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

It is now well established that the device scaling predicted by Moore's Law is no longer a viable option for increasing the clock frequency of future uniprocessor systems at the rate that had been sustained during the last two decades. As a result, future systems are rapidly moving from uniprocessor to multiprocessor configurations, so as to use parallelism instead of frequency scaling as the foundation for increased compute capacity. The dominant emerging multiprocessor structure for the future is a *Non-Uniform Cluster Computing* (NUCC) system with nodes that are built out of multi-core SMP chips with non-uniform memory hierarchies, and interconnected in horizontally scalable cluster configurations such as blade servers. Unlike previous generations of hardware evolution, this shift will have a major impact on existing software. Current OO language facilities for concurrent and distributed programming are inadequate for addressing the needs of NUCC systems because they do not support the notions of non-uniform data access within a node, or of tight coupling of distributed nodes.

We have designed a modern object-oriented programming language, X10, for high performance, high productivity programming of NUCC systems. A member of the *partitioned global address space* family of languages, X10 highlights the explicit reification of locality in the form of *places*; lightweight activities embodied in *async*, *future*, *foreach*, and *ateach* constructs; a construct for termination detection (*finish*); the use of lock-free synchronization (*atomic blocks*); and the manipulation of cluster-wide global data structures. We present an overview of the X10 programming model and language, experience with our reference implementation, and results from some initial productivity comparisons between the X10 and JAVA™ languages.

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

[†]UCLA Computer Science Department, Boelter Hall, Los Angeles, CA 90095, USA.

[‡]IBM Toronto Laboratory, 8200 Warden Avenue, Markham ON L6G 1C7, Canada.

OOPSLA'05, October 16–20, 2005, San Diego, California, USA.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming, Parallel programming*; D.3.2 [**Programming Languages**]: Language Classifications—*Concurrent, distributed, and parallel languages, Object-oriented languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures*

General Terms

Languages, Performance, Design

Keywords

X10, Java, Multithreading, Non-uniform Cluster Computing (NUCC), Partitioned Global Address Space (PGAS), Places, Data Distribution, Atomic Blocks, Clocks, Scalability, Productivity

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 one or more threads execute against a single shared heap in a single VM, and a loosely-coupled distributed computing system in which each node has its own VM and communicates with other nodes using inter-process protocols, such as Remote Method Invocation (RMI) [36].

However, recent hardware technology trends have established that the device scaling predicted by Moore's Law is no longer a viable option for increasing the clock frequency of future uniprocessor systems at the rate that had been sustained during the last two decades. As a result, future systems are rapidly moving from uniprocessor to multiprocessor configurations. Parallelism is replacing frequency scaling as the foundation for increased compute capacity. We believe future server systems will consist of multi-core SMP nodes with non-uniform memory hierarchies, interconnected in horizontally scalable cluster configurations such as blade servers. We refer to such systems 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 threads, the `java.util.concurrent` library and the `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 [51]. 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 across the application development, debugging, and maintenance life-cycle [33]. 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 future desktop systems will be SMP nodes, and the majority of server systems will be tightly-coupled, horizontally scalable clusters, we believe there is an urgent need for a new OO programming model for NUCC systems. Such a model will continue to make concurrent and distributed programming accessible to application programmers on current architectures.

X10 [16, 17, 47] 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). By 2010, the project aims to deliver new adaptable, scalable systems that will provide a 10× improvement in development productivity for parallel applications. To accomplish this goal, PERCS is using a hardware-software co-design methodology to integrate advances in chip technology, architecture, operating systems, compilers, programming language and programming environment design.

X10 is a "big bet" in the PERCS project. It aims to deliver on the PERCS 10× promise by developing a new programming model, combined with a new set of tools (as laid out in the PERCS Programming Tools agenda, [50]). X10 is intended to increase programmer productivity for NUCC systems without compromising performance. X10 is a type-safe, modern, parallel, distributed object-oriented language, with support for high performance computation over distributed multi-dimensional arrays. Work on X10 began in February 2004 using the PERCS Programming Model [50] as a starting point, and the core design was stabilized in a few months. To date, we have designed the basic programming model; defined the 0.41 version of the language (and written the Programmers' Manual); formalized its semantics [47] and established its basic properties; built a single-VM reference implementation; and developed several benchmarks and applications. We are at an early stage in the life-cycle of the language. We expect the ongoing work on several cutting-edge applications, tools for the X10 programmer, and an efficient compiler and multi-VM runtime system to significantly inform further development of the language.

This paper serves as an overview of the design of X10 v 0.41. Section 2 summarizes the motivation for X10, and the principles underlying the X10 design. Section 3 contains an overview of the X10 programming model. Section 4 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 contains a productivity analysis for X10 that uses publicly available benchmark programs to compare the programming effort required for parallelizing a serial application using mechanisms currently available in Java vs. X10. Section 6 summarizes our current reference implementation, and outlines the implications for an efficient implementation of X10. Finally, Section 7 discusses related work. Section 8 contains our conclusions, and mentions several areas of future work including future extensions to the X10 programming model.

2. X10 DESIGN RATIONALE

In this section, we outline the language and developmental principles underlying the X10 design.

2.1 Goals

To enable delivery of productivity and performance on NUCC platforms, X10 balances four major goals: *safety*, *analyzability*, *scalability*, and *flexibility*.

Safety. The X10 programming model is intended to be safe, in the interests of productivity. Large classes of errors common in HPC applications, such as illegal pointer references, type errors, initialization errors, buffer overflows are to be ruled out by design. Modern memory-managed OO languages such as the Java language show how this can be accomplished for sequential languages. Additionally, common patterns of usage are to be guaranteed to preserve determinacy and avoid deadlocks.

Analyzability. X10 programs are intended to be analyzable by programs (compilers, static analysis tools, program refactoring tools). Simplicity and generality of analyses requires that the programming constructs – particularly those concerned with concurrency and distribution – be conceptually simple, orthogonal in design, and combine well with as few limitations and corner cases as possible. For this a language close to the machine, (e.g. `C`) is not as attractive as a language with strong data-encapsulation and orthogonality properties such as the Java language.

Ideally, it should be possible to develop a formal semantics for the language and establish program refactoring rules that permit one program fragment to be replaced by another while guaranteeing that no new observable behaviors are introduced. With appropriate tooling support (e.g. based on Eclipse) it should be possible for the original developer, or a systems expert interested in optimizing the behavior of the program on a particular target architecture, to visualize the computation and communication structure of the program, and refactor it (e.g. by aggregating loops, separating out communication from computation, using a different distribution pattern etc). At the same time analyzability contributes to performance by enabling static and dynamic compiler optimizations.

Scalability. A program is scalable if the addition of computational resources (e.g. processors) leads to an increase in performance (e.g. reduction in time to completion). The scalability of a program fundamentally depends on the properties of the underlying algorithm, and not the programming language. However a particular programming language should not force the programmer to express such algorithms through language concurrency and distribution constructs that are themselves not scalable. Conversely, it should help the programmer identify hot spot design patterns in the code that mitigate against scalability (e.g. inner loops with large amounts of communication). Scalability contributes to performance, and also productivity because the effort required to make an application scale can be a major drain on productivity.

Flexibility. Scalable applications on NUCC systems will need to exploit multiple forms of parallelism: data parallelism, task parallelism, pipelining, input parallelism etc. To this end it is necessary that the data-structuring, concurrency and distribution mechanisms be general and flexible. For instance, a commitment to a single processor multiple data (SPMD) program model would not be appropriate.

Figure 1 outlines the software stack that we expect to see in future NUCC systems, spanning the range from tools and very high level languages to low level parallel and communication runtime systems and the operating system. X10 has been deliberately positioned at the midpoint, so that it can serve as a robust foundation for parallel programming models and tools at higher levels while still delivering scalable performance at lower levels. and thereby achieve our desired balance across safety, analyzability, scalability and flexibility.

2.2 Key Design Decisions

With these goals in mind, we made five key design decisions at the start of the X10 project:

1. Introduce a new programming language, instead of relying on other mechanisms to support new programming models.

2. Use the JAVA programming language as a starting point for the serial subset of the new programming model.

3. Introduce a *partitioned global address space* (PGAS) with explicit reification of locality in the form of *places*.

4. Introduce *dynamic, asynchronous activities* as the foundation for concurrency constructs in the language.

5. Include a rich *array sub-language* that supports dense and sparse *distributed multi-dimensional arrays*.

New programming language. Our first design decision was to pursue a solution based on a *new programming language*, rather than introducing new libraries, frameworks, or pseudo-comment directives to support our programming model. Our belief is that future hardware trends towards NUCC systems will have a sufficiently major impact on software to warrant introducing a new language, especially in light of the safety, analyzability goals and flexibility outlined above.

Extend a modern OO foundation. Our second decision was to use the JAVA programming language as the foundation for the serial subset of X10. A strong driver for this decision is the widespread adoption of the JAVA language and its accompanying ecosystem of documentation, libraries, and tools. Second, it supported our safety, analyzability and flexibility goals. Third, it enabled us to take advantage of the significant work that has already been done (e.g. in the context of the JAVA Grande Forum) to investigate and remove the limitations of the JAVA language for high-performance computing [43]. Other than a few constructs that are present in the X10 serial subset but not in the JAVA serial subset (notably *nullable*, *value types*, *multi-dimensional arrays*, *region iterators* and *extern*), the serial subsets of both languages are very similar.

This was a controversial decision because the JAVA language has not as yet proved itself as a viable platform for scalable high-performance computing. Based on the future roadmap projected for virtual machine and dynamic compilation technologies, we decided to bet that the performance gap between high performance JAVA applications and C/C++ applications will be eliminated (if not reversed) by the 2010 timeframe.

Partitioned Global Address Space. A key limitation in using the JAVA language on NUCC systems is that its programming model is tied to the notion of a *single uniform heap*. Past experience with shared virtual memory systems and cluster virtual machines *e.g.,* [3], has revealed significant scalability problems in trying to automatically map a uniform heap onto a non-uniform cluster. Thus, we decided to introduce a partitioned global address space in X10, with explicit reification of locality in the form of *places* (Section 3.3). Places address our scalability goal by enabling X10 programmers to control which objects and activities are co-located in a NUCC system. Other PGAS languages that have been developed in the past include Titanium [30], UPC [19] and Co-Array Fortran [44].

Focus on dynamic asynchronous activities. Another limitation of the JAVA language is that its mechanisms for intra-node parallelism (threads) and inter-node parallelism (messages and processes) are too heavyweight and cumbersome for programming large-scale NUCC systems. We decided to introduce the notion of *asynchronous activities* as a foundation for lightweight "threads" that can be created locally

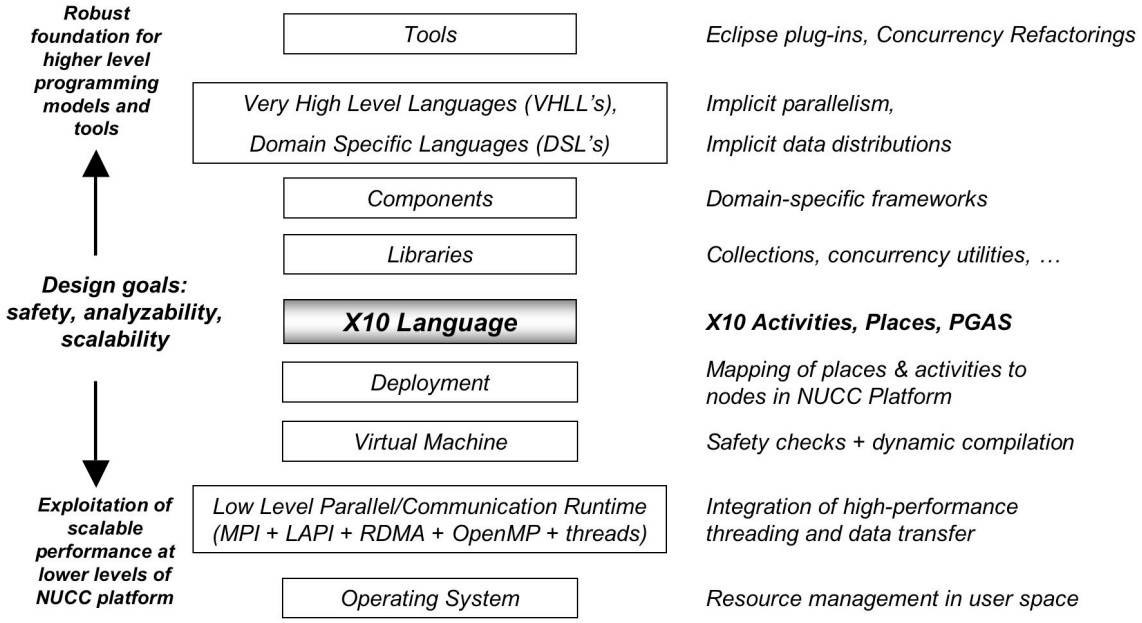

Figure 1: Position of X10 Language in Software Stack for NUCC Systems

or remotely. Asynchronous activities address the requirements of both thread-based parallelism and asynchronous data transfer in a common framework. When created on a remote place, an asynchronous activity can be viewed as a generalization of *active messages* [55]. A number of language constructs have been developed to create and coordinate asynchronous activities in an X10 program — *async*, *future*, *foreach*, *ateach*, *finish*, *clocks*, and *atomic blocks* (Section 3).

An array sub-language. We decided to include a rich array sub-language that supports dense and sparse, distributed, multi-dimensional arrays. An X10 *array object* (Section 3.4) is a collection of multiple elements that can be indexed by multi-dimensional *points* that form the underlying *index region* for the array. An array's *distribution* specifies how its elements are distributed across multiple places in the PGAS — this distribution remains unchanged throughout the program's execution.

While the previous design decisions are applicable to general-purpose programming in multiple domains, this decision was specifically targeted to high-performance computing domains which include a significant amount of array processing.

2.3 Methodological Issues

Additionally, we adopted a few methodological principles which ended up having a significant technical impact:

Focus. Identify the core issues to be tackled, solve them completely and cleanly, declare other problems, regardless of their technical attraction, out of scope.

In any new OO programming language design effort the temptation to tackle all that is broken in current languages is very strong. Because of limited resources we explicitly adopted a layered design approach. Several programming language issues deemed not to be at the core were postponed for future work even though they were of significant technical interest to members of the language team. These include: the design of a new module and component system (eliminating class-loaders); the design of new object-structuring mechanisms (e.g. traits, multi-dispatching); the design of a generic, place-based, type system integrated with dependent types (necessary for arrays); design for user-specified operator overloading; design of a new Virtual Machine layer; design of a weak memory model. We expect these and other related language issues to be revisited in subsequent iterations of the X10 language design (see Section 8.1). We expect to realize appropriate designs for these issues that would not require major revision of the core X10 model.

Build. Build early. Build often. Use what you build. We wrote a draft Language Report very early, and committed to building a prototype reference implementation as quickly as possible. This enabled us to gain experience with programming in the language, enabled us to run productivity tests with programmers who had no prior exposure to X10, and to bring application developers on board.

This methodological commitment forced us to reconsider initial technical decisions. We had originally intended to design and implement a technically sophisticated type system that would statically determine whether an activity was accessing non-local data. We soon realized that such a system would take much too long to realize. Since we had to quickly build a functional prototype, we turned instead to the idea of using runtime checks which throw exceptions (e.g. `BadPlaceExceptions`) if certain invariants associated with the abstract machine were to be violated by the current instruction.

This design decision leaves the door open for a future integration of a static type system. It regards such a type system as helping the programmer obtain static guarantees

that the execution of the program "can do no wrong", and providing the implementation with information that can be used to execute the program more efficiently (e.g. by omitting runtime checks).

3. X10 PROGRAMMING MODEL

This section provides an overview of the X10 programming model, using example code fragments to illustrate the key concepts. Figure 2 contains a schematic overview of the X10 programming model based on the five major design decisions outlined in the previous section. Briefly, X10 may be thought of as the JAVA language 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.

3.1 Places

Figure 2 contains a schematic overview of places and activities in X10. A *place* is a collection of *resident* (non-migrating) mutable data objects and the activities that operate on the data. Every X10 activity runs in a place; the activity may obtain a reference to this place by evaluating the constant `here`. The set of places are ordered and the methods `next()` and `prev()` may be used to cycle through them.

X10 0.41 takes the conservative decision that the number of places is fixed at the time an X10 program is launched Thus there is no construct to create a new place. This is consistent with current programming models, such as MPI, UPC, and OpenMP, that require the number of processes to be specified when an application is launched. We may revisit this design decision in future versions of the language as we gain more experience with adaptive computations which may naturally require a hierarchical, dynamically varying notion of places.

Places are virtual — the mapping of places to physical locations in a NUCC system is performed by a *deployment* step (Figure 1) that is separate from the X10 program. Though objects and activities do not migrate across places in an X10 program, an X10 deployment is free to migrate places across physical locations based on affinity and load balance considerations. While an activity executes at the same place throughout its lifetime, it may dynamically spawn activities in remote places as illustrated by the *outbound activities* in Figure 2.

3.2 Asynchronous Activities

An X10 computation may have many concurrent *activities* "in flight" at any given time in multiple places. An asynchronous activity is created by a statement `async (P) S` where `P` is a place expression and `S` is a statement. The statement `async S` is treated as shorthand for `async(here) S`, where `here` is a constant that stands for the place at which the activity is executing.

X10 requires that any method-local variable that is accessed by more than one activity must be declared as `final`[1]. This permits an implementation to copy the value (if necessary) when spawning a child activity, and maintain the invariant that the state of the stack remains private to the activity. Yet it is expressive enough to permit the determinate transmission of information from a parent activity to its children activities.

[1]This restriction is similar to the constraint that must be obeyed in the body of methods of inner classes in JAVA.

EXAMPLE 1 (COPY) *Consider the following piece of code, executing at some place* `i` *by activity* `A0`*. It is intended that at Line* `7` `t.val` *contains a reference to an object allocated at another place (specifically, place* `i+1`*).*

Line `1` *stores in the local variable* `Other` *a reference to the next place. Line* `2` *creates a new* `T` *object at the current place, and loads a reference into the local variable* `t`*. Line* `3` *spawns a new activity (call it* `A1`*) at* `Other`*, and waits until it finishes (Section 3.2.1).* `A1` *creates a new object (which must be located in the same place as* `A1`*, i.e. at* `Other`*), and stores it in a* `final` *local variable* `t1`*.* `A1` *may refer to the variable* `t` *(Line* `5`*) in a lexically enclosing scope (even though* `t` *is located at a different place) because* `t` *is declared to be* `final`*.* `A1` *spawns a new activity (call it* `A2`*) at the place where* `t` *is located. This activity is permitted to read* `t1` *because* `t1` *is declared* `final`*.* `A2` *writes a reference to* `t1` *into* `t.val` *(a field located in the same place).*

```
final place Other = here.next();  //1
final T t = new T();              //2
finish async (Other){             //3
  final T t1 = new T();           //4
  async (t) t.val = t1;           //5
}                                 //6
                                  //7
```

Note that creating X10 activities is much simpler than creating JAVA threads. In X10, it is possible for multiple activities to be created in-line in a single method. Of course the body of an activity may specify arbitrary computation, not just a single read or write; this leads to considerable flexibility in combining computation with communication. In contrast, the JAVA language requires that a thread be separated out into a new class with a new `run()` method when adding a new thread.

3.2.1 Activity Termination, Rooted Exceptions, and Finish

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 the 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 (if any) have, recursively, terminated globally.

An activity may terminate *normally* or *abruptly*. A statement terminates abruptly when it throws an exception that is not handled within its scope; otherwise it terminates normally. The semantics of abrupt termination is straightforward in a serial context, but gets complicated in a parallel environment. For example, the JAVA language propagates exceptions up the call stack until a matching handler is found within the thread executing the statement terminated abruptly. If no such handler is found, then an `uncaughtException` method is invoked for the current thread's `ThreadGroup`. Since there is a natural parent-child relationship between a thread and a thread that it spawns in all multi-threaded programming languages, it seems desirable to propagate an exception to the parent thread. However, this is problematic because the parent thread continues

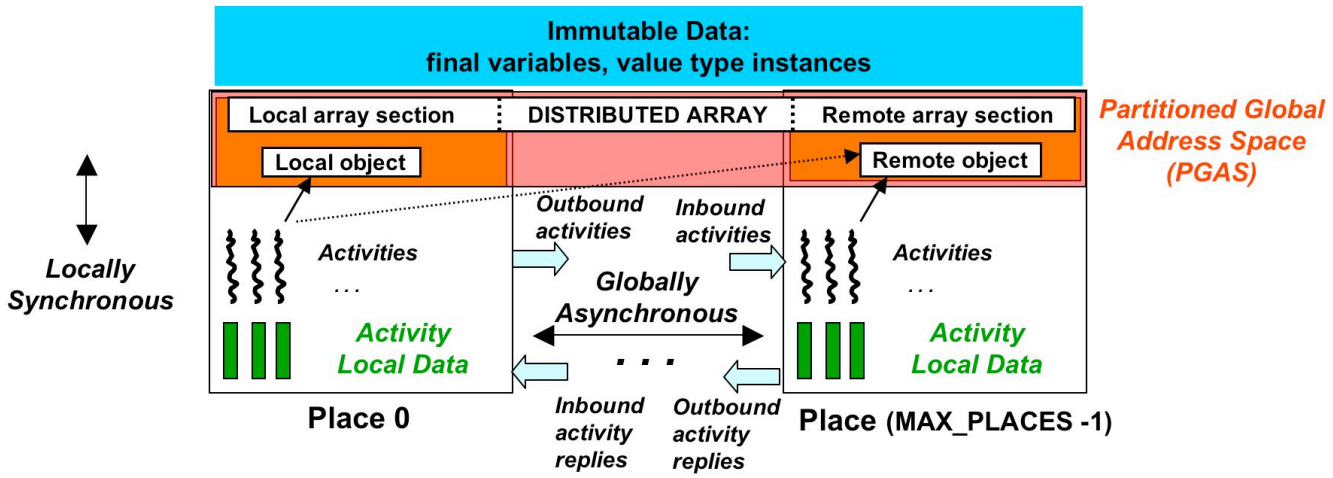

Figure 2: Overview of X10 Activities, Places and Partitioned Global Address Space (PGAS)

execution in parallel with the child thread, and may terminate prior to the child thread. Therefore it may not be available to catch exceptions thrown by the child thread.

The statement `finish S` in X10 converts global termination to local termination. `finish S` terminates locally (and globally) when `S` terminates globally. If `S` terminates normally, then `finish S` terminates normally and *A* continues execution with the next statement after `finish S`. Because `finish` requires the parent activity to suspend, it is also a very natural collection point for exceptions thrown by children activities. X10 requires that if `S` or an activity spawned by `S` terminates abruptly, and all activities spawned by `S` terminate, then `finish S` terminates abruptly and throws a single exception formed from the collection of all exceptions thrown by `S` or its spawned activities.

There is an implicit `finish` statement surrounding the main program in an X10 application. Exceptions thrown by this statement are caught by the runtime system and result in an error message printed on the console.

Example 2 (Copy, revisited) *Consider Example 1. The* `finish` *on Line* `3` *ensures global termination of both* `A1` *and* `A2`*; thus under normal execution* `A0` *advances to Line* `7` *only after* `t.val` *is not null.* `A1` *may terminate abruptly, e.g. with an* `OutOfMemoryException` *thrown at Line* `4`*. This will cause* `A0` *to terminate abruptly with an exception thrown at Line* `3`*;* `A0` *will not progress to Line* `7`*.*

3.3 Partitioned Global Address Space

X10 has a global address space. This means that it is possible for any activity to create an object in such a way that any other activity has the potential to access it.[2] The address space is said to be *partitioned* in that each mutable location and each activity is associated with exactly one place, and places do not overlap.

A *scalar object* in X10 is allocated completely at a single place. In contrast, the elements of an *array* (Section 3.4) may be distributed across multiple places.

X10 supports a *Globally Asynchronous Locally Synchronous* (GALS) semantics [12] for reads/writes to mutable locations. Say that a mutable variable is *local* for an activity if it is located in the same place as the activity; otherwise it is *remote*. An activity may read/write only local variables (this is called the *Locality Rule*), and it may do so synchronously. Any attempt by an activity to read/write a remote mutable variable results in a `BadPlaceException`. Within a place, activities operate on memory in a *sequentially consistent* fashion [38], that is, the implementation ensures that each activity reads and writes a location in one indivisible step, without interference with any other activity.[3] However, an activity may read/write remote variables only by spawning activities at their place. Thus a place serves as a *coherence boundary* in which all writes to same datum are observed in the same order by all activities in the same place. In contrast inter-place data accesses to remote variables have weak ordering semantics. The programmer may explicitly enforce stronger guarantees by using sequencing constructs such as `finish`, `force` (Section 3.8) or `clocks` (Section 3.6).

Example 3 (Copy, once again) *Consider Example 1 again. Both the* `async`*s are necessary to avoid a* `BadPlaceException` *in any execution of the program with more than one place.*

3.4 Arrays, Regions and Distributions

An array is associated with a (possibly multi-dimensional) set of *index points* called its *region*. 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. A *distribution* specifies a place for each point in the region. Several built in distributions are provided in X10, e.g. the constant distribution, a `block` distribution, a `blockCyclic` distribution etc. Various

[2] In contrast, MPI processes have a local address space. An object allocated by an MPI process is private to the process, and must be communicated explicitly to another process through two-sided or one-sided communications.

[3] As outlined in Section 3.7, *atomic blocks* are used to ensure atomicity of groups of intra-place read/write operations.

general combinators are also provided that create new regions from existing regions (e.g. set union, intersection, difference), and new distributions from existing distributions and regions.

The region and distribution associated with an array do not change during its lifetime. Regions and distributions are first-class constructs that can also be used independently of arrays.

Syntactically, an array type is of the form `T[.]` where `T` is the base type, and `[.]` is the multi-dimensional, distributed array constructor.[4] X10 permits `value` arrays (in analogy with `value` classes, Section 3.9), arrays whose elements cannot be updated; such arrays are of type `T value [.]` and must be initialized at creation time. When allocating an `[.]` X10 array, `A`, it is necessary to specify its distribution. The distribution and region associated with an array are available through the fields `.region` and `.distribution`. New arrays can be created by combining multiple arrays, performing pointwise operations on arrays (with the same distribution), and by using *collective* operations (e.g. `scan`) that are executed in parallel on existing (distributed) arrays to create new (distributed) arrays.

EXAMPLE 4 *The following* X10 *statement*

```
int value [.] A = new int value[ [1:10,1:10] ]
                (point[i,j]) { return i+j; } ;
```

creates an immutable 10×10 *array of* `int`*s such that element* `A[i,j]` *contains the value* `i+j`.

3.5 For, Foreach, and Ateach

Points, regions and distributions provide a robust foundation for defining three *pointwise iterators* in X10: 1) pointwise `for` for sequential iteration by a single activity, 2) pointwise `foreach` for parallel iteration in a single place, and 3) pointwise `ateach` for parallel iteration across multiple places.

The statement `for (point p : R) S` supports sequential iteration over all the points in region `R`, by successively assigning their values to index variable `p` in their *canonical lexicographic order*. The scope of variable `p` is restricted to statement `S`. When an iteration terminates locally and normally, the activity continues with the next iteration; the `for` loop terminates locally when all its iterations have terminated locally. If an iteration throws an exception, then the `for` statement also throws an exception and terminates abruptly.

A common idiom in X10 is to use pointwise iterators to iterate over an array's region:

```
for ( point p : A ) A[p] = f(B[p]) ;
```

(X10 permits a distribution or an array to be used after the `:` instead of a region; in both cases the underlying region is used.) The above code will throw an `ArrayIndexOutOfBounds` exception if `p` does not lie in `B`'s region, and a `BadPlaceException` if all the elements in `A` and `B` in the given region are not located `here`. However, it makes no assumption about the rank, size and shape of the underlying region. If needed, standard `point` operations can be used to extract individual index values from `p` *e.g.,* `p[0]` and `p[1]` return integer index values of the first and second dimensions of `p`.

The statement `foreach (point p : R) S` supports parallel iteration over all the points in region `R`. inally, the statement `ateach (point p : D) S` supports parallel iteration over all the points in a *distribution* `D`, by launching an iteration for each point `p` in `D.region` at the place designated by `D[p]`.

This is especially convenient for iterating over a distributed array *e.g.,*

```
ateach ( point [i,j] : A ) A[i,j] = f(B[i,j]) ;
```

3.6 Clocks

A commonly used construct in Single Program Multiple Data (SPMD) programs [15] is a *barrier*. For MPI programs, a barrier is used to coordinate all members of a *process group* in a *communicator* [51]. Two major limitations with this approach to coordinating processes are as follows:

1. It is inconvenient to perform MPI barrier operations on dynamically varying sets of processes. To do so, the user must explicitly construct each such set as a distinct process group.

2. It is possible to enter deadlock situations with MPI barriers, either when a process exits before performing barriers that other processes are waiting on, or when different processes operate on different barriers in different orders.

X10's *clocks* are designed to support a robust barrier-like functionality for varying sets of activities, while still guaranteeing determinate, deadlock-free parallel computation. At any step of program execution, a clock `C` is in a given *phase* and has a set of activities that are registered on `C`. Likewise, an activity may be registered on multiple clocks. An activity can perform a `next` operation to indicate that it is ready to advance *all* the clocks that it is registered on. When all activities that are registered on `C` enter this quiescent state, then clock `C` can be *advanced* and thereby enable all the activities registered on `C` to resume execution (assuming that they are not waiting on some other clock as well).

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, un-register 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.

3.7 Atomic Blocks

An *unconditional atomic block* is a statement `atomic S`, where `S` is a statement. X10 permits the modifier `atomic` on method definitions as a shorthand for enclosing the body of the method in `atomic`.

An atomic block is executed by an activity *as if* in a single step during which all other concurrent activities in the same place are suspended. Compared to user-managed locking as in the `synchronized` constructs in the JAVA language, the X10 user only needs to specify that a collection of statements should execute atomically and leaves the responsibility of lock management and other mechanisms for enforcing atomicity to the language implementation. Commutative

[4] For simplicity, X10 v0.41 permits the expression `T[]` as a type whose elements are all the arrays of `T` defined over the single-dimensional region `0:n-1` (for some integer `n`), all of whose points are mapped to the place `here`.

operations, such as updates to histogram tables and insertions in a shared data structure, are a natural fit for atomic blocks when performed by multiple activities.

An atomic block may include method calls, conditionals, and other forms of sequential control flow. From a scalability viewpoint, it is important to avoid including blocking or asynchronous operations in an atomic block.

Example 5 (Linked List)

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

By declaring the block as atomic, the user is able to maintain the integrity of a linked list data structure in a multithreaded context, while avoiding irrelevant details such as locking structure.

An X10 implementation may use various techniques (e.g. non-blocking techniques) for implementing atomic blocks [29, 41, 49]. For simplicity of implementation, X10 specifies isolation only during normal execution. If `S` terminates abruptly, then `atomic S` terminates abruptly. It is the responsibility of the user to ensure that any side-effects performed by `S` are cleaned up.

Atomicity is guaranteed for the set of instructions actually executed in the atomic block. If the atomic block terminates normally, this definition is likely to coincide with what the user intended. If the atomic block terminates abruptly by throwing an exception, then atomicity is only guaranteed for the instructions executed before the exception is thrown. If this is not what the user intended, then it is their responsibility to provide exception handlers with appropriate compensation code. However, compared to user-managed locking in other languages, the X10 user need not worry about guaranteeing atomicity in the presence of multiple locks, since lock management is the responsibility of the X10 system and is hidden from the user who programs at the level of atomic blocks and other high level constructs.

X10 guarantees that any program written using the parallel constructs that have been described so far (`async`, `finish`, `foreach`, `ateach`, `clock`, `atomic`) will never deadlock [47].

3.7.1 Conditional atomic blocks

X10 also includes a powerful *conditional* atomic block, `when (c) S`. Execution of `when (c) S` suspends until a state is reached in which the condition `c` is true. In this state the statement `S` is executed atomically. `atomic S` can be seen as the special case `when (true) S`. This construct is closely related to the conditional critical regions of [31, 26]. All other synchronization constructs can be implemented in terms of conditional atomic blocks. However, the unrestricted use of this construct can cause deadlock. Further we have not yet encountered programming idioms in the high performance computing space which require the full power of conditional atomic blocks. Therefore while this construct is in the language, its use is currently deprecated.

3.8 Asynchronous Expression and Future

We briefly describe futures, as they are incorporated in X10. When an activity `A0` executes the expression `future(P) E`, it spawns an activity `A1` at place `P` to execute the expression `E`, and terminates locally yielding a future [25], `F`. When A desires to access the result of computing `E` it executes a `.force()` operation on `F`: this may block if the value has not yet been computed. Like `async` statements, `future`s can be used as the foundation for many parallel programming idioms including asynchronous DMA operations, message send/receive, and scatter/gather operations.

3.9 Scalar Reference and Value Classes

An X10 scalar class has fields, methods and inner types (interfaces or classes), subclasses another class (unless it is the root class `x10.lang.Object`, and implements zero or more interfaces. X10 classes live in a single-inheritance code hierarchy with root `x10.lang.Object`. There are two kinds of scalar classes: *reference* classes and *value* classes.

A reference class can contain 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 must not have updatable fields (defined directly or through inheritance), and allows no reference subclasses. It is declared in the same way as a reference class, with the keyword `value` used in place of `class`. Fields of value classes may 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 (with copies bottoming out on fields of reference type). 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.

4. X10 EXAMPLES

In this section, we use sample programs to illustrate X10's features to implement concurrent and distributed computations, and contrast the resulting programs with the patterns employed in existing multithreaded and message-passing programming models.

4.1 MonteCarlo

The MonteCarlo benchmark [35] illustrates how X10's high-level threading features enable a seamless transition from a sequential to a multi-threaded program with data parallelism.

Figure 3 shows two variants of the main computation loop in the program, first the serial Java version, then the parallel X10 version. The difference between the serial and the parallel version is merely that the sequential `for` loop is replaced by the parallel `foreach` construct available in X10. Note that `Vector` is an implementation of a collection that employs locks in the Java version (`synchronized`) and atomic methods in the X10 version to achieve thread-safety.

The multithreaded Java version of the benchmark (which we do not show here due to space constraints, see [35]) is quite different: it is far more complicated than the serial Java version (38 lines instead of 10 lines) because it involves defining a new `Runnable` class for the thread instances, explicit thread creation, and the slicing of the iteration space across the thread instances.

```
Serial Java version:
--------------------

public void runSerial() {
  results = new Vector(nRunsMC);
  PriceStock ps;
  for (int iRun=0; iRun < nRunsMC; iRun++) {
    ps = new PriceStock();
    ps.setInitAllTasks((ToInitAllTasks) initAllTasks);
    ps.setTask(tasks.elementAt(iRun));
    ps.run();
    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 3: Comparison of JAVA vs. X10 codes for MonteCarlo benchmark

4.2 SPECjbb

A frequently used pattern in concurrent programming is that of a phased computation, where threads 'meet' repeatedly at synchronization points (e.g., to exchange intermediate results) but operate independently during the rest of the computation.

The SPECjbb benchmark [54] is an example of such a phased computation. Figure 4 shows a fragment of the JAVA version of the benchmark, which was developed during 1996–2000 and hence uses basic thread primitives rather than the more recent JAVA concurrency utilities [40].

There are two classes of activities in SPECjbb, the master activity, which controls the overall program execution along a sequence of operation modes, and one or several warehouse/terminal threads that issue requests to a simulated database and business logic. The master thread starts the warehouse threads and repeatedly synchronizes with them to communicate the current operation mode.

In the JAVA version, the current operation mode is explicitly communicated through a shared global variable (`Company.mode`) from the master thread to the warehouses. The warehouses in turn communicate their progress (initialized, stopped) to the master thread through global counter variables (`Company.initThreadsCount` and `Company.stopThreadsCount`). Access to these global counters must occur in an appropriate `synchronized` block to avoid inconsistency due to concurrent conflicting updates. Coordination among the master and warehouse threads is accomplished through `wait/notify` primitives. The implementation is quite sophisticated because the two classes of threads wait on different condition variables. Each invocation of `wait` must be matched by a call to `notify/notifyAll` on the same condition variable (see arrows in Figure 4); the synchronization necessary to communicate the ramp-down of all warehouse threads to the master is implemented similarly.

In contrast, the implementation in X10 (Figure 5) just uses a single *clock* to coordinate the work being done by the master and the warehouse/terminal activities. A `next` operation is used to enforce the first phase transition which marks the end of the initializations performed by warehouse activities (upper arrow in Figure 5). Similarly, a second `next` operation enforces the end of processing in each warehouse activity after the ramp-down mode is set (lower arrow, phase 2 in Figure 5). Note that the master and warehouse/terminal activities synchronize through the same clock, and that the counters that had to be explicitly managed in the JAVA implementation are implicit in the use of clocks in X10.

4.3 RandomAccess

While the previous examples focused on single-place parallelism, we use the RandomAccess HPC Challenge benchmark [32] to demonstrate that a distributed computation in X10 can be implemented as an incremental extension of a serial program. This is in contrast to approaches that distribute and communicate data according to the message passing model, where the implementation and algorithm of a distributed program can significantly differ from the parallel or the sequential program.

Figure 4.3 outlines an X10 implementation of the RandomAccess HPC Challenge benchmark [32]. The key X10 features used in this example are *distributed arrays*, `ateach`, and collective operations (array sum). First, a large global array is allocated with a in a *block* distribution across all places; this global array is referenced through variable `Table`. The initialization code executes in parallel for each array element at its appropriate place (akin to the `ateach` construct). `SmallTable` refers to a *value array*; such arrays can be replicated at each place since value arrays are immutable.

The computational kernel of RandomAccess is a long running sequential loop which is executed by one activity in each place using the `ateach` construct. The iteration space of the `ateach` construct is a *unique distribution*, which associates a single index value with each place. Each iteration initiates an atomic read-xor-write operation on a randomly selected element, `Table[j]`, in the global array by creating an `async` activity at place `Table.distribution[j]` to ensure that the Locality Rule is satisfied. The `finish` operation ensures global termination of all `ateach` and `async` activities before the `sum` operation is performed. X10's standard library includes several standard distributions (like `block`) and collective operations (like `sum`) that make it convenient to operate on global arrays.

This example illustrates the use of asynchronous activities and of partitioned global data in X10. In contrast, the MPI version of RandomAccess [32] is far more complicated because it needs to manage variable numbers of nonblocking message sends and receives on each processor to support the random communication patterns in this application. The

Company.java:

```
public void displayResultTotals(boolean forceGC)
{
   short warehouseId;
   TimerData warehouseTimerData;

   // Wait for all threads to start.
   synchronized (initThreadsStateChange) {
      while (initThreadsCount !=
         MaxWarehouses*MaxTerminalsPerWarehouse) {
         try { initThreadsStateChange.wait(); }
         catch (InterruptedException e) { }
      }
   }
   // Tell everybody it's time for warmups.
   mode = RAMP_UP;
   synchronized(initThreadsCountMonitor) {
      initThreadsCountMonitor.notifyAll();
   }
   . . .
   mode = RECORDING;
   . . .
   mode = RAMP_DOWN;
   . . .
   synchronized(threadsDoneCountMonitor) {
      while (threadsDoneCount !=
         MaxWarehouses*MaxTerminalsPerWarehouse) {
         try { threadsDoneCountMonitor.wait(); }
         catch (InterruptedException e) { }
      }
   }

   mode = STOP;
   synchronized(stopThreadsCountMonitor) {
      while (stopThreadsCount !=
         MaxWarehouses*MaxTerminalsPerWarehouse) {
         try {stopThreadsCountMonitor.wait();}
         catch (InterruptedException e) { }
      }
   }
   . . .
}
```

TransactionManager.java:

```
public void go()
{
   . . .
   synchronized (company.initThreadsCountMonitor) {
      synchronized (company.initThreadsStateChange) {
         company.initThreadsCount++;
         company.initThreadsStateChange.notify();
      }
      try { company.initThreadsCountMonitor.wait(); }
      catch (InterruptedException e) { }
   }

   while (company.mode != Company.STOP) {
      txntype = ... ;
      . . .
   } // end while

   if (timed && (company.mode == Company.STOP)) {
      elapsed_time = company.getElapsedTime();
      myTimerData.calculateResponseTimeStats();
      . . .
      synchronized (company.stopThreadsCountMonitor) {
         company.stopThreadsCount++;
         company.stopThreadsCountMonitor.notify();
      }
   }
}
```

Figure 4: Outline of JAVA version of SPECjbb

```
finish async {
   final clock c = new clock();
   final Company company = createCompany(...);
   foreach (point[w] : [0:wh_num])
     clocked(c) { // warehouse activity
        int mode;
        atomic { mode = company.mode; }
        initialize;
        next; // 1.
        while (mode != STOP) {
           select a transaction;
           think;
           process the transaction;
           if (mode == RECORDING)
              record data;
           if (mode == RAMP_DOWN)
              next; // 2.
           atomic { mode = company.mode; }
        } // while
   } // foreach

   // master activity contd.
   next; // 1.
   atomic { company.mode = RAMP_UP; }
   sleep rampuptime;
   atomic { company.mode = RECORDING; }
   sleep recordingtime;
   atomic { company.mode = RAMP_DOWN; }
   next; // 2.
   // all clients in RAMP_DOWN
   atomic { company.mode = STOP; }
} // finish async

// simulation completed.
print results.
```

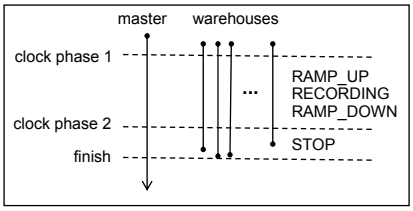

Figure 5: Outline of X10 version of SPECjbb

```
public boolean run() {
  long[.] Table = new long[dist.factory.block([0:TABLE_SIZE-1])] (point [i]) { return i; };
  long value[.] SmallTable = new long value[[0:S_TABLE_SIZE-1]] (point [i]) { return i*S_TABLE_INIT; };

  finish ateach (point [i]: dist.factory.unique()) {
    long ran = initRandom(N_UPDATES_PER_PLACE*i);
    for (point [count]: 1:N_UPDATES_PER_PLACE) {
      ran = nextRandom(ran);
      final int J = f(ran);
      final long K = SmallTable[g(ran)];
      async(Table.distribution[J]) atomic Table[J]^=K;
    }
  }
  return Table.sum() == EXPECTED_RESULT;
}
```

Figure 6: RandomAccess example in X10

MPI version uses nonblocking communications in the form of `MPI_Isend` and `MPI_Irecv` calls, and completion operations in the form of `MPI_Test`, `MPI_Wait`, and `MPI_WaitAll` calls, in addition to hard-coding into the program the logic for distributing a global array across nodes.

5. PRODUCTIVITY ANALYSIS — PARALLELIZING SERIAL CODE

This section presents a 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. Additional studies are being planned at IBM to assess other productivity aspects of X10 for different application work flows *e.g.,* writing new parallel code from scratch, porting from other languages to X10, and parallel programming with refactoring tools.

This assessment was performed using the publicly available Java Grande Forum Benchmark Suite [35] which contains benchmarks with source code available in three versions — serial, multithreaded [52] and message-passing based on mpiJava [53]. The key metric used to measure effort is *code size*. In general, code size metrics are 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 size will require more effort to develop and maintain, but there is no guarantee that a piece of code with larger size has more functionality than a piece of code with smaller size. This limitation of code size as a metric is not an issue for this productivity analysis since the functionality of the code is standardized by the characteristics of the benchmark.

For robustness, we use two metrics for measuring code size:

1. Source Lines Of Code (SLOC) — the number of *non-comment non-blank lines* in the source code of the program.

2. Source Statement Count (SSC) — the number of *syntactic statements* in the source program.

The SLOC metric is used widely, but is prone to variation based on coding styles such as placement of braces and of multiple statements in the same line. We use the SSC metric to guard against these variations. For X10, Java, and other languages with similar syntax, there are two rules for counting syntactic statements: 1) any sequence of non-semicolon lexical tokens terminated by a semicolon is counted as a statement (this includes variable declarations and executable statements), 2) a statement header is counted as a statement. Statement headers include compound statement headers (`if`, `while`, `for`, `foreach`, `ateach`, `switch`, etc.), `case`/`default` labels, class headers, interface headers, and method headers.

Consider the following code fragment as an example:

```
class Main {
  public static void main(String [] args)
  {
    int percentage = 0;
    if (args.length > 0)
    {
      percentage = 100 / args.length;
    }
    System.out.println("percentage = " +
                       percentage);
  }
}
```

The SLOC metric for this example is 12, but the SSC metric is only 6 since there are exactly 6 syntactic statements in the example:

1. `class Main`
2. `public static void main(String [] args)`
3. `int percentage = 0;`
4. `if (args.length > 0)`
5. `percentage = 100 / args.length;`
6. `System.out.println("percentage = " + percentage);`

This example exhibits a 2× gap between SLOC and SSC metrics. However, for the benchmarks studied in this paper, the SLOC metric is approximately 1.25× the SSC metric. This difference between SLOC and SSC is our main motivation for collecting both sets of metrics to ensure the robustness of our results. Fortunately, our results show that the code size *ratios* that are computed to compare different versions of the same code are approximately the same for SLOC metrics or SSC metrics.

Benchmark	Code size metric	Serial Java	Multithreaded Java	MPI Java
crypt	Classes/SLOC	3/255	4/310	3/335
(Section 2)	Total statement count (SSC)	206	252	268
	SSC ratio, relative to Serial version		1.22	1.30
	Statements deleted, relative to Serial version		50	11
	Statements inserted, relative to Serial version		96	73
	Change ratio, relative to serial		0.71	0.41
lufact	Total size (classes/SLOC)	3/317	6/502	3/449
(Section 2)	Total statement count (SSC)	252	399	351
	SSC ratio, relative to Serial version		1.58	1.39
	Statements deleted, relative to Serial version		36	13
	Statements inserted, relative to Serial version		183	112
	Change ratio, relative to serial		0.87	0.50
series	Total size (classes/SLOC)	3/131	4/175	3/204
(Section 2)	Total statement count (SSC)	91	128	146
	SSC ratio, relative to Serial version		1.41	1.60
	Statements deleted, relative to Serial version		14	10
	Statements inserted, relative to Serial version		51	65
	Change ratio, relative to serial		0.71	0.82
sor	Total size (classes/SLOC)	3/93	4/184	3/240
(Section 2)	Total statement count (SSC)	72	149	178
	SSC ratio, relative to Serial version		2.07	2.47
	Statements deleted, relative to Serial version		10	19
	Statements inserted, relative to Serial version		87	125
	Change ratio, relative to serial		1.35	2
sparsematmult	Total size (classes/SLOC)	3/95	4/185	3/169
(Section 2)	Total statement count (SSC)	80	156	139
	SSC ratio, relative to Serial version		1.95	1.74
	Statements deleted, relative to Serial version		9	18
	Statements inserted, relative to Serial version		85	77
	Change ratio, relative to serial		1.18	1.19
moldyn	Total size (classes/SLOC)	5/384	8/530	5/437
(Section 3)	Total statement count (SSC)	346	454	390
	SSC ratio, relative to Serial version		1.31	1.13
	Statements deleted, relative to Serial version		92	20
	Statements inserted, relative to Serial version		200	64
	Change ratio, relative to serial		0.84	0.24
montecarlo	Total size (classes/SLOC)	15/1134	16/1175	15/1195
(Section 3)	Total statement count (SSC)	892	926	940
	SSC ratio, relative to Serial version		1.04	1.05
	Statements deleted, relative to Serial version		14	9
	Statements inserted, relative to Serial version		48	57
	Change ratio, relative to serial		0.07	0.07
raytracer	Total size (classes/SLOC)	13/592	16/696	13/654
(Section 3)	Total statement count (SSC)	468	552	511
	SSC ratio, relative to Serial version		1.18	1.09
	Statements deleted, relative to Serial version		17	11
	Statements inserted, relative to Serial version		101	54
	Change ratio, relative to serial		0.25	0.14
jgfutil	Total size (classes/SLOC)	2/253	2/271	2/290
(Section 3)	Total statement count (SSC)	185	196	210
	SSC ratio, relative to Serial version		1.06	1.14
	Statements deleted, relative to Serial version		2	9
	Statements inserted, relative to Serial version		13	34
	Change ratio, relative to serial		0.08	0.23
TOTAL	Total size (classes/SLOC)	50/3254	64/4028	50/3973
(all benchmarks)	SLOC ratio, relative to Serial version		1.24	1.22
	Total statement count (SSC)	2592	3212	3133
	SSC ratio, relative to Serial version		1.24	1.21
	Total number of statements changed		1108	781
	Change ratio for statements, relative to Serial		0.43	0.30

Table 1: SLOC and SSC metrics for serial and parallel versions of benchmarks written in JAVA

Benchmark	Code size metric	Serial X10	Single-place Multi-activity X10	Multi-place Multi-activity X10
crypt	Total size (classes/SLOC)	3/246	3/246	3/248
(Section 2)	Total statement count (SSC)	197	198	201
	SSC ratio, relative to Serial version		1.00	1.02
	Statements deleted, relative to Serial version		8	23
	Statements inserted, relative to Serial version		9	27
	Change ratio, relative to serial		0.09	0.25
lufact	Total size (classes/SLOC)	3/304	3/314	3/316
(Section 2)	Total statement count (SSC)	246	263	264
	SSC ratio, relative to Serial version		1.07	1.07
	Statements deleted, relative to Serial version		74	77
	Statements inserted, relative to Serial version		91	95
	Change ratio, relative to serial		0.67	0.70
series	Total size (classes/SLOC)	3/134	3/138	3/151
(Section 2)	Total statement count (SSC)	98	98	108
	SSC ratio, relative to Serial version		1.00	1.10
	Statements deleted, relative to Serial version		5	19
	Statements inserted, relative to Serial version		5	29
	Change ratio, relative to serial		0.10	0.49
sor	Total size (classes/SLOC)	3/77	3/80	3/84
(Section 2)	Total statement count (SSC)	66	68	73
	SSC ratio, relative to Serial version		1.03	1.11
	Statements deleted, relative to Serial version		2	5
	Statements inserted, relative to Serial version		4	12
	Change ratio, relative to serial		0.09	0.26
sparsematmult	Total size (classes/SLOC)	3/94	3/138	3/139
(Section 3)	Total statement count (SSC)	81	128	132
	SSC ratio, relative to Serial version		1.58	1.63
	Statements deleted, relative to Serial version		27	29
	Statements inserted, relative to Serial version		74	80
	Change ratio, relative to serial		1.25	1.35
moldyn	Total size (classes/SLOC)	5/376	5/420	5/419
(Section 3)	Total statement count (SSC)	337	380	403
	SSC ratio, relative to Serial version		1.13	1.20
	Statements deleted, relative to Serial version		7	7
	Statements inserted, relative to Serial version		50	73
	Change ratio, relative to serial		0.17	0.24
montecarlo	Total size (classes/SLOC)	14/1080	14/1080	14/1080
(Section 3)	Total statement count (SSC)	857	857	857
	SSC ratio, relative to Serial version		1	1
	Statements deleted, relative to Serial version		1	2
	Statements inserted, relative to Serial version		1	2
	Change ratio, relative to serial		0.00	0.00
raytracer	Total size (classes/SLOC)	13/534	13/534	13/544
(Section 3)	Total statement count (SSC)	410	411	420
	SSC ratio, relative to Serial version		1	1.02
	Statements deleted, relative to Serial version		2	10
	Statements inserted, relative to Serial version		3	20
	Change ratio, relative to serial		0.01	0.07
jgfutil	Total size (classes/SLOC)	2/262	2/262	2/262
(Section 3)	Total statement count (SSC)	191	191	191
	SSC ratio, relative to Serial version		1	1
	Statements deleted, relative to Serial version		0	0
	Statements inserted, relative to Serial version		0	0
	Change ratio, relative to serial		0	0
TOTAL	Total size (classes/SLOC)	49/3107	49/3212	49/3243
(all benchmarks)	SLOC ratio, relative to Serial version		1.03	1.04
	Total statement count (SSC)	2483	2594	2649
	SSC ratio, relative to Serial version		1.04	1.07
	Total number of statements changed		363	510
	Change ratio for statements, relative to Serial		0.15	0.21

Table 2: Classes and SLOC changed, added and deleted to obtain parallel versions of a serial X10 program

Table 1 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 [35] for which all three versions are available — `crypt`, `lufact`, `series`, `sor`, `sparsematmult`, `moldyn`, `montecarlo`, and `raytracer`. In addition, `jgfutil` contains a common library that is shared by all eight benchmarks. Five of the eight benchmarks come from Section 2 of the benchmark suite, which contains kernels that are frequently 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.

Table 1 displays the SLOC and SSC code size metrics for the serial, multithreaded, and MPI Java versions of the eight benchmarks and the `jgfutil` library. An aggregate of the metrics is shown at the bottom of the table. In addition to the totals for the SLOC and SSC metrics, a *SLOC ratio* and an *SSC ratio* are included to capture the expansion in source code size that occurs when going from the serial JAVA version to the multithreaded and MPI JAVA versions. For the SSC metrics, we also report the number of statements *deleted* and *inserted*, when comparing the serial version with the multithreaded version and the MPI version. These numbers contribute to a *change ratio* for SSC metrics, which is defined as (# insertions + # deletions)/(original size). The change ratio helps identify cases where significant changes may have been performed, even if the resulting code size expansion ratio is close to 1. In general, we see that the use of JAVA's multithreaded programming model for parallelization increased the code size by approximately 24% compared to the serial Java version, and the use of MPI Java increased the code size by approximately 21% - 22%. (These expansion factors are almost identical for both SLOC and SSC metrics.) Using Java threads also led to 14 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/SSC increase (compared to the serial version) for a combination of both the threading and Java MPI programming models will approximately equal the sum of the individual increases *i.e.,* approximately 24% + 21% = 45%. Finally, the SSC *change ratios* at the bottom of the table adds up to approximately 43% + 30% = 73% for both models, indicating that the extent of code change is larger than just the code size increase.

In contrast, Table 2 displays the same metrics for serial, single-place multi-activity, and multi-place multi-activity versions of the eight benchmarks translated to X10. All X10 versions reported in the table were executed and validated using the X10 reference implementation described in Section 6. The serial X10 version is usually very similar to the serial Java version. Comparing Table 2 with Table 1, we see that the multi-place multi-activity case for X10 increased the code size by 4% to 7% depending on whether SLOC metrics or SSC metrics are used. This is less than one-third of the code size increase observed in the JAVA case. In addition, the change ratio is approximately 15% to 23% for X10, compared to 30% to 43% for JAVA. Of course, the comparison with JAVA becomes more significant when compared with the estimated 73% increase for combining Java threads and Java MPI.

These results underscore the fact that the effort required for writing parallel versions of an X10 application is significantly smaller than the effort required for writing multithreaded and MPI versions of a JAVA application.

6. X10 IMPLEMENTATION

We have two implementations of X10 in progress. Section 6.1 describes the *reference implementation*, and Section 6.2 outlines our plans for the *scalable optimized implementation*.

6.1 Reference Implementation

We have constructed a reference implementation that provides a prototype development environment for X10 which can be used to obtain feedback on the language design through experimentation and productivity assessments of sample X10 programs. Programs executing on this implementation can emulate multiple places on a single node.

The reference implementation supports all language constructs presented in Section 3. As shown in Figure 7, the core components of the X10 reference implementation consist of the following:

1. A *source-to-source translator* from the X10 language to the JAVA language, obtained by modifying and extending the Polyglot [45] infrastructure. The translator consists of an X10 *parser* (driven by the X10 *grammar*), *analysis passes*, and a *code emitter*. The analysis passes and code templates are used to implement the necessary static and dynamic safety checks required by the X10 language semantics. The output JAVA program is compiled into classfiles using the standard `javac` compiler in the JDK.

2. An X10 *execution environment*, which consists of a JAVA compiler and runtime environment, an X10 multithreaded *runtime system (RTS)*, and efficient dynamic linkage with extern code.

3. A Performance and Environment Monitoring (PEM) system [56] that (optionally) generates two kinds of PEM output events from an X10 program execution — summary abstract performance metrics, and trace events that can be visualized by the Performance Explorer [28] tool.

The X10 RTS is written in the JAVA programming language and thus can take advantage of object oriented language features such as garbage collection, dynamic type checking and polymorphic method dispatch.

The JAVA bytecodes generated from the X10 program execute on an unmodified JVM in the X10 *Execution Environment*. This environment contains the X10 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 characteristics of an application using PEM. For example, the number of activities initiated remotely can be reported

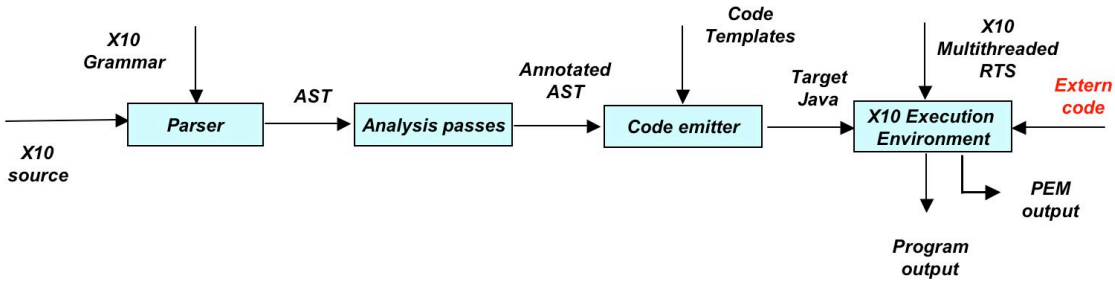

Figure 7: Structure of X10 Reference Implementation

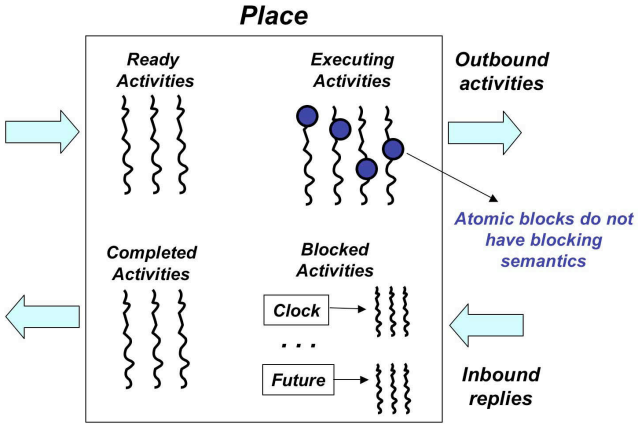

Figure 8: Structure of a single X10 place

for each place. This kind of data provides valuable information for tuning load balance and data distribution, independent of the performance characteristics of the target machine. Lower-level tuning for a given parallel machine will only be meaningful for the optimized implementation outlined in the next section.

The reference implementation operates in a single node and multiple X10 places are supported in a single JVM instance executing on that node. The design of a place corresponds to the *executor* pattern [39]. A place acts as an executor that dispatches individual activities to a pool of Java threads, as outlined in Figure 8. Activities in X10 can be created by async statements, future expressions, foreach and ateach constructs, and array initializers. 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 until 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 scalable optimized implementation, we are exploring techniques that will store the execution context of an activity and thereby bound the number of threads in a single place.

Conditional and unconditional atomic blocks are implemented using a single mutual exclusion lock per place. While this implementation strategy limits parallelism, it is guaranteed to be deadlock-free 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 the Java language. An X10 multi-dimensional array is flattened and represented internally as a combination of an X10 array descriptor object and a single-dimensional Java array [42]. 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.

X10 also provides for convenient and efficient interoperability with native code via the `extern` declaration. 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 does not allow direct manipulation of Java objects, and so passing a large array to a native library requires either copying the array within some native wrapper code, or modifying the library to use the JNI API to manipulate the array. X10 allows direct manipulation of arrays from native code, and simplifies the interface. An array descriptor is also passed to the native code so that multi-dimensional arrays can be handled if required. Any array that is used by an `extern` call has its memory pinned so it will not be moved by garbage collection.

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.

6.2 Scalable Optimized Implementation

In this section, we outline our plans for a multi-VM multi-node implementation that can be used to deliver scalable performance on NUCC systems, using IBM's high performance communication library, LAPI [10]. There are a number of design issues that arise when constructing such a multi-VM implementation. For example, the standard memory management and garbage collection implementations for Java will need to be enhanced for X10, to take into account the fact that object references can cross cluster boundaries in a NUCC system. Most other PGAS implementations, such as UPC [19], place the memory management burden on the

programmer and hence need not worry about supporting garbage collection (GC). Also, the extensively distributed nature of X10 programs implies that the implementation must be engineered to exploit *Ahead Of Time* (AOT) compilation and packaging more often than is seen in typical JAVA systems.

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.

Global arrays are X10 arrays in the global address space. That is, there is a single canonical name for an array, but the actual storage for it is distributed across multiple places. When a global array is dereferenced (recall that only the local section may be directly operated on in an X10 program) the address is devirtualized and mapped to local storage at the specific place where the dereference occurs. *Fat Pointers* are used as canonical handles for global objects such as these arrays. A fat pointer consists of a globally unique VM name (an integer) identifying the VM (or node within the cluster) where the global array's constructor was invoked, and a VM-unique (*i.e.,* unique within a given VM) name for the array. This allows for a unique name without using a central registry. Each place will provide a mapping from the fat pointer to the corresponding locally allocated storage.

Clocks are implemented as value objects with an internal reference to a mutable object o stored at the place at which the clock instance is created. Each place has a local copy of this value object, with a fat pointer to o. Communication is performed through the LAPI library.

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 complicates both bounds checking operations and GC, because the structure of objects containing nested header-less objects is more complicated.

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

The current implementation uses active messages transfer capabilities of the LAPI library [10]. Future implementations will make more effective use of these. In particular, the data involved in a large message will be stored in a memory region that does not interfere with frequent lightweight GC passes, 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). The compilation and runtime systems will be improved to aggregate LAPI calls in many cases. For example, when one place rapidly executes a series of asyncs at another place as it might in a loop, a customized active message that encapsulate the minor differences in program state between the large set of asynchronous activities will be used. As the language evolves and the implementation becomes more mature, bytecodes with X10 specific semantics will be added to the VM. Among other things, these will support future X10 features such as generics (which are more powerful than those available in current JAVA implementations.) They will also better support optimized implementations of existing X10 features such as atomic blocks [49].

7. RELATED WORK

X10 is a class-based, single-inheritance, mostly statically-typed object-oriented language that shares many common features with JAVA [24] and C# [18]. In particular, X10 adopts important safety properties from these languages, including a safe object model (objects are accessed through unforgeable reference) and automated memory management (garbage collection) to guarantee safe memory accesses. What distinguishes X10 from past object-oriented languages is the set of constructs available for managing concurrency and data distribution that enable X10 to target a wide range of parallel system configurations, including the NUCC systems discussed in Section 1. The following sections elaborate on the relationship between X10 and past work on parallel programming models and languages.

7.1 Language vs. Library

X10 follows, like, e.g., Split-C [37], CILK [14], HPF [20], Co-Array Fortran [44], Titanium [30], UPC [19], ZPL [11], Chapel [34], and Fortress [2], a *language-based* approach to concurrency and distribution, i.e., concurrency and distribution are controlled by first-class language constructs, rather than library extensions or pseudo-comment directives. The language-based approach facilitates the definition of precise semantics for synchronization constructs and the shared memory model [7] and enables a compiler to include concurrency related constructs in the static analysis and optimization. SHMEM [5], MPI [51], and PVM [23] in contrast, are library extensions of existing sequential languages, as are programming models for Grid computing [21]. The OpenMP [46] approach is based on pseudo-comment directives so that the parallelism can be enabled for platforms that support OpenMP, while the underlying program can still run correctly in sequential mode on other platforms by ignoring the pseudo-comment directives. While there are pragmatic reasons for pursuing an approach based on libraries or directives, our belief is that future hardware trends towards NUCC systems will have a sufficiently major impact on software to warrant introducing a new language.

7.2 Concurrency and Distribution

While HPF, ZPL, UPC and Titanium follow an SPMD style control-flow model where all threads execute the same code [15], CILK, X10, Chapel, and Fortress choose a more flexible model and allow more general forms of control flow among concurrent activities. This flexibility removes the implicit co-location of data and processing, that is characteristic of the SPMD model. X10's `async` and `finish` are conceptually similar to CILK's `spawn` and `sync` constructs. However, X10 is more general than CILK in that it permits

a parent activity to terminate while its child/descendant activities are still executing, thereby enabling an outer-level `finish` to serve as the root for exception handling and global termination. Fortress offers library support that enables the runtime system to distribute arrays and to align a structured computation along the distribution of the data. X10 is somewhat different, in that it introduces *places* as a language concept and abstraction used by the programmer to explicitly control the allocation of data and processing. A place is similar to a locale in Chapel and a site in Obliq [8]. Like X10, each object in Obliq is allocated and bound to a specific site and does not move; objects are accessed though "network references that can be transmitted from site to site without restrictions" [8]. In Obliq, arrays cannot be distributed across multiple sites. Chapel's model of allocation is different from X10 and Obliq because Chapel does not require that an object be bound to a unique place during its lifetime or that the distribution of an array remain fixed during its lifetime.

One of the key difference between X10 and earlier distributed shared object systems such as Linda [1], Munin [6], and Orca [4] is that those systems presented a uniform data access model to the programmer, while using compile-time or runtime techniques under the covers to try and address the locality and consistency issues of the underlying distributed platform. X10, in contrast, makes locality and places explicitly visible to the user, in keeping with the positioning of X10 in the NUCC software stack that was discussed in Section 2.

7.3 Communication

X10 activities communicate through the PGAS. The programming model and access mechanisms to the PGAS differ in X10 from other PGAS languages, like Split-C, Titanium, and UPC. In these languages, there is a uniform way to access shared memory, irrespective of whether the target location of the access is allocated on the local or a remote processor. Given an object reference `p`, it can be hard to tell from manual inspection if a field access of the form `p->x` or `p.x` will generate a non-local communication in these languages. This lack of performance transparency can in turn be a productivity bottleneck for performance tuning, since there's no indication a priori as to which data accesses in the program generate non-local communications. X10 follows the GALS model when accessing the PGAS, and has two key benefits compared to other PGAS languages. First, all non-local data accesses are explicitly visible in the code, and the language is well suited to source code level transformations that optimize communication *e.g.,* aggregation of two data accesses to the same place in a single "message" (activity). Second, the dynamic checking associated with the Locality Rule helps guide the user when an attempt is inadvertently made to access a non-local datum. We expect this level of locality checking to deliver the same kinds of productivity benefits to programming NUCC applications that has been achieved by null pointer and array bounds checking in modern serial applications.

7.4 Synchronization

X10 eschews monitors and locks, which are the main means of synchronization in Java, C#, and Titanium, in favor of atomic blocks [27]. Chapel [34] and Fortress [2] have similar language features: Fortress implements a transaction model with explicit abort; an abort or conflict of a transaction can be detected and handled similar to exceptions (`tryatomic`). Atomic blocks in X10 do not have such an abort mechanism and explicitly violate transaction semantics if an unhandled exception occurs during the execution of an atomic block. Unlike Chapel and Fortress, X10 supports conditional atomic blocks (`when`).

Clocks are a synchronization mechanism that is unique to X10. From an abstract viewpoint, clocks resemble (split) barriers, which are known, e.g., in UPC, Titanium, and other languages that adopt their control flow from the SPMD model. Clocks generalize split barriers in several ways: Clock synchronization is not necessarily global and may involve only a subset of activities in the overall system. Clocks operate in phases, where a phase corresponds to the number of times that the activities transited over the clock. A consequence of this phased computation model is a new class of variables, so-called *clocked final* variables, that are affiliated with a specific clock and are guaranteed to have a constant value during each phase of that clock. There are certain restrictions on the usage of clocks that enable a compiler to verify that clock-based synchronization is free from deadlock [47].

8. FUTURE WORK AND CONCLUSIONS

8.1 Future Work

In future work we plan to investigate the impact of X10 on high productivity and high performance.

8.1.1 Array Expressions

Our experience with writing applications in X10 indicates that many HPC programs can be expressed much more succinctly if the language is extended with richer support for operations on arrays, regions and distributions.

We are investigating the suitability of X10 to sophisticated HPC programs such as adaptive mesh refinement programs, and those involving multiple physics. It is conceivable that this work might lead us in a direction where it is natural to create more places dynamically (e.g. reflecting the need to create more concurrent locales for execution, based on the physics of the situation).

An allied notion is that of a *hierarchical* region, as developed for instance in the notion of hierarchically tiled arrays [22]. Here each point in a region is itself a region; thus one gets a hierarchy of regions. This leads naturally to the notion of hierarchical distributions, a concept that may be appropriate for handling adaptive mesh refinement.

Similarly, the need to handle sparse matrix computations naturally yields the idea of permitting *region transformations*, maps that transform one region to another. Consider a sparse 2-dimensional matrix represented as a single array `a`, along with an array `row` who's `i`th entry specifies the index in `a` of the first entry in the `i`th row, and an array `col` who's `i`th entry specifies the column of the `i`th entry in `a`. Now consider the region `ri` given by `[row[i]:row[i+1]-1]`. This specifies the indices of all the elements in `a` that correspond to the `i`th row of the matrix. Applying the array `col` to this region pointwise yields another region, designated by `col[ri]`, which represents the column indices of all the non-zero entries in the `i`th row of `a`. Now the dot product of the `i`th row of `a` and a column vector `x` may be represented

simply by `((a | ri)* (x | col[ri])).sum()`.[5] Such an implicit presentation of iteration over an array subregion clearly represents the programmer's design intent and permits a compiler to generate much more efficient code.

We believe that these extensions will be useful for both productivity and performance.

8.1.2 Type System

We intend to develop a rich place-based, clock-aware type system that lets a programmer declare design intent, and enables bugs related to concurrency and distribution to be captured at compile time. For instance we intend to develop a type-checker that can ensure (for many programs) that the program will not throw a `ClockUseException`.

We are exploring the use of semantic annotations [13]. For instance a method could be labeled with a modifier `now` to indicate that its execution does not invoke a `resume` or `next` operation on any clock. Similarly a modifier `here` on a statement may indicate that all variables accessed during the execution of the statement are local to the current place.

We are developing the notion of *clocked final* data. A variable declared to be `clocked(c) final`, for a clocked `c` is guaranteed to be `final` in each phase of the clock `c`. It may be read and written only by activities registered on `c`. Assignment to a clocked final variable is thought of as setting the value of the variable for the next phase of the clock; this value is switched over to be the current value when the clock transitions to the next phase. Clocked final values may be accessed freely from any place because they can only be accessed by activities to which they appear as `final`. Many applications (e.g. Moldyn in the JGF benchmarks) can be written to use clocked final arrays. This annotation succinctly captures design intent as well as provides information to the implementation that can be used to improve performance.

8.1.3 Determinate, Deadlock-free Computation

We have recently built on the clocked final idea to propose a general framework for determinate, parallel, deadlock-free imperative computation (the clocked final (CF) computation model, [48]). We intend to develop this computation model further and instantiate it in X10.

8.1.4 Development of Refactoring Tools

Work is underway to exploit the semantics of X10 and design refactoring tools that take advantage of X10's concurrency and distribution primitives. We believe such tools will offer a substantially more productive experience to the application programmer than tools based on Sequential JAVA.

8.1.5 Performance-oriented Compiler

We intend to develop an efficient compiler and multi-node runtime system for X10, in line with the work described in Section 6. This work is critical to establishing the viability of X10 for the high-performance computing marketplace.

8.1.6 Integration with Continuous Program Optimization

The application of X10 to dynamic HPC problems such as adaptive mesh refinement requires that the runtime dynamically manage the mapping from places to hardware nodes. We are contemplating the design of a "job control language" intended to interact with the continuous program optimization engine [9]. A programmer may write code in this language to customize the load-balancing algorithm. In an extension of the language which creates new places, such a layer would also specify the hardware location at which these places are to be constructed.

8.2 Conclusion

Future NUCC systems built out of multi-core chips and tightly-coupled cluster configurations represent a significant departure from current systems, and will have a major impact on existing and future software. The ultimate challenge is supporting high-productivity, high-performance parallel programming: that is, designing a programming model that is simple and widely usable and yet efficiently implementable on current and proposed architectures without requiring "heroic" compilation and optimization efforts. This challenge has been plaguing the High Performance Computing community for decades, but the future trends towards NUCC systems suggest that its impact will be felt by the software industry as a whole.

Current OO languages are inadequate for addressing the needs of NUCC systems. We have designed a modern object-oriented language, X10, for high-productivity high-performance parallel programming in the presence of non-uniform data accesses. In this paper, we presented the motivation and principles underlying the X10 design, an overview of the language itself, discussion of examples written in X10 compared to current programming models, a productivity analysis focused on the problem of parallelizing serial code, and a summary of the current implementation status. Our early experiences with the X10 design, implementation, application studies, and productivity analyses indicate that X10 has the potential to help address the grand challenge of high-productivity high-performance parallel programming, while using modern OO programming principles as its foundation.

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, Frank Tip, Guang Gao, David Grove, Radha Jagadeesan, Maged Michael, Robert O'Callahan, Filip Pizlo, V.T. Rajan, Armando Solar-Lezama, Mandana Vaziri, and Jan Vitek. We also thank the PI of the PERCS project, Mootaz Elnozahy, for his support and encouragement.

[5] Here `a | ri` is X10 syntax for the sub-array formed by intersecting the region of `a` with `ri`, `*` represents pointwise multiplication of two arrays defined over an equinumerous region, and `.sum()` represents the sum-reduction of the array.

9. REFERENCES

[1] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[2] Eric Allan, David Chase, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress language specification version 0.618. Technical report, Sun Microsystems, April 2005.

[3] Yariv Aridor, Michael Factor, and Avi Teperman. cJVM: A single system image of a JVM on a cluster. In *Proceedings of the International Conference on Parallel Processing (ICPP'99)*, pages 4–11, September 1999.

[4] Henri E. Bal and M. Frans Kaashoek. Object distribution in Orca using compile-time and run-time techniques. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'93)*, pages 162–177, November 1993.

[5] Ray Barriuso and Allan Knies. SHMEM user's guide. Technical report, Cray Inc. Research, May 1994.

[6] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type specific memory coherence. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'95)*, pages 168–176, March 1990.

[7] Hans Boehm. Threads cannot be implemented as a library. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'05)*, pages 261–268, June 2005.

[8] Luca Cardelli. A language with distributed scope. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'95)*, pages 286–297, January 1995.

[9] Calin Cascaval, Evelyn Duesterwald, Peter F. Sweeney, and Robert W. Wisniewski. Multiple page size modeling and optimization. In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, September 2005.

[10] IBM International Technical Support Organization Poughkeepsie Center. Overview of lapi. Technical report sg24-2080-00, chapter 10, IBM, December 1997. www.redbooks.ibm.com/redbooks/pdfs/sg242080.pdf.

[11] 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.

[12] Elaine Cheong, Judy Liebman, Jie Liu, and Feng Zhao. TinyGALS: A Programming model for event-driven embedded systems. In *Proceedings of 2003 ACM Symposium on Applied Computing*, 2003.

[13] Brian Chin, Shane Markstrum, and Todd Millstein. Semantic type qualifiers. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'05)*, pages 85–95, June 2005.

[14] CILK-5.3 reference manual. Technical report, Supercomputing Technologies Group, June 2000.

[15] F. Darema, D.A. George, V.A. Norton, and G.F. Pfister. A Single-Program-Multiple-Data Computational model for EPEX/FORTRAN. *Parallel Computing*, 7(1):11–24, 1988.

[16] 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.

[17] 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.

[18] ECMA. Standard ecma-334: C♯ language specification. http://www.ecma-international.org/publications/files/ecma-st/Ecma-334.pdf, December 2002.

[19] Tarek El-Ghazawi, William W. Carlson, and Jesse M. Draper. UPC Language Specification v1.1.1, October 2003.

[20] High Performance Fortran Forum. High performance fortran language specification version 2.0. Technical report, Rice University Houston, TX, October 1996.

[21] Ian Foster and Carl Kesselman. The Globus toolkit. *The Grid: Blueprint of a New Computing Infrastructure*, pages 259–278, 1998.

[22] Basilio B. Fraquela, Jia Guo, Ganesh Bikshandi, Maria J. Garzaran, Gheorghe Almasi, Jose Moreira, and David Padua. The hierarchically tiled arrays programming approach. In *Proceedings of the Workshop on Languages, Compilers, and Runtime Support for Scalable Systems (LCR'04)*, pages 1–12, 2004.

[23] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM – Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[24] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, 2000.

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

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

[27] Timothy Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*, pages 388–402, October 2003.

[28] Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, and Michael Hind. Vertical profiling: Understanding the behavior of object oriented applications. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*

(OOPSLA'04), October 2004.

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

[30] Paul Hilfinger, Dan Bonachea, David Gay, Susan Graham, Ben Liblit, Geoff Pike, and Katherine Yelick. Titanium Language Reference Manual. Technical Report CSD-01-1163, University of California at Berkeley, Berkeley, Ca, USA, 2001.

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

[32] HPC challenge benchmark. http://icl.cs.utk.edu/hpcc/.

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

[34] Cray Inc. The Chapel language specification version 0.4. Technical report, Cray Inc., February 2005.

[35] The Java Grande Forum benchmark suite. http://www.epcc.ed.ac.uk/javagrande/javag.html.

[36] The Java RMI Specification. http://java.sun.com/products/jdk/rmi/.

[37] Arvind Krishnamurthy, David E. Culler, Andrea Dusseau, Seth C. Goldstein, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, pages 262 – 273, 1993.

[38] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9), 1979.

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

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

[41] 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.

[42] 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.

[43] Jose E. Moreira, Samuel P. Midkiff, Manish Gupta, Pedro V. Artigas, Marc Snir, and Richard D. Lawrence. Java programming for high-performance numerical computing. *IBM Systems Journal*, 39(1):21–, 2000.

[44] Robert W. Numrich and John Reid. Co-Array Fortran for parallel programming. *ACM SIGPLAN Fortran Forum Archive*, 17:1–31, August 1998.

[45] 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.

[46] OpenMP specifications. http://www.openmp.org/specs.

[47] Vijay Saraswat and Radha Jagadeesan. Concurrent clustered programming. In *Proceedings of the International Conference on Concurrency Theory (CONCUR'05)*, August 2005.

[48] Vijay Saraswat, Radha Jagadeesan, Armando Solar-Iezama, and Christoph von Praun. Determinate imperative programming: A clocked interpretetation of imperative syntax. Submitted for publication, available at http://www.saraswat.org/cf.html, September 2005.

[49] 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.

[50] 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.

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

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

[53] Lorna A. Smith, J. Mark Bull, and Jan Obdrzalek. A parallel Java Grande benchmark suite. In *Proceedings of Supercomputing 2001, Denver, Colorado*, November 2001.

[54] Standard Performance Evaluation Corporation (SPEC). SPECjbb2000 (java business benchmark). http://www.spec.org/jbb2000.

[55] Thorsten von Eicken, David E. Culler, Seth C. Goldstein, and Klaus E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA'92)*, pages 256–266, May 1992.

[56] Robert W. Wisniewski, Peter F. Sweeney, Kartik Sudeep, Matthias Hauswirth, Evelyn Duesterwald, Calin Cascaval, and Reza Azimi. Performance and Environment Monitoring for Whole-System Characterization and Optimization. In *Conference on Power/Performance interaction with Architecture,Circuits, and Compilers*, 2004.