B:Int):void;
```

For network communications, our implementation uses a collection of idioms: teams for barrier, row and column broadcast, or pivot search, array asynchronous copies for row fetch or swap. Per-column and per-row teams are obtained through the `split` method of the `Team` class.

```
colRole = here.id % px;
rowRole = here.id / px;
col = Team.WORLD.split(here.id, rowRole, colRole);
row = Team.WORLD.split(here.id, colRole, rowRole);
```

On Power 775, team operations are backed by hardware-accelerated collective communication primitives and array asynchronous copies by RDMAs. But the same X10 code would run on a distinct system irrespective of the availability of such hardware-accelerated mechanisms.

Our implementation also takes advantage of finish pragmas for instance to make sure the compiler and runtime recognize that a row swap is a simple ping-pong pattern.

5.2 Performance

Places are mapped to hosts in groups of 32. We use about 70% of the memory of each host and a block size of 360. We run with up to 32,768 places in power-of-two increments. With 32,768 places, we measure an aggregated performance of about 589 Tflops, that is, about 60% of the theoretical peak of 1024

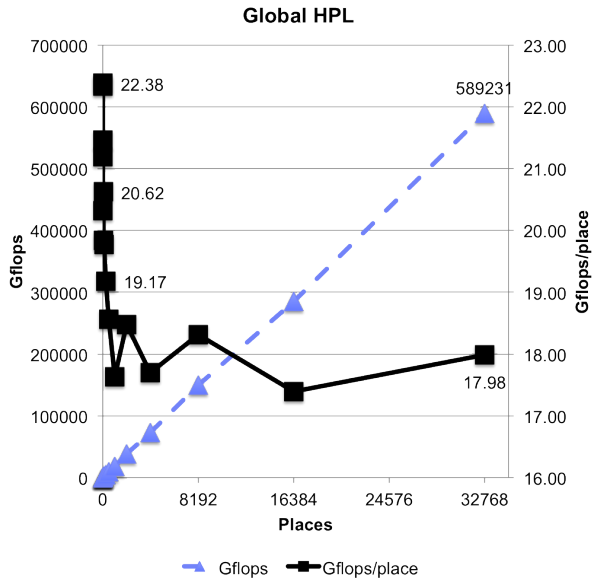

Figure 1: X10 Performance for Global HPL

hosts, which amounts to about 70% of the effective ESSL peek performance (DGEMM).

The per-place performance at scale is 17.98 Gflops/core. The single place performance is 22.38 Gflops. In other words, the code achieves an efficiency of 80% at scale.

Figure 1 plots both the aggregated and per-place performance. We can see that the efficiency drop primarily occurs 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 resp. even and odd powers of two.

6 Global RandomAccess

The Global RandomAccess benchmark is designed to measure the system 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 GUPS (Giga Updates Per Second) the system can sustain.

6.1 Implementation

Our implementation takes advantage of congruent memory allocation, allocating the per-place table fragment at the same virtual address in each place. The memory updates are issued using the following statement where `place_id`, `index`, and `update` are randomly generated.

```
imc.getCongruentSibling(Place(place_id))
   .remoteXor(index, update);
```

First the `getCongruentSibling` method return a handle for the local table at the destination place. In contrast with typical remote references in X10 (e.g., GlobalRef[T] or RemoteArray[T]), this handle is not abstract—it does not need to be translated into an address at the destination—but provides the virtual address of the remote table. The `remoteXor` method issues the update. On Power 775, it makes use of the remote update capability of the Torrent chip, to perform the XOR operations directly in the network and memory subsystems of the remote host. The remote CPU is not involved. The `remoteXor` method is not Power-775-specific however, the same code would run fine on commodity hardware (for instance using our TCP/IP transport implementation). In the latter, the update will be handled by the remote CPU.

The timed portion of the code measures the cost of issuing the updates. It does not include the time it takes to complete the remote updates. The hardware is designed to maximize GUPS and does not provide means to track the completion of individual updates. The difference would be negligible anyway as the number of "in flight" updates is many order of magnitude less than the number of issued updates.

6.2 Performance

Places are mapped to hosts in groups of 32. 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. The table must be backed with huge pages to avoid costly TLB misses (when the Torrent processes the updates). We run with up to 32,768 cores. At scale, we measure 843 GUPS.

Figure 2 plots two performance curves. The bisection bandwidth on Power 775 depends on the placement of hosts w.r.t. supernodes. For the "sparse hostfile" curve, we distribute the hosts across the entire system so as to maximize bisection bandwidth. In essence, we use many supernodes, but not all octants in each supernode. In this configuration, the bottleneck is always the per-host bandwidth so the scaling is linear. The 843 GUPS we measure at scale in actually slightly better than 1,024 times the 0.8 GUPS measured for a single host.

For completeness, we also plot a "dense hostfile" curve with fully populated supernodes. Here, the bisection bandwidth is the bottleneck, scaling is therefore super linear.

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+PAMI implementation of the benchmark.

7 Global FFT

Global FFT performs a Discrete Fourier Transform on a one-dimensional array of double-precision complex values. The source and destination arrays are evenly distributed across the system. It stresses the floating point units, network, and memory subsystems. Performance is measured in Gflops.

7.1 Implementation

Our implementation follows from the reference implementation. It 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. For the per-row FFT—a single-place, sequential computation—we simply defer to the reference C code. The global transposition is implemented with local data shuffling followed by an all-to-all collective followed by another round of local data shuffling.

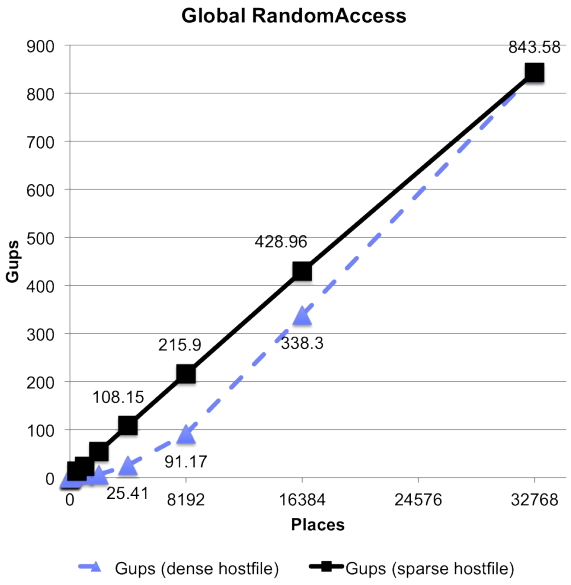

Figure 2: X10 Performance for RandomAccess

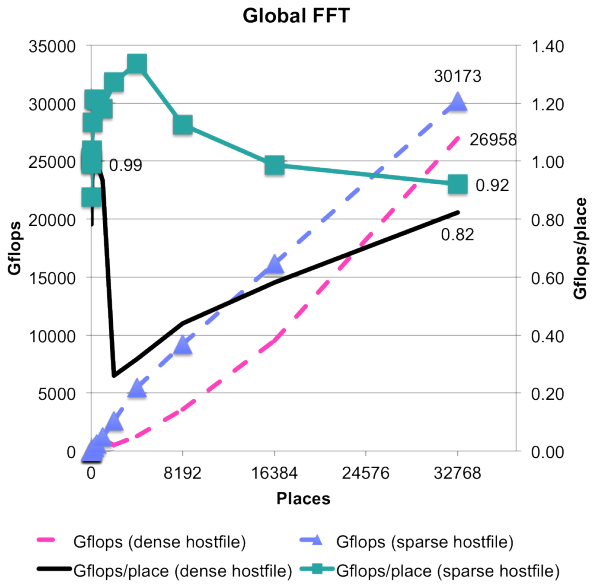

Figure 3: X10 Performance for Global FFT

To maximize the performance of the All-To-All collective, we use the congruent memory allocator to ensure that all per-place array fragments (either source or destination) are allocated at the same virtual address and registered for RDMA operations.

Depending on the system, an All-To-All collective may benefit from an initial warmup operation, to initialize any data structures or other components of the network before the main calculation. So we include a warmup in our program as well.

7.2 Performance

We run with up to 32,768 places in power-of-two increments. For even powers of two, each place uses 2 GB of memory, for odd powers of two only 1 GB is used, hence either half or a quarter of the memory of the host.

Because this code is very dependent on bisection bandwidth and bisection bandwidth on Power 775 depends on the placement of hosts w.r.t. supernodes, we plot two sets of performance curves in Figure 3. The "sparse hostfile" set of curves shows performance results if the places are spread across supernodes to maximize bandwidth. The "dense hostfile" curves confirm that the performance drops significantly for medium size problems if the places are packed in a small number of supernodes. In each case, we show the total floating point rate of execution as well as the per-place rate.

With one place, we measure 0.99 Gflops. At scale the rate reduces to 0.92 Gflops with a sparse hostfile and 0.82 Gflops with a dense hostfile. In between, the per-place rate fluctuates between 0.26 Gflops and 1.34 Gflops depending on the number and placement of the hosts.

8 EP Stream

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

8.1 Implementation

The implementation of this benchmark in X10 follows a straightforward SPMD style of programming. The main activity launches an activity at every place using a broadcast method of X10's standard library (more efficient than a sequential for loop over all places). 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.

The critical loop is written:

```
for(var i:Int=0; i<localSize; i++) a(i)=b(i)+beta*c(i);
```

It is followed with a global barrier.

```
Team.WORLD.barrier(here.id);
```

The execution time, including the barrier time, is measured at place 0.

8.2 Performance

We run with from 1 place to 32,768 places in power-of-two increments and also at scale with 47,040 places. Below 32 places we map all places to a single host. Above 32 places, we map 32 places per host. We use 1.5 GB of memory per place, that is, 48GB per host. Figure 4 shows the aggregated memory bandwidth achieved by the system (in GB/s) as well as the per-place memory bandwidth (GB/s/host).

The per-place memory bandwidth decreases as the number of places per host increases. It drops from about 12.6 GB/s with one place to about 7.2 GB/s with 32 places due to the QCM architecture. Hence, the per-place memory bandwidth plot is first a vertical line. 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, hence the horizontal line. The total system bandwidth at 47,040 places (1470 hosts) is about 335 TB/s,

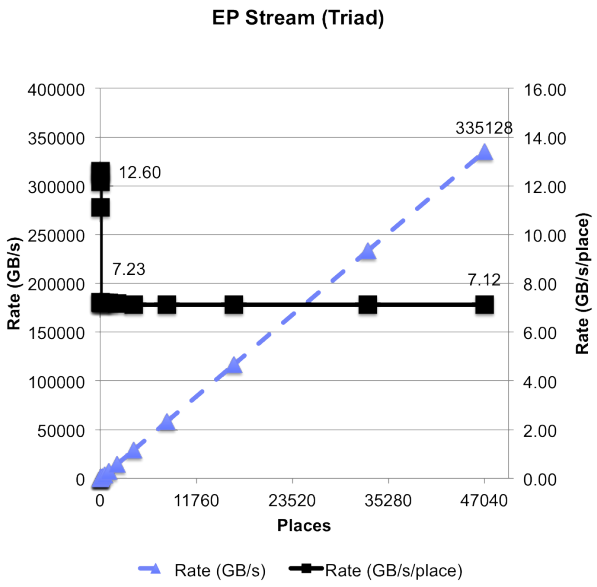

Figure 4: X10 Performance for EP Stream

which represents 98% of 1470 times the single-host bandwidth. We attribute the 2%-loss to jitter and synchronization overheads.[3]

9 UTS

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 submission, we used a *fixed geometric law* for the random number generator with branching factor $b_0 = 4$, seed $r = 19$, and tree depth d varying from 14 with 1 place to 22 at scale with 47,040 places.

> 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, 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.

The depth cut-off makes it possible to caliber the trees and shoot for a target execution time. To be fair, the distance from the root to a particular node is never used in our benchmark implementation to predict the size of a subtree. In other words, all nodes are treated equally, irrespective of the current depth.

9.1 Design

The material in this section is excerpted from [9].

One common way to load balance is to use *work-stealing*. For shared memory system this technique has been pioneered in the Cilk system. Each worker maintains a double ended queue (deque) of tasks. When a worker encounters a new task, it pushes its continuation onto the bottom of the deque, and descends into the task. On completion of the task, it checks to see if its deque has any work, if so it pops a task (from the bottom) and continues with the work. Else (its deque is empty) it looks for work on other workers' deques. It randomly determines a *victim* worker, checks if its queue is non-empty, and if so, pops a task from the top of the dequeue (the end other than the one being operated on by the owner of the dequeue). If the queue is empty, it guesses again and continues until it finds work. Work-stealing systems are optimized for the case in which the number of steals is much smaller than the total amount of work. This is called the "work first" principle – the work of performing a steal is incurred by the thief (which has no work), and not the victim (which is busy performing its work).

Distributed work-stealing must deal with some additional complications. First, the cost of stealing is usually significantly higher than in the shared memory case. For many systems, the target CPU must be involved in processing a steal attempt. Additionally, one must solve the distributed termination detection problem. The system must detect when all workers have finished their work, and there is no more work. At this point all workers should be shut down, and the computation should terminate.

X10 already offers a mechanism for distributed termination detection – `finish`. Thus, in principle it should be possible to spawn an activity at each place, and let each look for work. To trigger `finish` based termination detection however, these workers must eventually terminate. But when? One simple idea is that a worker should terminate after k steal attempts have failed. However this leaves open the possibility that a worker may terminate too early – just because it happened to be unlucky in its first k guesses. If there is work somewhere else in the network then this work can no longer be shared with these terminated workers, thereby affecting parallel efficiency. (In an extreme case this could lead to sequentializing significant portion of the work.)

Therefore there must be a way by which a new activity can be launched at a place whose worker has already terminated. This leads to the idea of a *lifeline graph*. For each place p we predetermine a set of other places, called *buddies*. Once the worker at p has performed k successive (unsuccessful) steals, it examines its buddies in turn. At each buddy it determines whether there is some work, and if so, steals a portion. However, if there is no work, it records at the buddy that p is waiting for work. If p cannot find any work after it has examined all its buddies, it dies – the place p now becomes quiescent.

But if P went to a buddy B, and B did not have any work, then it must itself be out looking for work – hence it is possible that it will soon acquire work. In this case we require that B *distribute* a portion of the work to those workers that attempted to buddy steal from it but failed. Work must be spawned on its own `async` – using the `at (p) async S` idiom. If P had no activity left, it now will have a fresh activity. Thus, unlike pure work-stealing based systems, a system with lifeline graphs will see its nodes moving from a state of processing work (the active state), to a state in which they are stealing to a state in which they are dead, to a state (optionally) in which they are woken up again with more work (and are hence back in the active state).

Note that when there is no work in the system, all places will be dead, there will be no active `async` in the system and hence the top-level `finish` can terminate.

The only question left is to determine the assignment of buddies to a place. We are looking for a directed graph that is fully

[3] Each host runs a full Linux image.

Program	X10	Native (C/C++)
G-HPL	588	120
G-RandomAccess	143	0
G-FFT	213	1117
EP Stream (Triad)	60	0
UTS	493	637

Table 1: Lines of Code (non-blank, non-comment)

connected (so work can flow from any vertex to any other vertex) and that has a low diameter (so latency for work distribution is low) and has a low degree (so the number of buddies potentially sendng work to a dead vertex is low). z-dimensional hyper-cubes satisfy these properties and have been implemented.

9.2 Implementation

Our implementation follows from [9] using random steals followed by lifeline steals and hyper-cubes for the lifeline graphs.

We improve the scalability beyond [9] by further reducing the overhead of termination detection. First, our implementation takes advantage of the `FINISH_DENSE` pragma which minimizes network contention. Second, we annotate the asyncs associated with attempted steals with `@Uncounted`. It dispenses the runtime from explicitly keeping track of the completion of these asyncs. This is fine since a thief always waits for the response from the victim (positive or not). Therefore, the only counted asyncs in our implementation are those distributing work (i) initially, or (ii) later along a lifeline.

We also adopt a more compact representation of the "active" nodes in a place, by directly representing intervals of siblings as intervals (lower, upper bounds) instead of using an expanded list.

Finally to counteract the bias introduced by the depth cut off, a thief steals fragments of every active interval of a place. There are few of them since we traverse the tree deep first.

9.3 Performance

The performance achieved by the X10 implementation of the UTS benchmark on the PERCS prototype is shown in Figure 5 for a number of places varying from one to 32,768 in power-of-two increments and then at scale with 47,040 places. It shows the total total number of nodes processed per second as well as the per-place processing rate.

The depth of the tree progressively increases from 14 with one place to 22 with 47,040 places so as to permit runs within 1 to 5 minutes.

The per-place processing rate varies from 10.9 million nodes per second for a single place to 10.75 million nodes per second at scale. The efficiency never drops below 98%.

At scale with 47,040 places, we traverse a tree of 69,312,400,211,958 nodes in 137s, that is about 505 billion nodes/s. As part of this traversal, we compute 17,328,102,175,815 SHA1 hashes (for the random number generator).

10 Summary

Table 1 reports the lines of X10 and native code for each of the five programs. For X10, the table reports the number of non-

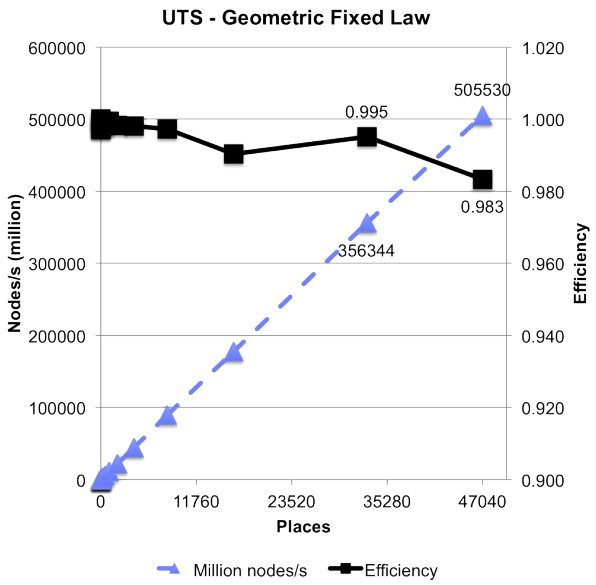

Figure 5: X10 Performance for UTS

Program	Places	Perf. at scale	Efficiency
G-HPL	32,768	589 Tflops	80%
G-RandomAccess	32,768	843 Gups	100%
G-FFT	32,768	30 Tflops	93%
EP Stream (Triad)	47,040	335 TB/s	98%
UTS	47,040	505 B edges/s	98%

Table 2: Performance

blank, non-comment lines. For the native code the table reports lines as counted by David A. Wheeler's 'SLOCCount' utility.

In all three programs that use native code, the role of the native code is to provide computationally intensive sequential kernels. In the case of HPL, the native code wraps the IBM ESSL libraries which provide `DGEMM`, etc. For FFT, the native code consists of kernels taken from the FFTE package. For UTS, the native code implements the SHA1 hash function.

In Table 2, we summarize our performance results for the five kernels. As explained in Section 4, we run with single-threaded places, binding each place to a specific core of the system. Each host provides 32 cores. There are 1,470 hosts available for a maximum of 47,040 cores.

For G-HPL and UTS the efficiency at scale is computed as:

$$\frac{\text{performance at scale}}{\text{number of places} * \text{single-place performance}}$$

The remaining three kernels are much more sensitive to memory bandwidth, which on Power 775 decreases as the number of places on a host increases. So we report the efficiency computed as:

$$\frac{\text{performance at scale}}{\text{number of hosts} * \text{single-host performance}}$$

Our X10 implementations of G-HPL, EP Stream (Triad), and G-RandomAccess implementations deliver between 75% and 85% of the performance of IBM's HPC Class I Challenge implementations of these benchmarks. In other words, we achieve at

least 3/4 of the system's potential for these benchmarks. Our G-FFT implementation is more primitive and delivers only about half of the best number obtained on the system.

To the best of our knowledge, no other UTS implementation is capable of such scaling on this hardware or any other. While our algorithmic insights may now be transposed to other programming languages or back-ported to the reference 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 enabling us to discover the keys to performance at scale.

Acknowledgments

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

Our work on X10 has been greatly enriched by collaboration and thoughtful feedback on the language and its implementation from innumerable collaborators both within IBM and in the external X10 research and open source communities. Thank You!

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

References

[1] B. Bloom, B. Herta, D. Grove, A. Sabharwal, H. Samulowitz, and V. Saraswat. Scalable Plug & Play Parallel SAT Solver using X10. Submitted to SAT 2012, 2012.

[2] M. Dayarathna, C. Houngkaew, and T. Suzumura. Introducing scalegraph: an x10 library for billion scale graph analytics. In *Proceedings of the 2012 ACM SIGPLAN X10 Workshop*, X10 '12, pages 6:1–6:9, New York, NY, USA, 2012. ACM.

[3] 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.

[4] 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.

[5] J. Milthorpe, V. Ganesh, A. P. Rendell, and D. Grove. X10 as a parallel language for scientific computation: practice and experience. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium*, IPDPS '11. IEEE Computer Society, May 2011.

[6] 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.

[7] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. The X10 reference manual, v2.2. June 2011.

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

[9] 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, New York, NY, USA, 2011. ACM.

[10] A. Shinnar, D. Cunningham, B. Herta, and V. Saraswat. M3R: Increased performance for in-memory Hadoop jobs. In *Proceedings of VLDB Conference*, VLDB '12, 2012.

[11] T. Suzumura, S. Kato, T. Imamichi, M. Takeuchi, H. Kanezashi, T. Ide, and T. Onodera. X10-based massive parallel large-scale traffic flow simulation. In *Proceedings of the 2012 ACM SIGPLAN X10 Workshop*, X10 '12, pages 3:1–3:4, New York, NY, USA, 2012. ACM.

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

[13] X10 Global Matrix Library. `https://x10.svn.sourceforge.net/svnroot/x10/trunk/x10.gml`, Oct. 2011.