X10: An Object-Oriented Approach to Non-Uniform Cluster Computing

Philippe Charles
pcharles@us.ibm.com

Christopher Donawa
donawa@ca.ibm.com

Kemal Ebcioglu
kemal@us.ibm.com

Christian Grothoff
christian@grothoff.org

Allan Kielstra
kielstra@ca.ibm.com

Christoph von Praun
praun@us.ibm.com

Vijay Saraswat
vsaraswa@us.ibm.com

Vivek Sarkar
vsarkar@us.ibm.com

ABSTRACT

The next generation of high performance computers (e.g. those capable of $O(10^{15})$ operations per second) will be based on scale-out techniques rather than increasing clock rates. This leads to a notion of *clustered computing*: a single computer may contain hundreds of thousands of tightly coupled nodes. Unlike a distributed model, failure of a single node is tantamount to failure of the entire machine. However, the cost of memory access by a hardware processor may vary as much as five orders of magnitude across the cluster; hence the notion of a single shared memory may no longer be appropriate for such machines.

We have designed a concrete modern object-oriented programming language, X10, for high performance, high productivity programming of such machines. Past work in the Java Grande Forum has exposed the need for substantial changes in modern OO languages in order to support high performance computation (e.g. support for true multi-dimensional arrays, value types, relaxed exception model, changes to the concurrency model, support for distribution etc.) X10 builds on this past work. A member of the *partitioned global address space* family of languages (which includes Titanium, UPC and Co-Array Fortran), X10 is distinguished by the explicit reification of locality (*places*), termination detection (`finish`), by the use of lock-free synchronization (conditional atomic sections), and by the ability to express clustered data-structures (e.g. arrays scattered across multiple places). X10 smoothly integrates concurrent shared memory access (e.g. as expressed by OpenMP [2] or Java threads [8]) with message-passing (e.g. as expressed by MPI [21]). We present the design of the core features of the language, experience with a reference implementation and present results from some initial productivity studies.

1. INTRODUCTION

Modern OO languages, such as JAVA™ and C#, together with their runtime environments, libraries, frameworks and tools, have been widely adopted in recent years. Simultaneously, advances in technologies for *managed runtime environments* and *virtual machines* (VMs) have improved software productivity by supporting features such as portability, type safety, value safety, and automatic memory management. These languages have also made *concurrent* and *distributed* programming accessible to application developers (rather than just system programmers). They have supported two kinds of platforms: a uniprocessor or shared-memory multiprocessor (SMP) system where multiple threads execute against a single shared heap (in a single VM), and a distributed computing system in which each node has its own VM and communicates with other nodes using protocols such as Remote Method Invocation (RMI).

However, recent hardware technology trends have that established that frequency scaling in future systems can no longer maintain the steady "Moore's Law" growth of the last two decades. Instead, the dominant emerging structure is exemplified by SMP nodes with hierarchical heterogeneous levels of parallelism and severe non-uniformities in data access interconnected using centrally managed tightly-coupled cluster configurations (such as blade servers). We refer to these structures as *Non-Uniform Cluster Computing* (NUCC) systems to emphasize that they have attributes of both Non-Uniform Memory Access (NUMA) systems and cluster systems.

Current OO language facilities for concurrent and distributed programming, such as Java's thread construct, `java.util.concurrent` library and `java.rmi` package, are inadequate for addressing the needs of NUCC systems. They do not support the notions of non-uniform access within a node or tight coupling of distributed nodes. Instead, the state of the art for programming NUCC systems comes from the High Performance Computing (HPC) community, and is built on libraries such as MPI [21]. These libraries are accessible primarily to system experts rather than OO application developers. Further, even for the system experts, it is now common wisdom that the increased complexity of NUCC systems has been accompanied by a *decrease in software productivity* for developing, debugging, and maintaining ap-

plications for such machines [13]. As an example, current HPC programming models do not offer an effective solution to the problem of combining multithreaded programming and distributed-memory communications. Given that the majority of desktop systems in the future will be SMP nodes, and the majority of server systems will be configured as tightly-coupled clusters, there is an urgent need for new OO programming models and languages that simplify application development for NUCC systems, thereby making them more accessible to the mainstream community of OO application developers.

X10 [5, 6] is an experimental new language 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) with a technical agenda focused on hardware-software co-design that combines advances in chip technology, computer architecture, operating systems, compilers, programming environments and programming language design, so as to deliver new adaptable scalable systems in the 2010 timeframe. The X10 language is a big bet in the PERCS project to increase the programming productivity for future NUCC systems, without compromising performance. Combined with the PERCS Programming Tools agenda [20], 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 parallel applications by 2010.

To take advantage of recent language advances discussed above, X10 starts with a state-of-the-art OO programming model. To address non-uniformities in memory access, X10 introduces a notion of *places*. A place is a repository for data and the activities that operate on the data; a computation may consist of hundreds of thousands of places, with millions of activities. To support the overlap of concurrency with computation, X10 introduces *asynchronous operations* in lieu of threads. To support a simple high-level notion of shared-memory interaction, X10 provides a single construct – conditional atomic sections – inspired by the conditional critical regions of Hoare [12] and Brinch Hansen [10]. To repeatedly coordinate activities across multiple places, X10 introduces a notion of *clocks*, as a generalization of SIMD barriers. X10 permits multi-dimensional arrays to be scattered over multiple places, using the concept of regions and distributions from the language ZPL [4]. We believe that these high-level constructs are well-suited to scalable programming on NUCC systems and will be amenable to automatic static and dynamic optimizations within the 2010 time-frame.

The rest of the paper is organized as follows. Section 2 contains an overview of the X10 language. Section 4 contains a preliminary "productivity analysis" for X10 that uses publicly available benchmark programs to compares the programming effort required for parallelizing a serial application using currently avaiulable mechanisms in Java vs. using our proposed language features in X10. Section 3 discusses some common X10 idioms for concurrent and distributed programming of NUCC systems, and contrasts them with idioms employed in current programming models. Section 5 outlines new implementation implications for the X10 language, and provides a summary of our current reference implementation. Finally, Section 6 discusses related work and Section 7 contains our conclusions.

2. X10 LANGUAGE OVERVIEW

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, conditional atomic sections, clocked final and the nullable type constructor. Briefly, X10 may be thought of as (generic) JAVA with its current support for concurrency, arrays and primitive built-in types removed, and new language constructs introduced that are motivated by high-productivity high-performance parallel programming for future non-uniform cluster systems — *places*, *activities*, *clocks*, (distributed, multi-dimensional) *arrays* and *value* types. A summary of these language constructs can be found in Table 1.

2.1 Places and activities

Figure 1 contains a schematic overview of places and activities in the X10 programming model. An X10 computation acts on *data objects* through the execution of asynchronous lightweight threads called *activities*. A central new concept in X10 is that of a *place*. A place can be thought of as a collection of resident (non-migrating) activities and mutable data objects. Each place has place-local and activity-local storage classes that are not visible by remote places. In addition, X10 has a *partitioned global address space* (PGAS) that spans all the places in the program. An object in the PGAS is allocated at a specified place, but may be referenced by activities at other places. X10 supports a Globally Asynchronous Locally Synchronous (GALS) model for data access, which enables an activity to synchronously read and write data items in the (local) place where the activity is running but requires that all accesses to remote data be performed asynchronously. Though an activity executes at the same place throughout its lifetime, it may dynamically spawn activities in both the current and remote places. A remote location can be written into only by asynchronously spawning an activity to run at that location and perform the write operation. Similarly, to read a remote location, an activity must asynchronously spawn another activity at the remote place to return the value at the desired location. This operation returns immediately, leaving the spawning activity with a *future* for the result. Asynchronous activities need not be limited to a single read or write operation, and are permitted to contain more general computations.

Objects are of two kinds: a *scalar* object has a statically fixed set of fields, each of which has a distinct name. A scalar object is located at a single place and stays at that place throughout its lifetime. An *aggregate* object has many fields (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. The distribution of an aggregate object remains unchanged throughout the program's execution. X10 assumes an underlying garbage collector will dispose of (scalar and aggregate) objects and reclaim the memory associated with them once it can be determined that these objects are no longer accessible from the current state of the computation. There are no operations in the language to allow a programmer to explicitly release memory.

Statements:	
`async (Exp) Stm`	*Spawn an asynchronous activity*
`atomic Stm}`	*Atomically execute* `Stm`
`when (SimpleExp) stm [or (SimpleExp) Stm]*`	*Atomically execute if condition is satisfied*
`finish Stm`	*Wait until* `Stm` *terminates globally.*
`clocked (Exp,...,Exp) Stm`	*Execute* `Stm` *with the given clocks.*
`now (Exp) Stm`	*Ensure* `Stm` *terminates before clock* `Exp` *advances.*
`foreach (point i:  Region) Stm`	*Execute* `Stm` *in a separate local activity for each index point*
`ateach (point i:  Distribution) Stm`	*Execute* `Stm` *in a separate local activity for each index point*
Array Expressions:	
`new ArrayType ( Formal ) { Stm } }`	*Create a new initialized array.*
`Exp[ Exp1, ..., Expn ]`	*Access an n-dimensional array.*
`ArrayExpr | Region`	*Restrict an array to a given region.*
`ArrayExpr || ArrayExpr`	*Composition of two arrays, defined over disjoint regions.*
`ArrayExpr.scan( FunExpr )`	*Return parallel-prefix scan of array, with given function.*
`ArrayExpr.reduce( FunExpr )`	*Reduce the array with the given function.*
`ArrayExpr.lift( FunExpr )`	*Apply the function to each element of the array.*
Region:	
`Expr : Expr`	*Create a 1-dimensional array with given bounds.*
`[ Region, ..., Region]`	*Create a multi-dimensional array with given regions.*
`Region && Region`	*Intersection of two regions.*
`Region || Region`	*Disjoint union of two regions.*
`Regon - Region`	*Difference of two regions.*
`BuiltInRegion`	
Distribution:	
`Region -> Place`	*Constant map to the given place.*
`Distribution | Place`	*The inverse image region for this distributionfor this place.*
`Distribution | Region`	*The restriction of the distribution over the given region.*
`Distribution || Distribution`	*The parallel composition of distributions over disjoint regions.*
`Distribution - Distribution`	*Set difference of distributions*
`Distribution.overlay(Distribution)`	*Overlay distribution with another.*
`BuiltInDistribution`	

Table 1: X10 **Cheat Sheet**

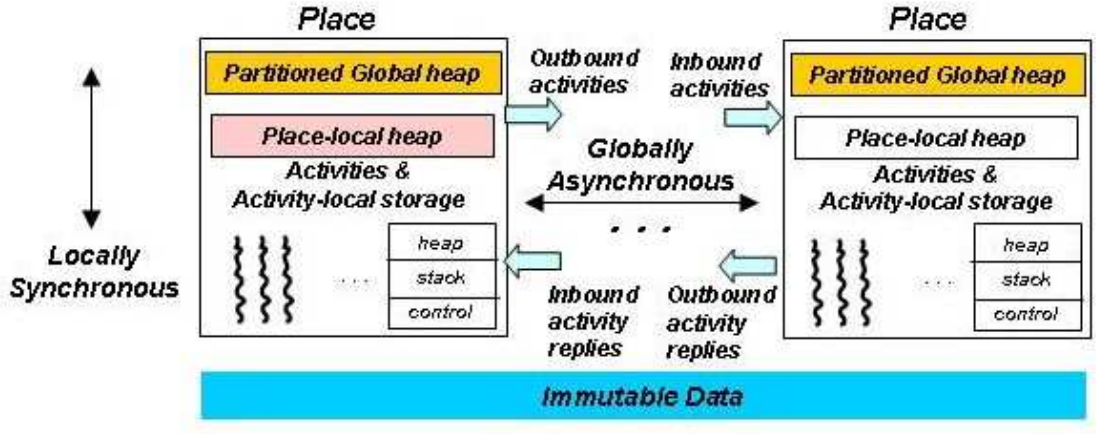

Figure 1: Overview of X10 **Places and Activities**

2.2 Asynchronous activities

An X10 computation may have many concurrent *activities* "in flight" at any give time. We use the term activity to denote a dynamic execution instance of a piece of code (with references to data). An activity is intended to execute in parallel with other activities. An activity may be thought of as a very light-weight thread. An activity may asynchronously and in parallel launch activities at other places. An activity is spawned in a given place and stays in that place for its lifetime. An activity may be *running*, *blocked* on some condition or *terminated*.

X10 distinguishes between *local* termination and *global* termination of a statement. The execution of a statement by an activity is said to terminate locally when the activity has finished all its computation related to that statement. For example, the creation of an asynchronous activity terminates locally when the activity has been created. A statement is said to terminate globally when it has terminated locally and all activities that it may have spawned at any place (if any) have, recursively, terminated globally.

An X10 computation is initiated as a single activity from the command line. This activity is the *root activity* for the entire computation. The entire computation terminates when (and only when) the root activity globally terminates. Thus X10 does not permit the creation of so called "daemon threads" – threads that outlive the lifetime of the root activity.

2.3 Activity spawning

An asynchronous activity is created by a statement `async (P) S` where `P` is a place expression and `S` is a statement. Such a statement is executed by spawning an activity at the place designated by `P` to execute statement `S`.

The statement is subject to the restriction that it must be acceptable as the body of a `void` method for an anoymous inner class declared at that point in the code, which throws no checked exceptions. As such, it may reference variables in lexically enclosing scopes provided that such variables are `final`.

An activity A executes the statement `async (P) S` by launching a new activity B at the designated place, to execute the specified statement. The statement terminates locally as soon as B is launched. The activation path for B is that of A, augmented with information about the statement which spawned B.

Multiple activities launched by a single activity at another place are not ordered in any way. They are added to the pool of activities at the target place and will be executed in sequence or in parallel based on the local scheduler's decisions. If the programmer wishes to sequence their execution s/he must use X10 constructs, such as clocks and `finish` to obtain the desired effect.

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 concurrent activities as threads or remote 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 parallel programming idioms including fine-grained threads, asynchronous DMA operations, message send (for an active or passive message), and scatter operations.

2.4 Finish

The statement `finish S` enforces global termination if `S`, and also provides a root activity for all activitivies spawned within `S`. Uncaught exceptions thrown or propagated by any activity spawned by `S` are accumulated at `finish S`. `finish S` terminates locally when all activities spawned by `S` terminate globally (either abruptly or normally). If `S` terminates normally, then `finish S` terminates normally and A continues execution with the next statement after `finish S`. If `S` terminates abruptly, then `finish S` terminates abruptly and throws a single exception formed from the collection of exceptions accumulated at `finish S`.

2.5 Asynchronous expressions and futures

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* [9] 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 parallel programming idioms including fine-grained threads, asynchronous DMA operations, message send/receive, and scatter/gather operations.

For example, the X10 statement,

```
future<int> F = future ( A.distribution[i] )
                { atomic{ return A[i]+B[k]; }  }
. . .
int n = F.force();
```

creates a new activity at the place containing element `A[i]`, with the values of local variables `A`, `B`, `i`, and `k` passed as implicit parameters to this activity. As with async statements, X10 also supports an *implicit syntax* for futures e.g., the above example could simply be written as `n = A[i] + B[k];`, which denotes the same asynchronous activity to be executed at the place containing `A[i]` and also `B[k]`. (If `A[i]` and `B[k]` are located at different places, then the implicit syntax will create futures for `A[i]` and `B[k]`, and will not be able to enforce atomicity on the accesses and addition of their values.)

2.6 Atomic sections

The simplest form of an atomic section is the *unconditional atomic section*: `atomic S` is a statement if `S` is a statement.

`S` may include method calls, conditionals, etc. It may not include the construction of any asynchronous activity. It

may not include any statement that may potentially block at runtime (e.g. `when`, `force` operations, `next` operations on clocks, or `finish`).

All these locations must statically satisfy the *locality condition*: they must belong to the place of the current activity. The compiler checks for this condition by checking whether the statement could be the body of a `void` method annotated with `local` at that point in the code.

An atomic statement is executed by an activity as if in a single step during which all other concurrent activities in the same place are suspended. If execution of the statement may throw an exception, it is the programmer's responsibility to wrap the atomic section within a `try/finally` clause and include appropriate recovery code in the finally clause. Thus the `atomic` statement only guarantees atomicity on successful execution, not on a faulty execution. X10 requires that all accesses to mutable data that can be shared my multiple activities in the same place must occur from within an atomic section.

We allow methods of an object to be annotated with `atomic`. Such a method is taken to stand for a method whose body is wrapped within an `atomic` statement.

An important property of an (unconditonal) atomic section is the following:

$$\texttt{atomic atomic}\ S \quad = \quad \texttt{atomic}\ S \tag{1}$$

Furthermore, an atomic section will eventually terminate successfully or thrown an exception; it cannot introduce a deadlock.

We pay special attentiotn to three important categories of atomic sections, which are amenable to optimized implementations:

1. No-op: these are cases (*e.g.,* a single load or store of a primitive data type) for which the atomicity is automatically guaranteed by the underlying hardware, and no additional support is required from the X10 compiler and runtime system.

2. Non-blocking: these are cases where atomicity can be guaranteed by non-blocking implementations that use non-blocking synchronization constructs [11] such as compare-and-swap, as exemplified in recent non-blocking concurrent algorithms [15].

3. Analyzable: these are cases where all memory references in the atomic section can be analyzed so that their addresses can be computed on entry to the atomic section. Analyzable atomic sections lend themselves to optimized lock assignment and consistency management approaches, as outlined in [19].

Consider the following atomic section as an 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 a non-blocking implementation.

2.7 Clocks

The MPI style of coordinating the activity of multiple processes with a single barrier is not suitable for the dynamic network of (possibly diverse) activities in an X10 computation. Instead, it becomes necessary to allow a computation to use multiple barriers. X10's *clocks* are designed to offer the functionality of multiple barriers in a dynamic context while still supporting determinate, deadlock-free parallel computation.

Activities may use clocks to repeatedly detect quiescence of an arbitrary programmer-specified, data-dependent set of activities. Each activity is spawned with a known set of clocks and may dynamically create new clocks. At any given time an activity is *registered* with zero or more clocks. It may register newly created activities with a clock, unregister itself with a clock, suspend on a clock or require that a statement (possibly involving execution of new async activities) be executed to completion before the clock can advance. At any given step of the execution a clock is in a given phase. It advances to the next phase only when all its registered activities have *quiesced* (by executing a `continue` operation on the clock), and all statements scheduled for execution in this phase have terminated. When a clock advances, all its activities may now resume execution.

Thus clocks act as *barriers* for a dynamically varying collection of activities. They generalize the barriers found in MPI style program in that an activity may use multiple clocks simultaneously. Yet programs using only clocks are guaranteed not to suffer from deadlock. Clocks are also integrated into the X10 type system, permitting variables to be declared so that they are `final` in each phase of a clock.

2.8 Scalar classes

An X10 scalar class has fields, methods and inner types (interfaces or classes), subclasses another class, and implements one or more interfaces. X10 classes live in a single-inheritance code hierarchy.

There are two kinds of scalar classes: *reference* classes and *value* classes.

A reference class typically has updatable fields. Objects of such a class may not be freely copied from place to place. Methods may be invoked on such an object only by an activity in the same place.

A value class has no updatable fields (defined directly or through inheritance), and allows no reference subclasses. Fields of value classes maybe be of a reference class type and therefore may contain references to objects with mutable state. Instances of value classes may be freely copied from place to place. Methods may be invoked on these instances from any place.

X10 has no primitive classes. However, the standard library `x10.lang` supplies (final) value classes `boolean`, `byte`, `short`, `char`, `int`, `long`, `float`, `double`, `complex` and `String`. The user may define additional arithmetic value classes using the facilities of the language.

2.9 Arrays, Regions and Distributions

An X10 array is a function from a *distribution* to a base type (which may itself be an array type).

A distribution is a map from a *region* to a subset of places. A region is a collection of *points* or *indices*. For instance, the region `[0:200,1:100]` specifies a collection of two-dimensional points `(i,j)` with `i` ranging from `0` to `200`

and `j` ranging from `1` to `100`. Points are used in array index expressions to pick out a particular array element.

Operations are provided to construct regions from other regions, and to iterate over regions. Standard set operations, such as union, disjunction and set difference are available for regions.

A primitive set of distributions is provided, together with operations on distributions. A *sub-distribution* of a distribution is one which is defined on a smaller region and agrees with the distribution at all points. The standard operations on regions are extended to distributions.

A new array can be created by restricting an existing array to a sub-distribution, by combining multiple arrays, and by performing pointwise operations on arrays with the same distribution.

X10 allows array constructors to iterate over the underlying distribution and specify a value at each item in the underlying region. Such a constructor may spawn activities at multiple places. Similarly, X10 provides *collective operations* that can be performed in parallel on distributed arrays.

2.10 Foreach and ateach

We introduce k-dimensional versions of iteration operations, `for` and `foreach`. In both statements, the expression is intended to be of type `region`. Expressions `e` of type `distribution` and `array` are also accepted, and treated as if they were `e.region`. The compiler throws a type error in all other cases.

The formal parameter must be of type `point`. Exploded syntax may be used. The parameter is considered implicitly final, as are all the exploded variables.

An activity executes a `for` statement by enumerating the points in the region in canonical order. The activity executes the body of the loop with the formal parameter(s) bound to the given point. If the body locally terminates successfully, the activity continues with the next iteration, terminating successfully when all points have been visited. If an iteration throws an exception then the for statement throws an exception and terminats abruptly.

An activity executes a `foreach` statement in a similar fashion except that separate `async` activities are launched in parallel in the local place for each point in the region. The statement terminates locally when all the activities have been spawned. It never throws an exception, though exceptions thrown by the spawned activities are propagated through to the root activity.

In an `ateach` statement, the expression is intended to be of type `distribution`. Expressions `e` of type `array` are also accepted, and treated as if they were `e.distribution`. The compiler throws a type error in all other cases. This statement differs from `foreach` only in that each activity is spawned at the place specified by the distribution for the point. That is, `ateach( point p[i1,...,ik]: A) S` may be thought of as standing for:

```
foreach (point p[i1,...,ik] : A)
  async (A.distribution[p]) {S}
```

2.11 The X10 rooted exception model

The rooted nature of X10 computations permits the definition of a *rooted* exception model. In multi-threaded programming languages there is a natural parent-child relationship between a thread and a thread that it spawns. Typically the parent thread continues execution in parallel with the child thread, and may terminate prior to the child thread. Therefore the parent thread cannot serve to catch exceptions thrown by the child thread.

Informally, the exception model for X10can be stated as follows. An uncaught exception propagates up the activation path to its nearest root activity, where it may be handled locally or propagated up the `root-of` tree when the activity terminates (based on the semantics of the statement being executed by the activity).[1] Thus, unlike other concurrent languages such as JAVA, X10 is able to avoid the cumbersome use of special constructs such as `ThreadGroup`s to catch asynchronously thrown exceptions.

3. X10 EXAMPLES

In this section, we use sample programs to discuss common X10 patterns for concurrency, and contrast them with the patterns employed in Java's multithreaded programming model.

3.1 RandomAccess

Figure 3 outlines an implementation for the RandomAccess HPC Challenge benchmark in X10. The statement labeled (1) is used to both allocate and initialize `Table` as a global block-distributed array, as a parallel operation spanning every place.

Next, the statement 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 statement 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 remote place containing `Table[j]`. This async statement spawns a remote activity to do an atomic read-exor-write operation on `Table[j]`. The `finish` statement guarantees that all `Table` updates are done before proceeding to statement (6).

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

3.2 SOR

Figure 2 outlines an X10 implementation for the *sor* benchmark, which will be discussed later in Section 4 along with other Java Grande Forum benchmarks. The ateach construct spawns one activity per place, as in RandomAccess. The projection, `R | here`, is used to compute the value of the local region, `Rlocal`, at each place. The sequential `start` loop is used to separate the parallel execution of odd/even iterations, by placing a `next` operation at the end of each iteration of the `start` loop. Finally, the `foreach` construct is used to specify intra-place parallel execution of activities.

3.3 MonteCarlo

Figure 4 shows three versions of the main computation loop for the MonteCarlo benchmark (discussed later in Section 4 along with other Java Grande Forum benchmarks)

[1] In X10 v0.41 the `finish` statement is the only statement that marks its activity as a root activity. Future versions of the language may introduce more such statements.

```
public boolean run() {
 // (1) Allocate and initialize Table as a block-distributed array
 long[.] Table = new long[distribution.factory.block(TABLE_SIZE)]
      (point [i]){return i;};
 // (2) Allocate and initialize RanStarts as a unique-distributed array
 // with one random number seed for each place
 long[.] RanStarts = new long[distribution.factory.unique()]
     (point [i]) {return starts(N_UPDATES_PER_PLACE*i);};
 // (3) Allocate a small immutable table that can be copied on all processors
 // and is used in generating the update values
 long value[.] SmallTable =
     new long value[S_TABLE_SIZE]
       (point [i]) {return i*S_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 [i]: RanStarts) {
     long ran = nextRandom(RanStarts[i]);
     // (5) Sequential loop
     for(point [count]: 1:N_UPDATES_PER_PLACE) {
         int J = f(ran);
         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.sum()==EXPECTED_RESULT;
```

Figure 2: RandomAccess example

```
  . . .
  region R = [0:(N+1), 0:(N+1)];
  distribution D = distribution.factory.block(R);
  double[.] G = new double[D];

  clock C = clock.factory.clock();
  finish clocked (C) ateach(point [pl]: distribution.factory.unique()) {
    // Execute one instance of the following code at each place
    region Rlocal = R | here;
    for (point [p]; 0:(num_iterations-1)) {
      for (point [start] : 1 : 2 ) {
        // Use start value to separate parallel execution of odd/even iterations
        region Rlocal_inner = [(Rlocal.low(0)+start):(Rlocal.high(0)-1):2,
                               (Rlocal.low(1)+1):(Rlocal.high(1)-1)];
        foreach ( point[i,j] : Rlocal_inner ) {
          G[i,j] = omega_over_four * (G[i-1,j] + G[i+1,j] + G[i,j-1] + G[i,j+1]) + one_minus_omega * G[i,j];
        }
        next; // acts as a barrier for activities registered on clock C
      }
    }
  }
  // finish imposes a logical barrier at the end of the ateach construct
  . . .
```

Figure 3: SOR example

— serial Java, multithreaded Java, and multithreaded X10. It can be seen clearly that the multithreaded X10 version is very similar in structure to the serial Java version — the main difference is in replacing the sequential `for` loop in the serial Java version by the parallel `foreach` construct in X10. (Since Vector.addElement() is a synchronized method in the Java versions, the X10 version of addElement was declared as atomic.) In contrast, the multithreaded Java version is far more complicated than the serial Java version because it involves creating a new `Runnable` class and computing the iteration slices to be executed by each thread.

Figure 5 shows two additional versions of the main computation loop for the MonteCarlo benchmark — MPI Java and multi-place X10. As before, we see that the MPI Java version is far more complicated than the multi-place X10 version. Note the use of the `async` operation in the multi-place X10 version to ensure that the `addElement()` operations are all performed at the place containing `D[0]`.

4. PRODUCTIVITY ANALYSIS — PARALLELIZATION OF SERIAL APPLICATIONS

This section presents preliminary productivity assessment of the X10 language. The focus of this assessment is on the effort required to convert a serial application to a) a shared-memory multithreaded parallel version, and b) a distributed-memory message-passing parallel version. This assessment was performed using the publicly available Java Grande Forum Benchmark Suite [1] which contains benchmarks that are available in three versions — serial, multithreaded [22] and message-passing based on mpiJava [23]. The metric used to measure effort is the *number of classes* and the *number of physical SLOC* (Source Lines of Code), which is a count of non-comment non-blank lines in the source code [18]. Number of classes is a well accepted indicator of the scale of an object oriented application. SLOC is known to be a reliable predictor of effort, but not of functionality: it has been empirically observed that a piece of code with a larger SLOC will require more effort to develop and maintain, but there is no guarantee that a piece of code with larger SLOC has more functionality than a piece of code with smaller SLOC. This limitation of SLOC is not an issue for our productivity analysis since the functionality of the code is standardized by the characteristics of the benchmark and this evaluation uses SLOC as a metric for effort and not functionality.

Table 2 summarizes the statistics for the publicly available serial, multithreaded and message-passing versions of the eight Java benchmark programs in the Java Grande Forum (JGF) Benchmark Suite for which all three versions are available. Five of the eight benchmarks come from Section 2 of the benchmark suite, which contains kernels that are frequenty used in high performance applications. The remaining 3 come from Section 3, which includes more complete applications. We did not include any examples from Section 1, since they consist of low level "microkernel" operations, which are not appropriate for productivity analysis.

Column 3 in Table 2 shows the total size of the Serial Java version of each benchmark. Column 4 shows the total size of the multithreaded Java version, as well as the number of classes and SLOC that were changed to obtain this version from the serial version. Similarly, Column 5 shows the total size of the multithreaded Java version, as well as the number of classes and SLOC that were changed to obtain this version from the serial version. Based on the classes and SLOC metrics, we see that the use of Java's multithreaded programming model for parallelization added 16% more SLOC compared to the serial Java version, and the use of MPI Java added 18% more SLOC. Using Java threads also led to additional classes being created. The JGF benchmarks do not contain any examples of combining Java threads with Java MPI, which is what's ultimately needed for programming NUCC systems. Since both levels of parallelism are orthogonal, it is not unreasonable to expect that the SLOC increase (compared to the serial version) for a combination of both programming models will approximately equal the sum of the individual increases *i.e.,* approximately 34%.

Table 3 summarizes the classes and SLOC statistics for X10 versions of six of the eight Java benchmark programs studied in Table 2. (The only reason for omitting the two missing benchmarks, `lufact` and `raytracer`, was lack of time to port them and validate a correct execution on the X10 reference implementation – this will be fixed in the final version of the paper.) All X10 versions reported in the table were executed and validated using the X10 reference implementation described in Section 5.2. Column 3 shows the total size of the Serial X10 version of each benchmark. In each case, the serial X10 version is very similar to the serial Java version, and differs primarily in the introduction of the *nullable* type constructor for certain field or variable declarations in the X10 version. Column 4 shows the total size of the single-place multi-activity X10 version, as well as the number of classes and SLOC that were changed to obtain this version from the serial version. Similarly, Column 5 shows the total size of the multi-place multi-activity X10 version, as well as the number of classes and SLOC that were changed to obtain this version from the serial version.

Comparing Table 3 with Table 2, we see that the multi-place multi-activity case for X10 added 10% more SLOC compared to the serial X10 version. This is almost half the half the SLOC increase observed for Java MPI (18%), and the difference becomes more significant when compared with the estimated 34% increase for combining Java threads and Java MPI.

5. X10 IMPLEMENTATION

5.1 Implementation Implications

This section summarizes the implementation implications of the X10 language, and outlines the requirements for the production-strength implementation currently being designed. The next section describes the existing prototype reference implementation.

A production-strength multi-node implementation of the X10 language will be implemented by a combination of a source compiler, a virtual machine, a supporting run-time and a dynamic Just In Time (JIT) compiler. with one VM instance per cluster node. The language design has implications on the implementation of each of these. Among the language features that have impacts on these implementations are the Partitioned Global Address Space (PGAS) model, the activity model and activity dispatch mechanisms, the object model and access to other languages.

The X10 type system allows the programmer to encode, in a data type, the placement of objects within the PGAS. References to PGAS objects are stored as *fat pointers*. The

```
Serial Java version:
--------------------
public void runSerial() {
  results = new Vector(nRunsMC);
  // Now do the computation.
  PriceStock ps;
  for( int iRun=0; iRun < nRunsMC; iRun++ ) {
      ps = new PriceStock();
      ps.setInitAllTasks((ToInitAllTasks) initAllTasks);
      ps.setTask((x10.lang.Object) tasks.elementAt(iRun));
      ps.run();
      results.addElement(ps.getResult());
  }
}

Multithreaded Java version:
---------------------------
public void runThread() {
  results = new Vector(nRunsMC);
  Runnable thobjects[] = new Runnable [JGFMonteCarloBench.nthreads];
  Thread th[] = new Thread [JGFMonteCarloBench.nthreads];
    for(int i=1;i<JGFMonteCarloBench.nthreads;i++) {
         thobjects[i] = new AppDemoThread(i,nRunsMC);
         th[i] = new Thread(thobjects[i]);
         th[i].start();
    }
    thobjects[0] = new AppDemoThread(0,nRunsMC); thobjects[0].run();
    for(int i=1;i<JGFMonteCarloBench.nthreads;i++) {
        try { th[i].join(); }
        catch (InterruptedException e) {}
    }
}

class AppDemoThread implements Runnable {
    int id,nRunsMC;
    public AppDemoThread(int id,int nRunsMC) {
        this.id = id;
        this.nRunsMC=nRunsMC;
    }
    public void run() {
        PriceStock ps;
        int ilow, iupper, slice;
        slice = (nRunsMC + JGFMonteCarloBench.nthreads-1)/JGFMonteCarloBench.nthreads;
        ilow = id*slice;
        iupper = (id+1)*slice;
        if (id==JGFMonteCarloBench.nthreads-1) iupper=nRunsMC;
        for( int iRun=ilow; iRun < iupper; iRun++ ) {
            ps = new PriceStock();
            ps.setInitAllTasks(AppDemo.initAllTasks);
            ps.setTask(AppDemo.tasks.elementAt(iRun));
            ps.run();
            AppDemo.results.addElement(ps.getResult());
        }
    }
}

Multithreaded X10 version:
--------------------------
public void runThread() {
    results = new Vector(nRunsMC);
    finish foreach (point [iRun] : 0:(nRunsMC-1) ) {
        PriceStock ps = new PriceStock();
        ps.setInitAllTasks((ToInitAllTasks)initAllTasks);
        ps.setTask(tasks.elementAt(iRun));
        ps.run();
        results.addElement(ps.getResult());
    }
}
```

Figure 4: Comparison of Java vs. X10 multithreaded code from MonteCarlo benchmark

```
MPI Java version:
-----------------
 public void runSerial() throws MPIException{
    int ilow,ihigh;
    if(JGFMonteCarloBench.rank==0) {
      results = new Vector(nRunsMC);
    }
    p_nRunsMC = (nRunsMC + JGFMonteCarloBench.nprocess -1) / JGFMonteCarloBench.nprocess;
    p_results[0] = new Vector(p_nRunsMC);
    ilow = JGFMonteCarloBench.rank*p_nRunsMC;
    ihigh = (JGFMonteCarloBench.rank+1)*p_nRunsMC;
    if (JGFMonteCarloBench.rank==JGFMonteCarloBench.nprocess-1) ihigh = nRunsMC;
    // Now do the computation.
    PriceStock ps;
    for( int iRun=ilow; iRun < ihigh; iRun++ ) {
        ps = new PriceStock();
        ps.setInitAllTasks(initAllTasks);
        ps.setTask(tasks.elementAt(iRun));
        ps.run();
        p_results[0].addElement(ps.getResult());
    }

    if(JGFMonteCarloBench.rank==0) {
     for(int i=0;i<p_results[0].size();i++){
       results.addElement((ToResult) p_results[0].elementAt(i));
     }
     for(int j=1;j<JGFMonteCarloBench.nprocess;j++) {
       p_results[0].removeAllElements();
       MPI.COMM_WORLD.Recv(p_results,0,1,MPI.OBJECT,j,j);
       for(int i=0;i<p_results[0].size();i++){
           results.addElement((ToResult) p_results[0].elementAt(i));
       }
     }

    } else {

     MPI.COMM_WORLD.Send(p_results,0,1,MPI.OBJECT,0,JGFMonteCarloBench.rank);
    }

 }

Multiple-place X10 version:
---------------------------
    public void runDistributed() {
        results = new Vector(nRunsMC);
        distribution D = distribution.factory.block(0:(nRunsMC-1));
        // Now do the computation
        finish ateach (point[iRun] : D ) {
            final PriceStock ps = new PriceStock();
            ps.setInitAllTasks((ToInitAllTasks) initAllTasks);
            ps.setTask((x10.lang.Object) tasks.elementAt(iRun));
            ps.run();
            async (D[0]) results.addElement(ps.getResult());
        }
    }
```

Figure 5: Comparison of MPI Java vs. X10 multiple-place code from montecarlo benchmark

Benchmark	Physical SLOC	Serial Java	Multithreaded Java	MPI Java
crypt	Total size (classes/SLOC)	7/526	8/591	7/634
(Section 2)	SLOC ratio, relative to Serial version		1.12	1.21
	SLOC changed in Serial version		4	17
	SLOC added to Serial version		114	108
	SLOC deleted from Serial version		49	0
lufact	Total size (classes/SLOC)	7/588	8/783	7/748
(Section 2)	SLOC ratio, relative to Serial version		1.33	1.27
	SLOC changed in Serial version		3	18
	SLOC added to Serial version		38	161
	SLOC deleted from Serial version		42	1
series	Total size (classes/SLOC)	7/402	8/456	7/503
(Section 2)	SLOC ratio, relative to Serial version		1.13	1.25
	SLOC changed in Serial version		10	18
	SLOC added to Serial version		60	101
	SLOC deleted from Serial version		6	0
sor	Total size (classes/SLOC)	7/364	8/465	7/539
(Section 2)	SLOC ratio, relative to Serial version		1.28	1.48
	SLOC changed in Serial version		12	27
	SLOC added to Serial version		101	175
	SLOC deleted from Serial version		0	0
sparsematmult	Total size (classes/SLOC)	7/366	8/466	7/468
(Section 2)	SLOC ratio, relative to Serial version		1.27	1.28
	SLOC changed in Serial version		11	24
	SLOC added to Serial version		100	102
	SLOC deleted from Serial version		0	0
moldyn	Total size (classes/SLOC)	9/655	12/811	9/736
(Section 3)	SLOC ratio, relative to Serial version		1.24	1.12
	SLOC changed in Serial version		85	13
	SLOC added to Serial version		111	94
	SLOC deleted from Serial version		8	13
montecarlo	Total size (classes/SLOC)	19/1405	20/1456	19/1494
(Section 3)	SLOC ratio, relative to Serial version		1.04	1.06
	SLOC changed in Serial version		14	15
	SLOC added to Serial version		51	89
	SLOC deleted from Serial version		0	0
raytracer	Total size (classes/SLOC)	17/863	20/977	17/953
(Section 3)	SLOC ratio, relative to Serial version		1.13	1.10
	SLOC changed in Serial version		12	17
	SLOC added to Serial version		66	90
	SLOC deleted from Serial version		5	0
TOTAL	Total size (classes/SLOC)	80/5169	92/6005	80/6075
(all benchmarks)	SLOC ratio, relative to Serial version		1.16	1.18

Table 2: Classes and SLOC changed, added, and deleted to obtain parallel versions of a serial Java program

Benchmark	Classes+SLOC	Serial X10	Single-place Multi-activity X10	Multi-place Multi-activity X10
crypt	Total size (classes/SLOC)	7/515	8/575	7/584
(Section 2)	SLOC ratio, relative to Serial version		1.12	1.13
	SLOC changed in Serial version		21	23
	SLOC added to Serial version		104	87
	SLOC deleted from Serial version		44	9
lufact	Total size (classes/SLOC)	7/598	8/724	7/756
(Section 2)	SLOC ratio, relative to Serial version		1.21	1.26
	SLOC changed in Serial version		49	89
	SLOC added to Serial version		21	176
	SLOC deleted from Serial version		39	9
series	Total size (classes/SLOC)	7/399	8/451	7/463
(Section 2)	SLOC ratio, relative to Serial version		1.13	1.16
	SLOC changed in Serial version		23	25
	SLOC added to Serial version		58	81
	SLOC deleted from Serial version		6	8
sor	Total size (classes/SLOC)	7/345	7/347	7/367
(Section 2)	SLOC ratio, relative to Serial version		1.01	1.06
	SLOC changed in Serial version		7	9
	SLOC added to Serial version		2	20
	SLOC deleted from Serial version		0	0
moldyn	Total size (classes/SLOC)	7/646	8/748	7/721
(Section 3)	SLOC ratio, relative to Serial version		1.16	1.12
	SLOC changed in Serial version		97	56
	SLOC added to Serial version		110	82
	SLOC deleted from Serial version		8	7
montecarlo	Total size (classes/SLOC)	18/1356	18/1343	18/1343
(Section 3)	SLOC ratio, relative to Serial version		0.99	0.99
	SLOC changed in Serial version		3	4
	SLOC added to Serial version		0	0
	SLOC deleted from Serial version		13	13
TOTAL	Total size (classes/SLOC)	53/3859	57/4188	53/4234
(all benchmarks)	SLOC ratio, relative to Serial version		1.09	1.10

Table 3: Classes and SLOC changed, added and deleted to obtain parallel versions of a serial X10 program

presence of globally visible objects and the design of X10 fat pointers impact the garbage collection (GC) system in the VM. Unlike other PGAS languages that do not provide a managed runtime system (such as UPC [7]), X10 fat pointers are designed to allow relocation of global objects so as to support intra-place compaction. For scalability, we plan to support local GC within a place, combined with a distributed GC algorithm.

The production strength X10 implementation will be built on a highly efficient inter-place communication system with properties similar to those found in IBM's Low-level Application Programming Interface (LAPI) library [3] or the Message Passing Interface (MPI) library [21]. The communication system will be capable of efficiently supporting short messages (e.g., asynchronous activities with a small number of scalar parameters) as well as long messages (e.g., asynchronous activities or collective operations on large array sections) with zero-copy transfer. In addition, the data involved in a large message will be stored in a memory region that does not interfere with frequent lightweight GC pasess, as in a generational collector. The inter-place communication system will also provide efficient support for operations on X10 clocks (comparable to the support provided for barriers in other languages).

X10 activities are created, executed and terminated in very large numbers and must be supported by an efficient lightweight threading model. The thread scheduler will efficiently dispatch activities as they are created or become unblocked (and possibly re-use threads that are currently blocked). The scheduler will also attempt to schedule activities in such a way as to reduce the likelihood of blocking. The compilers (source and JIT) also reduce the demands on the threading system by optimizations such as coalescing of asynchronous statements and futures, where possible.

Unlike Java, X10 allows *header-less* value class objects to be embedded (inlined) in other objects or created on the stack, similar to C# structs. The production design will ensure efficient inlining for multi-dimensioned arrays of value types, such as `complex`. The inlined array will contain exactly one header for the element value type in the array descriptor, thereby reducing the overhead of including a header for each array element. The array descriptor contains a reference to its distribution (which, in turn, refers to its region and index bounds). Relative to a Java implementation, this scheme slightly complicates both GC (because the structure of objects containing nested header-less objects is more complicated) and bounds checking operations.

The X10 "extern" declaration allows programs written in other languages such as C and FORTRAN direct access to X10 data. To implement this feature, the system will accommodate restrictions on the layout of objects imposed by the "extern" language as well as allow pointers to X10 objects to leave the controlled confines of the virtual machine for periods of time.

Finally, the extensively distributed nature of X10 programs (and other characteristics of HPC programs) implies that the implementation must be engineered to exploit more ahead of time (AOT) compilation and packaging than is seen in typical Java systems.

5.2 Reference Implementation

This section describes the current reference implementation for X10. The goal of the reference implementation is to provide a prototype development environment for X10, that can be used to obtain feedback on the language design by experimenting with sample programs written in X10 as well as perform productivity assessments. The reference implementation is currently functional and supports most of the language constucts presented in Section 2. The main limitations are that the `clocked final` construct is currently not supported, and static checking of type rules and other language properties is currently incomplete.

The prototype is based on a *source-to-source translator* that translates X10 source to Java and the X10 *runtime* which controls the execution of the generated Java code. The X10 runtime itself is written in Java and runs on top of a Java Virtual Machine (JVM). It reuses JVM servics such as a garbage collector, dynamic type checking and polymorphic method dispatch to implement the corresponding X10 features.

Figure 6 shows the structure of the X10 reference implementation. The X10-to-Java translator is built on top of the Polyglot [17] compiler framework, and consists of an X10 parser, multiple analysis phases on the X10Abstract Syntax Tree (AST), followed by a template-driven code emitter. Most X10 constructs are translated to Java using template files, called X10 Compiler Definition (XCD) files, which map nodes of the abstract syntax tree to fragments of Java source code. Different implementation strategies for the various X10 constructs can be realized simply by changing the template definitions. The Java code in the XCD files include calls to the X10 runtime for functionalities that are not directly available in the Java runtime environment.

The generated Java code executes on a standard JVM, in conjunction with the X10 Run Time System (RTS) which provides abstractions for places, regions, distributions, arrays and clocks. Some RTS functions are directly available to the X10 programmer in the form of `x10.lang.*` libraries. Other RTS functions are only accessible to the Java code generated by the X10 compiler. In addition to implementing the core features of the X10 language, the runtime system is instrumented to collect data about the dynamic execution chacteristics of an application. For example, the number of remote vs. local data accesses can be computed for each place. This kind of data provides valuable information to programmer for high-level of tuning load balance and data distribution, independent of the performance characteristics of the target machine. Lower-level tuning for real parallel hardware will only be meaningful for the production-strength implementation, not the reference implementation.

Multiple X10 places are simulated in a single JVM instance in the reference implementation. The design of a place corresponds to the *executor* pattern [14]. A place acts as an executor that dispatches individual activities to a pool of Java threads, as outlined in Figure 7. Activities in X10 can be created by async statements, future expressions, and foreach and ateach iterations. The thread pool model makes it possible to reuse a thread for multiple X10 activities. However if an X10 activity blocks (*e.g.,* due to a force operation on a future, or a next operation on a clock) its thread will not be available till the activity resumes and completes execution. In that case, a new Java thread will need to be created to serve any outstanding activities. As a result, the total number of threads that can be created in a single place is unbounded in the reference implementation. For the production-strength implementation, we are exploring tech-

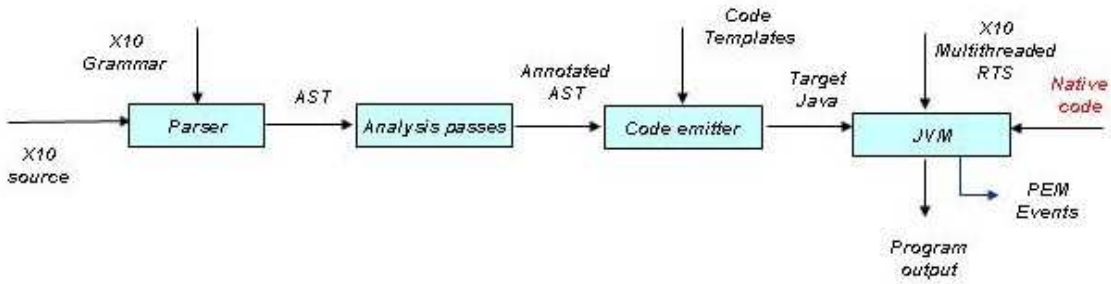

Figure 6: Structure of X10 Reference Implementation

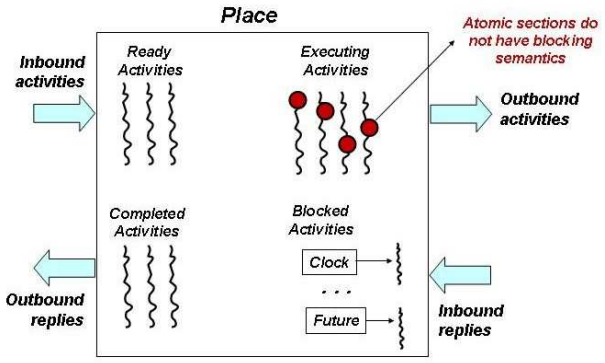

Figure 7: Structure of a single X10 place

niques that will enable us to store the execution context of an activity and thereby bound the number of threads in a single place.

Conditional and unconditional atomic sections are implemented using a single mutual exclusion lock per place. While this implementation strategy limits parallelism, it is guaranteed to be a deadlock-free implementation, since no X10 computation can require the implementation to acquire locks for more than one place.

The X10 language includes support for rectangular multi-dimensional arrays, in addition to the nested arrays available in Java. An X10 multi-dimensional arrays is flattened and represented internally as a combination of an X10 array descriptor object and a single-dimensional Java array [16]. The array descriptor is part of a boxed array implementation which provides information on the distribution and shape of the array, as well as convenient support for collective operations.

An important design goal for X10 is convenient and efficient inter-operability with native code via the `extern` construct. It should be possible for X10 programs to use existing components and libraries implemented in statically-compiled langauges such as C, C++, and Fortran. Though rich in functionality, the Java Native Interface (JNI) is neither convenient nor efficient for many common cases such as invoking a high-performance mathematical library on multi-dimensional arrays created by a Java application. JNI is designed for the general case in which the Java representation of a data type (including arrays) can be different from the representation in native code, and therefore imposes a restricted API for remote accesses between Java and native data spaces. That is, to inspect or change the value of a Java object from C code (say), an explicit call must be made specifying the new value – there is no direct manipulation of a Java object from native space, or vice versa. This restriction can be especially inefficient and onerous to program when working with arrays. If some native code manipulated an array, and we wanted to use this code on a Java array, then either the code would need to be modified so that each array access was replaced by a JNI call, or the java array would need to be copied element by element to a native array so it could be manipulated, and then copied back via JNI calls on each element. Since X10 multi-dimensional arrays are represented in row-major format, we permit native C/C++ code to operate directly on X10 arrays, thereby reducing the burden on the native code implementation. The same approach can be used to interface with Fortran code, but right now it is the responsibility of the programmer ot be aware of the fact that X10 arrays follow a row-major layout and Fortran arrays follow a column-major layout.

The process for invoking extern code from an X10 program is as follows. The programmer includes an `extern` declaration in the X10 program for eahc native function that is to be invoked, and a `System.loadLibrary()` call for each native DLL to be loaded. The X10 compiler generates C wrapper files that are used when recompiling the native library. Our current deisgn assumes that re-compiling the native library will not be a significant inhibitor, compared to rewriting the internals of the native code as is necessary for JNI. Also, all arrays that are passed to native code are allocated in an unmanaged memory area. The plan for the production-strength implementation is to explore the use of protection techniques that will only allow native code access to the unmanaged memory area, while ensuring safety of object references in the managed memory area.

6. RELATED WORK

X10 shares many common features with JAVA, C# and other modern, class-based, single-inheritance, mostly statically-typed, OO languages. It is distinguished from them primarily in its treatment of concurrency and distribution. Monitors/locks are eschewed in favor of the considerably simpler notion of atomic sections. The programmer does not have to concern herself with trying to detrermine whether certain memory accesses should be guarded by a lock, and if so, by how many locks and in which order. Rather she can concern herself with which object invariants need to be maintained, and ensure that all updates to variables involved in

an invariant be done atomically, moving the object from a consistent state to a consistent state. Mechanisms for distribution, asynchronous operation, coordination (clocks) are introduced. Value types are introduced to permit free copying of data-structures between places.

In the area of scientific computing, the programming languages community responded to the challenges of programming shared-memory and distributed-memory multiprocessors with the design of several new programming languages, including Sisal, Fortran 90, High Performance Fortran, Kali, ZPL, UPC, Co-Array Fortran, and Titanium. At a high level, X10 differs from these languages – while liberally borrowing ideas from them – by being squarely based in a type-safe, garbage-collected, OO model. X10 guarantees type safety: every variable has a statically known type and is guaranteed to contain at runtime only those values which satisfy the invariants of the type. Only those operations can be performed on such a variable which are sanctioned by its static type. X10 guarantees *memory safety*: an object may only access the memory within its representation, and within the representation of other objects it has a reference to. In particular, X10 does not support pointer arithmetic. Array acess are bounds-checked dynamically (if necessary). No object can have a reference to an object whose memory has been freed (no "illegal memory reference" errors). The programming model guarantees that every variable is initialized, and only that value can be read from a variable which has earlier been written into a variable by program action. X10 supports *pointer safety*. It introduces a special type constructor `nullable` and guarantees that no operation on a variable of type `T` that is not a `nullable` type will throw a Null Pointer Exception.

7. CONCLUSIONS

The ultimate challenge facing the HPC 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.

We believe that X10 has the potential to take significant steps towards this goal. In future work we plan to investigate both the high-productivity and high-performance hypotheses.

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 on the design and implementation of X10: George Almasi, David Bacon, Bob Blainey, Calin Cascaval, Perry Cheng, Julian Dolby, Guang Gao, David Grove, Radha Jagadeesan, Maged Michael, Robert O'Callahan, Filip Pizlo, V.T. Rajan, Mandana Vaziri, and Jan Vitek.

8. REFERENCES

[1] The java grande forum benchmark suite. www.epcc.ed.ac.uk/javagrande/javag.html.

[2] Openmp specifications. www.openmp.org/specs.

[3] IBM International Technical Support Organization Poughkeepsie Center. Technical presentation for pssp version 2.3. Technical report sg24-2080-00, December 1997. http://www.redbooks.ibm.com/redbooks/pdfs/sg242080.pdf.

[4] Bradford L. Chamberlain, Sung-Eun Choi, 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.

[5] Kemal Ebcioğlu, Vijay Saraswat, and Vivek Sarkar. X10: Programming for hierarchical parallelism and nonuniform data access (extended abstract). In *Language Runtimes '04 Workshop: Impact of Next Generation Processor Architectures On Virtual Machines (colocated with OOPSLA 2004)*, October 2004. www.aurorasoft.net/workshops/lar04/lar04home.htm.

[6] Kemal Ebcioğlu, Vijay Saraswat, and Vivek Sarkar. X10: an experimental language for high productivity programming of scalable systems (extended abstract). In *Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, February 2005.

[7] T El-Ghazawi, W. Carlson, and J.Draper. UPC Language Specification v1.1.1, October 2003.

[8] J. Gosling, W. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, 2000.

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

[10] Per Brinch Hansen. Structured multiprogramming. *CACM*, 15(7), July 1972.

[11] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

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

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

[14] Dough Lea. *Concurrent Programming in Java, Second Edition*. Addison-Wesley, Inc., Reading, Massachusetts, 1999.

[15] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275. ACM Press, 1996.

[16] Jose Moreira, Samuel Midkiff, and Manish Gupta. A comparison of three approaches to language, compiler, and library support for multidimensional arrays in java computing. In *Proceedings of the ACM Java Grande - ISCOPE 2001 Conference*, June 2001.

[17] Nathaniel Nystrom, Michael R. Clarkson, and

Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *Proceedings of the Conference on Compiler Construction (CC'03)*, pages 1380–152, April 2003.

[18] Robert Park. Software size measurement: A framework for counting source statements. Technical report, CMU/SEI-92-TR-020 Technical Report, 1992.

[19] 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.

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

[21] Anthony Skjellum, Ewing Lusk, and William Gropp. *Using MPI: Portable Parallel Programming with the Message Passing Iinterface*. MIT Press, 1999.

[22] L. A. Smith and J. M. Bull. A multithreaded java grande benchmark suite. In *Proceedings of the Third Workshop on Java for High Performance Computing*, June 2001.

[23] L. A. Smith, J. M. Bull, and J. Obdrzalek. A parallel java grande benchmark suite. In *Proceedings of Supercomputing 2001, Denver, Colorado*, November 2001.