X10: an Experimental Language for High Productivity Programming of Scalable Systems
(Extended Abstract)

Kemal Ebcioğlu Vijay Saraswat Vivek Sarkar

IBM Research
T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598

{kemal,vsaraswa,vsarkar}@us.ibm.com

Abstract

It is well established that application development productivity is a significant bottleneck in the time to solution for obtaining production applications on High-End Computing (HEC) systems. Previously, we introduced a simple model for defining application development productivity in the presence of multiple expertise levels, and used this model to motivate the programming model and tools solution being pursued in the IBM PERCS project [9]. In this paper, we describe X10*, an experimental language that embodies a new parallel programming model serves as the foundation for multiple productivity-improving technologies in PERCS ranging from visualization and refactoring tools to static and dynamic optimizing compilers.*

1 Introduction and Motivation

The key challenges faced by current and future-generation large-scale systems are 1) *Scalability:* the ability to effectively utilize multiple levels of available parallelism in a high system, such as clusters, SMPs, multiple cores on a chip, co-processors, SMT, and SIMD levels, and 2) *Non-uniform data access:* the ability to support a global data model in the presence of severe nonuniformities in latency, bandwidth and interfaces for accessing data in different parts of the system. It is now common wisdom that the ongoing increase in hardware complexity of large-scale parallel systems to address these challenges has been accompanied by a *decrease in software productivity* for developing, debugging, and maintaining applications for such machines [12]. This is a serious problem because current trends for next generation systems, including SMP-on-a-chip and tightly coupled "blade" servers, indicate that these complexities will be faced not just by programmers for large-scale parallel systems, but also by mainstream application developers.

In the area of scientific computing, the programming languages community responded to these challenges with the design of several programming languages, including Sisal, Fortran 90, High Performance Fortran, Kali, ZPL, UPC, Co-Array Fortran, and Titanium. The ultimate challenge facing this community is supporting *high-productivity, high-performance programming*: that is, designing a programming model that is simple and widely usable (so that hundreds of thousands of application programmers and scientists can write code with felicity) and yet efficiently implementable on current and proposed architectures without requiring "heroic" compilation efforts. This is a grand challenge, and past languages, while taking significant steps forward, have fallen short of this goal either in the breadth of applications that can be supported or in the ability to deliver the underlying performance of the target machine. MPI still remains the most common model used in obtaining high performance on large-scale systems, despite the productivity limitations inherent in its use.

During the same period, significant experience has also been gained with the widespread adoption of object-oriented languages, such as JAVA and C#, that are executed on *virtual machines* and *managed runtime environments*. These languages, along with their accompanying libraries, frameworks and tools, have enjoyed much success in improving *productivity* for commercial applications.

X10 is an experimental new object-oriented language for high performance computing that is currently under development at IBM in collaboration with academic partners.

The X10 effort is part of the IBM PERCS project (Productive Easy-to-use Reliable Computer Systems) whose goal is to design adaptable scalable systems for the 2010 timeframe. The PERCS technical agenda is focused on hardware-software co-design that combines advances in chip technology, computer architecture, operating systems, compilers, programming environments and programming language design. The main role of X10 is to simplify the programming model so as to increase the programming productivity for future systems like PERCS, without degrading performance. Combined with the PERCS Programming Tools agenda [9], the ultimate goal is to use a new programming model and a new set of tools to deliver a 10× improvement in development productivity for large-scale parallel applications by 2010.

To manage concurrency and distribution, X10 introduces constructs that are expected to be amenable to automatic static and dynamic optimizations by 2010. Specifically, X10 introduces *atomic sections* in lieu of locks, *clocks* in lieu of barriers, and *asynchronous operations* in lieu of threads. To increase performance transparency, X10 integrates new constructs (notably, *places*, *regions* and *distributions*) to model hierarchical parallelism and non-uniform data access.

X10 is a strongly typed language that emphasizes the static expression of program invariants (e.g. about locality of computation). Such static expression improves both programmer productivity (in documenting design invariants) and performance. The X10 type system supports generic type-abstraction (over value and reference types), is place- and clock-sensitive and guarantees the absence of deadlock (for programs without conditional atomic sections), even in the presence of multiple clocks. X10 specifies a rigorous, clean and simple semantics for programming constructs independently from a specific implementation.

In the remainder of this extended abstract, we present an overview of the X10 design, and use example programs to illustrate some of the individual features in X10.

2 The X10 language design

This section provides a brief summary of the X10 language, focusing on the core features that are most relevant to locality and parallelism. A number of other features in X10 are not mentioned here due to space limitations. These include generic interfaces, generic classes, type parameters, sub-distributions, array constructors, exceptions, place casts and the nullable type constructor.

A central concept in X10 is that of a *place*. A place is a collection of resident light-weight threads (called *activities*) and *data*, and is intended to map to a data-coherent unit in a large scale system such as an SMP node or a single co-processor. It contains a bounded, though perhaps dynamically varying, number of *activities* and a bounded amount of storage. Cluster-level parallelism can be exploited in an X10 program by creating multiple places.

There are four storage classes in an X10 program:

1. *Activity-local* — this storage class is private to the activity, and is located in the place where the activity executes. The activity's stack and thread-local data are allocated in this storage class.

2. *Place-local* — this storage class is local to a place, but can be accessed coherently by all activities executing in the same place.

3. *Partitioned-global* — this storage class represents a *unified* or *global address space*. Each element in this storage class has a unique place that serves as its home location, *references* to the element can be manipulated by both local activities (activities in the same place as the element) and remote activities (activities in a different place from the element). However, as discussed below, *accesses* to the element can only be performed by local activities.

4. *Values* — Instances of *value classes* (value objects), are immutable and stateless in X10, following the example of Kava[7]. Such value objects are in this storage class. Since value objects do not contain any updatable locations, they can be freely copied from place to place. (The choice of when to clone, cache or share value objects is left to the implementation.) In addition, methods may be invoked on such an object from any place.

X10 activities operate on two kinds of *data objects*. A *scalar* object has a small, statically fixed set of fields, each of which has a distinct name. The mutable state of a scalar object is located at a single place. An *aggregate* (array) object has many elements (the number may be known only when the object is created), uniformly accessed through an index (e.g. an integer) and may be distributed across many places. Specifically, an X10 array specifies 1) a set of indices (called a *region*) for which the array has values, 2) a *distribution mapping* from indices in this region to places, and 3) the usual array mapping from each index in this region to a value of the given base type (which may itself be an array type). Operations are provided to construct regions (distributions) from other regions (distributions), and to iterate over regions (distributions). These operations include standard set-based operations such as unions and intersections, some of which are available in modern languages such as ZPL [10]. It is also worth noting that commonly-used basic types such as `int`, `float`, `complex` and `string` are defined as *value classes* in the x10.lang standard library, rather than as primitive types in the language.

Activities represent lightweight threads in X10. An activity is created in a given place and remains in that place for its lifetime, but each place may have several activities executing in parallel. An activity can recursively spawn additional activities at places of its choosing. Throughout its lifetime an activity executes at the same place, and has direct access only to data stored at that place. Remote data can only be accessed by spawning asynchronous activities at the places at which data is resident. Any attempt by an activity to directly access a non-local datum is made manifest either as a type-checking error during compilation or as a BadPlaceException during execution.

The X10 type system is used to catch many common cases

Asynchronous activities have two forms — statements and expressions. The expression form of an asynchronous activity is called a *future*, and is discussed further below. The statement form of an asynchronous activity is async (P) S where S is a statement and P is a place expression. Such a statement is executed by spawning an activity at the place designated by P to execute statement S. As a convenient means of identifying the place of a datum in the partitioned-global storage class, when the expression P specifies an array element or object, it evaluates to the place containing that array element or object. (P) can also be omitted, in which case, it is inferred to be the place of the data accessed by statement S (provided that a single place can be unambiguously inferred).

For example, the X10 statement,

```
async (A[99]) { A[99] = k }
```

creates a new activity at the place containing element A[99] of a global distributed array A. The values of local variables such as k are passed as implicit parameters to this activity. We believe that the use of implicit parameters aids in productivity, since it relieves the programmer of the burden of encapsulating remote activities as procedure calls with explicit parameters. As an additional productivity aid, X10 also supports an *implicit syntax* for async statements and other constructs e.g., the above example could simply be written as A[99] = k;, which denotes the same asynchronous activity to be executed at the place containing A[99]. This example illustrates how an async statement can be used to accomplish a remote store operation. However, async statements can be used as the foundation for many other common programming idioms in HEC application development including fine-grained threads, asynchronous DMA operations, message send (for an active or passive message), and scatter operations.

In addition to the *async* statement, the *foreach* construct serves as a convenient mechanism for spawning local activities across a specified index set (*region*) and the *ateach* (pronounced "at each") construct serves as a convenient mechanism for spawning activities across a set of local/remote places or objects.

X10 provides five mechanisms for the coordination of activities — *clocks*, *force operations*, *finish operations*, *atomic sections*, and *conditional atomic sections* — which are summarized below in the following paragraphs.

Clocks Clocks are a generalization of barriers, which have been used as a basic synchronization primitive for MPI process groups and in other SPMD programming models. X10 *clocks* are designed to offer the functionality of multiple barriers in the context of dynamic, asynchronous, hierarchical networks of activities, while still supporting determinate, deadlock-free parallel computation.

A clock is defined as a special value class instance, on which only a restricted set of operations can be performed. At any given time an activity is *registered* with zero or more clocks. The activity that creates a clock, is automatically registered with this clock. An activity may register other activities with a clock, or may un-register itself with a clock. At any given step of the execution a clock is in a given *phase*. The first phase of the clock starts when the clock is created. The clock *advances* to its next phase only when all its currently registered activities have quiesced (either by performing a next operation, or by terminating), and all statements scheduled for execution in the current phase have terminated. In this manner, clocks serve as a generalization of *barriers* for a dynamically varying collection of activities. From an activity's viewpoint, when it performs a next operation, it quiesces on *all* the clocks it is registered with, and suspends until all of them have advanced to their next phase.

Force Operations When an activity A executes the statement, F = future (P) E, it asynchronously spawns an activity B at the place designed by P to evaluate the expression E. Execution of the expression in A terminates immediately, yielding a *future* [4] in F, thereby enabling A to perform other computations in parallel with the evaluation of E. A may also choose to make the future stored in F accessible to other activities. When any activity wishes to examine the value of the expression E, it invokes a force operation on F. This operation blocks until B has completed the evaluation of E, and returns with the value thus computed. Like async statements, future's can be used as the foundation for many other common programming idioms in HEC application development including fine-grained threads, asynchronous DMA operations, message send receive, and gather operations.

Finish Operations When an activity A executes the statement, `finish S`, where `S` is a statement, it is guaranteed that the finish statement will not be completed till all activities that are (recursively) spawned by `S` have terminated. Therefore, `finish` is a convenient operation that can be used to enforce global termination.

Unconditional Atomic Sections A statement block or method that is qualified as `atomic` has the semantics of being executed by an activity as if in a single step, during which all other activities are frozen[1]. Thus, atomic sections may be thought of as executing in some global sequential order, even though this order is indeterminate. An atomic section is a generalization of user-controlled locking, so that the `X10` programmer only needs to specify that a collection of statements should execute atomically and can leave the responsibility of lock management and other mechanisms for enforcing atomicity to the language implementation. Primitives such as fetch-and-add, updates to histogram tables, updates to a bucket in a hash table, airline seat reservations arriving at an online data base, and many others, are a natural fit for coordination using atomic sections. `X10` also requires that each access to shared mutable data (*i.e.*, mutable data that can be accessed by multiple activities) must occur in an atomic section, thereby easing the constraints imposed by the memory consistency model.

Consider the following atomic section as a concrete example:

```
atomic { node = new Node(data, head);
         node.next = head; head = node; }
```

By declaring the statement block as atomic, the programmer is able to maintain the integrity of a linked list data structure in a multithreaded context, while still giving the `X10` system the flexibility of using fine-grained synchronization or even non-blocking implementations.

From a scalability viewpoint, it is important to avoid including long-running or blocking operations in an atomic section. In addition, we call an atomic section *analyzable* if the locations and places of all data to be accessed in the atomic section can be computed on entry to the atomic section. Analyzability of atomic sections is not a language requirement, but serves as an important special case for which optimized implementations of atomic sections can be developed [8].

Conditional atomic sections Conditional atomic sections in `X10` are akin to conditional critical regions [3], and have the form `when (c) S`. If the guard `c` is false in the current state, the activity executing the statement blocks until `c` becomes *true*. Otherwise, as far as any other concurrently executing activity is concerned, the statement is executed *in a single step* which begins with the evaluation of `c` = *true*, and ends with the completion of statement `S`. This implies that `c` is not allowed to change between the time it is detected to be true and the time `S` begins execution. `X10` currently does not permit the statement `S` to contain or invoke a nested conditional atomic section.

A conditional atomic section for which the condition `c` is statically true is considered to be equivalent to an unconditional atomic section.

[1]The implementation may of course allow concurrent execution of atomic sections, using techniques such as non-blocking algorithms and optimistic concurrency, as long as atomic sections are made to appear to execute in a "single step" to the rest of the program.

3 RandomAccess Example

Figure 1 outlines one possible implementation for the RandomAccess HPC Challenge benchmark in `X10`. The group of statements labeled (1) is used to allocate and initialize `table` as a global block-distributed array. Note the definitions of `region` *r* and `distribution` *d*, which provide the foundation for allocating the `table` array. Since the index variable used in the `ateach` construct has the same distribution as the `table` array, it is guaranteed that each access to `table[i]` will be performed by a local activity *i.e.*, by an activity located in the same place as `table[i]`. The use of the `finish` operator ensures that all initialization activities spawned in the `ateach` construct must be completed before execution moves to group (2).

Next, the group of statements labeled (2) is used to allocate and initialize `ranStarts` as a "unique-distributed" array *i.e.*, an array with exactly one element per place, and the group of statements labeled (3) is used to allocate and initialize a *value* array named `smallTable`.

The group of statements labeled (4) defines the core computational kernel of RandomAccess, with one activity per place that executes a long running sequential loop, (5). Each iteration of the loop performs an `async` statement on the place containing `table[j]` group of statements labeled (2) , and the async statement performs an atomic read-exor-write operation on `table[j]`.

Finally, the statement labeled (6) performs a sum reduction on `table[]`, and compares the sum value with an expected result.

4 Jacobi Example

Figure 2 outlines one possible implementation for the Jacobi example program in `X10`. The group of statements labeled (1) is used to create a block distribution, `D`, a second distribution, `D_inner`, that contains only the interior elements of `D`, and a third distribution, `D_boundary`, that

```
public boolean run() {
    // (1) Allocate and initialize table as a block-distributed array
    final region r = new region(0,TABLE_SIZE-1);
    final distribution d = distribution.block(r);
    ranNum[d] table = new ranNum[d];
    finish ateach(int i:d) {table[i]=new ranNum(i);}

    // (2) Allocate and initialize ranStarts as a unique-distributed array
    // with one random number seed for each place
    final distribution d2= distribution.unique(place.places);
    ranNum[d2] ranStarts = new ranNum[d2];
    finish ateach(int i:d2) {ranStarts[i]=new ranNum(...);}

    // (3) Allocate a small immutable table that can be copied on all processors
    // and is used in generating the update values
    final region r3=new region(0,SMALL_TABLE_SIZE-1);
    final place valuePlace=(1).place;
    final distribution d3=distribution.constant(r3,valuePlace);
    value ranNum[d3] smallTable = new ranNum[d3];
    foreach(int i:r3) {smallTable[i]=new ranNum(i*SMALL_TABLE_INIT);}

    // (4)In all places in parallel, repeatedly generate random table indices
    // and perform atomic read-modify-write operations on corresponding table elements
    finish ateach (point p : ranStarts.distribution) {
       long ran = nextRandom(ranStarts[p]);
       // (5) Sequential loop
       for (int count=1; count<=N_UPDATES_PER_PLACE; count++) {
           final int j = f(ran);
           final long k = smallTable[g(ran)];
           async(table.distribution[j]){atomic{table[j]^=k};}
           ran = nextRandom(ran);
       }
    }

    // (6) Return true iff sum of elements in table[] matches expected result
    return table.reduce(ranNum.add,0)==EXPECTED_RESULT;
}
```

Figure 1: RandomAccess example in X10

contains all the remaining elements. Next, the group of statements labeled (2) is used to allocate and initialize array `b`. Note the use of different initialization statements for the inner and boundary elements.

The statements in (3) creates a new array, `temp[]`, that is used to compute the new values of the interior elements of `b[]`. The *overlay* operator is used to merge in temp[] values into `b[]`.

Finally, the statement labeled (4) performs a sum reduction on `b[]`, and compares the sum value with an expected result.

5 Conclusion

`X10` is considerably higher-level than thread-based languages in that it supports dynamically spawning very lightweight activities, the use of atomic operations for mutual exclusion, and the use of clocks for repeated quiescence detection of a data-dependent set of activities. Yet it is much more concrete than languages like HPF in making explicit the distribution of data objects across places. In this, the language reflects the designers' belief that issues of locality and distribution cannot be hidden from the programmer of high-performance code in high-end computing. A performance model that distinguishes between computation and communication must be made explicit and transparent. At the same time we believe that the interaction between the concurrency constructs and the place-based type system (including first-class support for type parameters) will enable much of the burden of generating distribution-specific code and coordination of activities to be moved from the programmer to the underlying implementation.

In future work we plan to extend `X10` along two dimensions. First we plan to develop an *implicit syntax* which allows the programmer to elide certain details. The compiler will automatically fill-in these details based on type information. For instance, the programmer may specify an assignment `l = x` where `x` is not known to be local; the compiler may automatically introduce a `force/future` combination to read the remote value synchronously and store it in `l`. Several simplifications to the `X10` syntax are possible in this fashion.

Second we plan to develop mechanisms to support library developers writing place- and clock-generic code and their own high-level domain-specific abstractions. For instance, it should be possible for library developers to write code for hierarchically tiled arrays [2], and for distributed data-structures [6]. It should be possible for such developers to use the equivalent of `foreach/ateach` over their own distributed data-structures.

We plan to evaluate the effectiveness of the `X10` language by designing and running *productivity trials*. These trials will primarily be designed to evaluate the ease of developing new code in the HPC domain using `X10`. We plan to target developers in the HPC domain who are focused on developing performance-efficient library code, as well as developers interested in rapidly prototyping new applications (that must use high degrees of concurrency). Once a performance-efficient implementation of `X10` is available we also plan to evaluate the performance of `X10`, for a range of benchmark programs.

Acknowledgments

`X10` is being developed in the context of the IBM PERCS (Productive Easy-to-use Reliable Computing Systems) project, which is supported in part by DARPA under contract No. NBCH30390004. We are grateful to the following people for their feedback and contributions to the design and implementation of `X10`: David Bacon, Bob Blainey, Philippe Charles, Perry Cheng, Julian Dolby, Guang Gao, Christian Grothoff, Allan Kielstra, Robert O'Callahan, Filip Pizlo, Christoph von Praun, V.T. Rajan, Lawrence Rauchwerger, Mandana Vaziri, and Jan Vitek. We are also grateful to John McCalpin, Mootaz Elnozahy and Lisa Spainhower for helpful discussions.

References

[1] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley Professional.

[2] Gheorge Almasi and Luiz de Rose and Jose Moreira and David Padua. Programming for Locality and Parallelism with Hierarchically Tiled Arrays.

[3] C.A.R. Hoare. Monitors: An operating system structuring concept. *CACM*, 17(10):549–557, October 1974.

[4] R. Halstead. MULTILISP: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7:501–538, 1985.

[5] Doug Lea. The Concurreny Utilities, 2001. JSR 166, http://www.jcp.org/en/jsr/detail?id=166.

[6] Steven Saunders and Lawrence Rauchwerger. Armi: an adaptive, platform independent communication library. pages, 230–241. ACM Press, 2003.

[7] David Bacon. Kava: A Java dialect with a uniform object model for lightweight classes. *Concurrency – Practice and Experience*, 15:185–206, 2003.

[8] V. Sarkar and G. R. Gao. Analyzable atomic sections: Integrating fine-grained synchronization and weak consistency models for scalable parallelism. Technical report, CAPSL Technical Memo 52, February 2004.

[9] Vivek Sarkar and Clay Williams and Kemal Ebcioğlu. Application development productivity challenges for high-end computing. In *Proceedings of Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, February 2004. http://www.research.ibm.com/arl/pphec/pphec2004-proceedings.pdf.

```
/**
  * Jacobi iteration
  *
  * At each step of the iteration, replace the value of a cell with
  * the average of its adjacent cells in the (i,j) dimensions.
  * Compute the error at each iteration as the sum of the changes
  * in value across the whole array. Continue the iteration until
  * the error falls below a given bound.
  *
  */

 public class Jacobi  {
     . . .

     // (1) Create distributions D, D_inner and D_boundary
     final region R=new region(new region(0,N+1),
                               new region(0,N+1));
     final region R_inner=new region(new region(1,N),
                                     new region(1,N));
     final distribution D = distribution.block(R);
     final distribution D_inner = D.restriction(R_inner);
     final distribution D_Boundary = D.difference(D_inner);

     public boolean run() {
         int iters = 0;
         // (2) Initialize array b
         double[D] b= new double[D];
         finish ateach(point [i,j]:D_inner) {b[i,j]=(double)i*N+j;}
         finish ateach(point [i,j]:D_boundary) {b[i,j]=0.0;}
         while(true) {
             // (3) Create array temp, and overlay it with array b
             final double[D_inner] temp = new double[D_inner];
             finish ateach(point [i,j]:D_inner) {
                 temp[i,j]=(b[i+1,j]+b[i-1,j]+b[i,j-1]+b[i,j+1])/4.0;}
             if ( b.restriction(D_inner).lift(double.sub,temp).lift(double.abs)
                   .reduce(double.add,0.0) < epsilon) break;
             b = b.overlay(temp);
             iters++;
         }

         // (4) Validate correctness of the run
         return b.reduce(double.add,0.0)==EXPECTED_CHECKSUM &&
                iters==EXPECTED_ITERS;
     }
 }
```

Figure 2: Jacobi example in X10

[10] Bradford L. Chamberlain and Sung-Eun Choi and Steven J. Deitz and Lawrence Snyder. The high-level parallel language ZPL improves productivity and performance. In *Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing*, 2004.

[11] W. Pugh. Java Memory Model and Thread Specification Revision, 2004. JSR 133, http://www.jcp.org/en/jsr/detail?id=133.

[12] HPL Workshop on High Productivity Programming Models and Languages, May 2004. http://hplws.jpl.nasa.gov/.