A Brief Introduction To X10
(For the High Performance Programmer)

Vijay A. Saraswat Olivier Tardieu David Grove
David Cunningham Mikio Takeuchi Benjamin Herta

September 10, 2012

Preface

The vast and omni-present computer industry of today is based fundamentally on what we shall call the *standard* model of computing. This is organized around the idea that a single stream of instructions operates on *memory* – discrete cells holding *values* – by *reading* these values, operating on them to produce new values, and *writing* these values back into memory.

In the last few years this model has been changed fundamentally, and forever. We have entered the era of *concurrency* and *distribution*.

Moore's Law continues, unabated – making available twice the number of transistors every 18 months for the same silicon. Traditionally, computer architects have used this extra real estate to provide faster implementations of the standard model by increasing clock speed and deepening the pipelines needed to speculatively execute many instructions in parallel. Thus, the standard model has needed no change.

Unfortunately, clock speeds have now run up against the *heat envelope* – speed cannot be increased without causing the chips to run hotter than can be tolerated by current packaging.

What should this real estate be used for, then? Computer manufacturers have turned to multi-core parallelism, a radical idea for mainstream computing. Instead of using all the real estate to support the illusion of a single thread of execution, they have divided up the real estate to support multiple such threads of control. Multiple threads may now read and write the same locations simultaneously.

This change has been coupled with a second, less visible but equally powerful change. The widespread availability of cheap commodity processors and advances in computer networking mean that *clusters* of multiple computers are now commonplace. Further, a vast increase in the amount of data available for processing means that there is real economic value in using clusters to analyze this data, and act on it.

Consequently, the standard model must now give way to a new programming model. This model must support execution of programs on thousands of multi-core computers, with tens of thousands of threads, operating on petabytes of data. The model should smoothly extend the original standard model so that familiar ideas, patterns, idioms continue to work, in so far as possible. It should permit easy problems to be solved easily, and must allow sophisticated programmers to solve hard problems. It should permit programmers to make reasonably accurate performance predictions by just looking at the code. It should be practical and easily implementable on all existing computer systems.

Since 2004, we have been developing just such a new programming model. We began our work as part of the DARPA-IBM funded PERCS research project. The project set out to develop a petaflop computer (capable of 10^{15} operations per second), which could be programmed ten times more productively than computer of similar scale in 2002. Our specific charter was to develop a programming model for such large scale, concurrent systems that could be used to program a wide variety of computational problems, and could be accessible to a large class of professional programmers.

The programming model we have been developing is called the *APGAS* model, the *Asynchronous, Partitioned Global Address Space* model. It extends the standard model with two core concepts: *places* and *asynchrony*. The collection of cells making up memory are thought of as partitioned into chunks called places, each with one or more simultaneously operating threads of control. A cell in one place can refer to a cell in another – i.e. the cells make up a (partitioned) global address space. Four new basic operations are provided. An `async` spawns a new thread of control that operates asynchronously with other threads. An async may use an `atomic` operation to execute a set of operations on cells located in the current place, as if in a single step. It may use the `at` operation to switch the place of execution. Finally, and most importantly it may use the `finish` operation to execute a sequence of statements and wait for all asyncs spawned (recursively) during their execution to terminate. These operations are orthogonal and can nest arbitrarily with few exceptions. The power of the APGAS model lies in that many patterns of concurrency, communication and control – including those expressible in other parallel models of computation such as PThreads, OpenMP, MPI, Cilk – can be effectively realized through appropriate combinations of these constructs. Thus APGAS is our proposed replacement for the standard model.

Any language implementing the old standard model can be extended to support the APGAS model by supplying constructs to implement these operations. This has been done for Java^{TM}(X10 1.5), C (Habanero C), and research efforts are underway to do this for UPC.

Our group has designed and implemented a new programming language, X10, organized around these ideas. X10 is a modern language in the strongly typed, object-oriented programming tradition. Its design draws on the experience of team members with foundational models of concurrency, programming language design and semantics, type systems, static analysis, compilers, runtime systems, virtual machines. Our goals were simple – design a language that fundamentally focuses on concurrency and distribution, and is capable of running with good performance at scale, while building on the established productivity of object-oriented languages. In this, we sought to span two distinct programming language traditions – the old tradition of statically linked, ahead-of-time compiled languages such as Fortran, C, C++, and the more modern dynamically linked, VM based languages such as Java, C#, F#. X10 supports both compilation to the JVM and, separately, compilation to native code. It runs on the PERCS machine (Power 7 CPUs, P7IH interconnect), on Blue Gene machines, on clusters of commodity nodes, on laptops; on AIX, Linux, MacOS; on PAMI, and MPI; on Ethernet and Infiniband.

We believe the X10 language has now reached a stage of maturity where its introduction to a wider class of programmers is appropriate. As of August 2012, the language has been stable for over a year. Many tens of thousands of lines of X10 application code have been written, by the core X10 development team and other project participants. M3R, a main memory implementation of the Hadoop Map Reduce Java API, has been developed. It executes Map Reduce jobs on a cluster by transparently caching data in memory, and delivers significant performance improvements over the Hadoop engine. Work is under way to develop a comprehensive library of multinode graph algorithms. The PERCS project milestone to demonstrate execution of X10 at scale on the PERCS machine (over 47,000 cores) has been met.

This book presents an introduction to programming in X10 for the high performance programmer. It uses the PERCS benchmark programs to illustrate key X10 programming concepts. It discusses particular concurrency, communication and computation idioms and shows how they can be expressed in such a way that their implementation can scale to tens of thousands of cores.

In conclusion, we would like to acknowledge the key role played by several members of the core X10 team. We particularly thank Vivek Sarkar, Kemal Ebcioglu, Nathaniel Nystrom, Igor Peshansky, Yoav Zibin, Kiyokuni Kawachiya, Hiroshi Horii, Yuki Makino, Bob Fuhrer, Emmanuel Geay, Sayantan Sur, Bowen Alpern, Tong Wen, Mandana Vaziri, Bard Bloom, Evelyn Duesterwald and Avraham Shinnar for their substantial contributions to the project. We also thank Philippe Charles, Sreedhar Kodali, Ganesh Bikshandi, Krishna Nandivada Venkat, Pradeep Varma, Bruce Lucas, Juemin Zhang, Anju Kambadur, Jim Xia, Shivali Agarwal, R K Shyamsundar, Haichuan Wang, Yan Li, Qi Ming, Hai Bo Lin, and Christoph von Praun for their many contributions. We also thank our (current and erstwhile) colleagues on the PERCS project, particularly Kevin Gildea, Steve Cooper, George Almasi, Calin Cascaval, Vickie Robinson and Pat Esquivel. We thank John Richards, Jonathan Brezin, Cal Swart, John Thomas and Rachel Bellamy of the PERCS Productivity Assessment team.

We are grateful to have had the long term support of Michael Hind, Charles Lickel, Tamiya Onodera, Brent Hailpern and David McQueeney in IBM Research. Only an organization as rich as IBM Research can bring together the talent, funding and opportunity for the vast (decade long!) effort it takes to develop a new programming language!

Vijay wishes to thank Prof Martha Kim from Columbia University for co-teaching several editions of the X10-based course on “Principles and Practice of Parallel Programming”. Some of the material from this book has been used in this course.

Hawthorne, NY
Tokyo, Japan
August 2012

Vijay, Olivier, Dave,
Dave, Mikio and Ben

Contents

Part I

Programming Framework

This part motivates and introduces X10. §1 provides motivation for the development of X10 and outlines progress made on the project so far. §2 introduces the basic elements of the language. §3 presents in detail the core concurrency and distribution constructs of the APGAS model. Finally, §4 sketches the X10 v2.2.3 implementation (focusing on the Native Runtime), and discusses several pragmas and their applicability conditions.

1 Why X10?

X10 is a new high-performance, high productivity programming language developed in the IBM "Productive, Easy-to-use, Reliable Computing System" project (PERCS, [35]), supported by the DARPA High Productivity Computer Systems initiative (HPCS, [7]). X10 is a class-based, strongly typed, explicitly concurrent, garbage-collected, multi-place object-oriented programming language ([27, 4]).

The X10 language was designed to address the central productivity challenge of High Performance Computing (HPC). We recognized the reality that there was no overlap between the models, tools, and techniques that underlie mainstream computing practice and those that underlie HPC. Main stream computing has developed eco-systems capable of supporting tens of millions of programmers (with a wide variety of talent and expertise), on a diversity of hardware platforms using object-oriented programming concepts (modularity, separation of concerns, use of libraries, ...) and powerful tools built on frameworks such as Eclipse [37]. They remain at the cutting edge by successfully leveraging research in the theory and practice of programming languages (development of advanced static analysis tools, type systems, program understanding and refactoring systems, etc).

The HPC eco-system, on the other hand, is highly specialized and extremely small (perhaps thousands of programmers), of use on a handful of advanced architectures, and focused on programming models such as MPI ([30]) that speak to the narrow needs of regular computations at extreme scale, and remain virtually disconnected from advances in programming languages. Consequently, the national labs have to take in programmers from academia and industry who have virtually no background in HPC and train them afresh. The loss in productivity for the HPC field is manifold: narrow focus implies limited talent pool, implies small pool of trained programmers, implies no economic incentive for advanced tools and programming environments, implies high bar to entry, implies limited talent pool.

In January 2004, we set out to break this vicious cycle with the creation of the X10 project. We recognized the impending multi-core crisis, and that both the commercial and HPC models would need to change to accommodate concurrency. We made two critical decisions.

First, drawing on our past work in theoretical computer science (and the slogan "Clean, but Real" many in the community share), we designed a core programming model that fundamentally accounts for concurrency (asyncs) and distribution (places), with just four basic, orthogonal primitives – `async`, `finish`, `atomic` (and friends), and `at`. This model we later dubbed the Asynchronous Partitioned Global Address Space (APGAS) model [25]. We showed that many well-known patterns of communication, computation and concurrency arise through combinations of these four operations. At the same time these constructs are simple and elegant enough that we could formulate foundational models and establish important semantic properties – such as deadlock-freedom for a very large fragment of the language (see, e.g.[27]).

Second, we decided to place X10 firmly in the modern object oriented programming languages tradition ([13, 36]) – even though languages such as Java have long been regarded with aversion by HPC programmers because of their perceived shortcomings for high performance computing. We analyzed Java in depth and recognized that we could build a programming language (based on APGAS) that was different from Java in its treatment of concurrency, distribution and some core sequential programming constructs while addressing these shortcomings, and yet remaining within the broad tradition and hence being quite familiar to Java programmers. Furthermore, we realized that we could build two different back-ends for the same language — a *Native back-end* that compiles X10 to C++ and uses static ahead of time compilation techniques, and a *Managed back-end* that compiles X10 to byte-codes for the JVM and executes X10 on a cluster of JVMs. Even though this substantially complicated our implementation task, we realized that this was critical to the eventual success of X10 – not just as an "HPC" language used by a handful of HPC programmers, but as a commercially viable language used to address a diversity of programming problems that are best tackled with JVM-based programs.

In these two major design decisions we explicitly and consciously went against the prevalent HPC dogma which holds that the way to design HPC languages is to start with a very narrowly focused programming model that is known to execute efficiently, e.g. Single Program Multiple Data (SPMD) computation, and expand it out carefully to meet certain desirable criteria, e.g. permitting overlap of computation with communication. Arguably, this was the philosophy that led to UPC ([9]). These two decisions were challenged at the first major review of X10 in Dec 2004, conducted by a blue ribbon panel put together by DARPA, including designers of MPI ([31]) and UPC and other key HPC programming model leaders. But we stayed firm.

We demonstrated the first implementation of X10 on a single JVM in February 2005, leveraging the Polyglot compiler framework [21]. In 2007, we demonstrated X10 running on multiple places, on the native back-end, and successfully met an internal milestone (the "CEO milestone"). We showed that the Berkeley UPC team's state-of-the-art performance on 3-d FFT (achieved by overlapping computation and communication) [20] could be replicated in X10 ([2]), and our submission to the 2007 HPC Challenge competition won the award for "Most productive research implementation" ([34]).

We are now reaping the rewards of those foundational decisions. First, to our knowledge, X10 is the first post-MPI programming language that has demonstrated performance at scale ($40K$ cores and above), on a variety of benchmarks of interest to the HPC community. Our paper on lifeline-based global load balancing ([28]) showed that X10 constructs (async, at, finish) provide the expressiveness for a new, scalable approach to globally load balancing irregular workloads (such as Unbalanced Tree Search), surpassing results obtained in prior research work (using, e.g. languages such as UPC). Simultaneously, the productivity impact of X10 for developing HPC applications has been recognized by the studies performed by the PERCS Productivity Assessment team [8, 5, 16, 10].

Second, the attractiveness of X10 for commercial workloads was recognized in 2009 by IBM Research leadership, leading to increased funding for the Managed back-end. This funding (independent of DARPA PERCS funding) paved the way for the development of a large set of libraries in X10 (ported from Apache Harmony) and Java inter-operability research.

This funding has been instrumental in X10 being usable as a foundation for parallel application frameworks for the commercial world. The commercial world itself had seen the rise of application frameworks such as Hadoop [33] which rely on a simple but powerful programming framework (completely divorced from MPI, and hence not rooted in the HPC tradition) to bring the aggregate compute and memory power of thousands of nodes to solve commercially important problems in machine learning, data mining, descriptive and predictive analytics etc. With the Main Memory Map Reduce (M3R) project [29], we have demonstrated that programs written in Java against the Hadoop APIs could in fact be executed by an X10 map reduce engine that leverages HPC techniques to keep application data in memory, perform in-memory shuffles, and reduce the amount of communication between nodes. We showed that this resulted in performance improvements of up to 40x on iterative map reduce applications, such as the Page Rank computation. We showed that programs written in higher-level languages that compile down to Hadoop Map Reduce programs, such as Pig ([38]) and DML ([12]) , can be executed unchanged on M3R.

M3R is now recognized as having enabled a whole new class of applications – interactive big data analysis applications – and is being commercially deployed by IBM.

Third, X10 is beginning to attract interest from outside the original development groups. The research and education community has come to recognize the importance of X10. X10 and the APGAS framework are the foundations for the spin-off Habanero project at Rice University. After an independent evaluation, a group at UCLA decided to base their new compiler project on X10. X10 is being taught at over thirty universities ([40]). For instance, 2012 will mark the fourth edition of an X10-based course on "Principles and Practice of Parallel Programming" being taught at Columbia University. (This course is now part of the regular curriculum for undergraduates at Columbia University.) We are aware of over 70 publications on X10 ([39]) involving over 130 researchers. The first three papers on X10 (from 2004-05) now have over 700 citations. The X10 workshop series ([24, 23]) has been established at PLDI, the third edition will be held in 2013.

We believe these results speak amply to the success of our basic approach to address the productivity challenges of High Performance Computing systems. We believe X10 is the first HPC language to demonstrate both performance at scale for traditional HPC benchmarks, and advance the state of the art in handling certain kinds of commercial work-loads. The simplicity of core X10 constructs, and its proximity to well-established commercial programming languages has meant that X10 has already found acceptance in academia. X10 is available on a wide variety of platforms, including the PERCS hardware, as well as x86 and Blue Gene systems, on Linux, Mac OSX, and Windows.

2 X10 basics

2.1 Core Object-oriented features

The core object-oriented features of X10 are very similar to those of Java or C#.

A program consists of a collection of top-level compilation units ("unit" for short). A unit is either a *class*, a *struct* or an *interface*.

```
package examples;
import x10.io.Console;
public class Hello {
    public static def main(args:Rail[String]){
     if(args.size>0) Console.OUT.println("The first arg is: "+args(0));
     Console.OUT.println("Hello, X10 world!");
     val h = new Hello();
     var result: Boolean = h.run();
     Console.OUT.println("The answer is: "+result);
  }
  public def run():Boolean=true;
}
```

Classes, structs and interfaces live in *packages* (e.g. `examples` above). Packages typically consist of a sequence of identifiers separated by "`.`" (with no spaces).

Packages are used to control access to top-level units. If they are marked `public` they can be accessed from code in any other package. If they have no accessibility modifier, they can be accessed only by code living in the same package.

A file can contain multiple units – however only one of them may be marked `public`. The name of the file must be the same as the name of the `public` unit, with the suffix "`.x10`".

The *fully qualified name* (FQN) of a class, a struct or an interface is the name of the class, the struct or the interface prefixed with the name of the package that unit lives in. For instance, the FQN for the class above is `examples.Hello`. A unit `A` must use the FQN of another unit `B` unless it has an explicit `import` statement.

For instance:

```
import x10.io.Console;
```

permits the name `Console` to be used in all the units in the file without being qualified by the package name.

Packages are "flat" – importing a package `x10` does not imply (for instance) that the package `x10.examples` is imported automatically. However, the name of a package is connected to the directory structure of the code. All the units defined in a package `x10.examples` must live in files in the directory `x10/examples` (relative to some base directory). All the units defined in the package `x10` must live in files in the directory `x10`.

2.1.1 Class

A class is a basic *bundle* of data and code. It specifies a number of *members*, namely *fields*, *methods*, *constructors*, and member classes and interfaces. Additionally a class specifies the name of another class from which it *inherits* (the *superclass*) and zero or more interfaces which it *implements*.

The members available at a class (i.e. that can be used by variables whose type is that class) are those defined at a class, and those defined in superclasses.

A class may be marked `final`. `final` classes cannot be subclassed.

Fields

A field specifies a data item that belongs to the class:

```
var nSolutions:Int = 0;
public static val expectedSolutions =
    [0, 1, 0, 0, 2, 10, 4, 40, 92, 352, 724, 2680, 14200, 73712, 365596, 2279184, 14772512];
val N:Int;
```

Fields may be mutable (`var`) or immutable (`val`). The type of a mutable field must always be specified (it follows the name of the variable, and a "`:`"). A mutable field may or may not be initialized. The type of an immutable field may be omitted if the field declaration specifies an *initializer*. For instance, in the above fragment the type `Rail[Int]` is inferred for the field `expectedSolutions`. The value of an immutable field does not have to be specified through an initializer – as long as it is supplied in every constructor for the class. However, if an initializer is not specified the type of the field must be specified.

Fields may be instance or static. By default fields are instance fields, static fields are marked with the flag `static` (as illustrated above). Every object has one copy of an instance field. However, in each place, there is only one copy of each static field for all instances of that class in the place. In X10 static fields are required to be immutable.

Instance fields are inherited by subclasses. This means that an object always has enough space for the instance fields of the class of which the object is an instance, and all of the object's superclasses.

A field defined in a class may be *shadowed* in a subclass if the subclass defines a field with the same name (but possibly a different type). The value of a shadowed field can always be accessed by using the qualifier `super`.

It is a compile-time error for a class to declare two fields with the same name.

Properties

A class may specify one or more properties. Properties are immutable instance fields that can be used in constrained types §2.3.2. They are also distinguished from other instance fields in that they are initialized in a particular way, through the invocation of a `property` call during execution of the constructor §2.1.1 for the object or struct. It is a compile time error for a constructor to have a normal execution path which does not contain a `property` call.

To make the compile-time type-checking of constraints more tractable, X10 v 2.2.3 requires that the types of properties are "simpler" than the type of the class or struct they occur in. Specifically, the graph formed with classes or structs as nodes and an edge from node v to node w if the class or struct corresponding to v has a field whose base type is w cannot have cycles. One consequence of this restriction is that if a class or struct has a type parameter `T`, then the type of a field cannot be `T`.

Methods

A method is a named piece of code, parametrized by zero or more variables (the *parameters*). A method may be `void` – in which case it returns no value and is usually called just for its effect on the store – or it may return a value of a given type.

A method may have one or more type parameters; such a method is said to be *type generic*. A method may have a *method guard* : the guard may specify constraints on the type parameters.

The *signature* of a method consists of its name and the types of its arguments. A class may not contain two methods whose signatures are the same. Additionally, a method specifies a *return type*.

A value `e` may be returned from the body of a method by executing a `return e;` statement. The return type of a method may be inferred in X10 (that is, it does not need to be explicitly supplied by the user). It is the least upper bound of the types of all expressions `e` where the body of the method contains a `return e` statement.

Methods may be *instance* methods or *static* methods. By default methods are instance methods. Methods are marked static by using the qualifier `static`.

Consider the code in Table 2.1. The methods `safe(j:Int)`, `search(R:Region(1))` and `search()` are instance methods. The instance methods of a class are available for

```
 1 class Board {
 2     val q: Rail[Int];
 3     def this() {
 4         q = new Rail[Int](0, (Int)=>0);
 5     }
 6     def this(old: Rail[Int], newItem:Int) {
 7         val n = old.size;
 8         q = new Rail[Int](n+1, (i:Int)=> (i < n? old(i) : newItem));
 9     }
10     def safe(j: Int) {
11         val n = q.size;
12         for ([k] in 0..(n-1)) {
13             if (j == q(k) || Math.abs(n-k) == Math.abs(j-q(k)))
14                 return false;
15         }
16         return true;
17     }
18     def search(R: IntRange) {
19         for (k in R)
20             if (safe(k))
21                 new Board(q, k).search();
22     }
23 
24     def search() {
25         if (q.size == N) {
26             nSolutions++;
27             return;
28         }
29         this.search(0..(N-1));
30     }
31 }
```

Table 2.1: NQueens Board

every object that is an instance of the class. Instance methods are invoked using the syntax `e.m(e1,..., en)`. `e` is said to be the *receiver* of the method invocation, and `e1,...,en` the *arguments*.

Each expression in X10 has a *static type*(Section §2.3). The compiler reports an error when processing a method invocation `e.m(e1,..., en)` if it cannot find precisely one method named `m` on the static type of `e` which has `n` arguments and which is such that the static type of `ei` is a subtype of the declared type of the `i`th argument.

The body of an instance method may access the state of the receiver (called the *current object*) through the special variable `this` (Line 29). Unless there is risk of ambiguity, the prefix "`this.`" may be omitted; thus the code can also be written as:

```
def search() {
  for ([k] in R) searchOne(k);
}
```

The code for static methods does not have an associated current object, and hence cannot use `this`. Static methods are invoked by naming the class as the receiver, e.g. `NQueens.main(null)`.

Inheritance Methods may be inherited. That is, methods defined on superclasses are available as methods on a subclass, unless they are overridden by another method declaration with the same signature. Instance methods are said to be *virtually dispatched*: this means that when a method is invoked on an object `o` that is an instance of class `C`, the inheritance tree for `C` is searched (starting from `C`) for a method definition with the same signature. The first definition found is executed. The type rules for X10 guarantee that at runtime such a method will exist.

Overloading In X10 methods may be overloaded. This means that a class may have multiple methods with the same name – of necessity they must have a different signature. These methods have nothing to do with each other.

Overloading is very convenient – it permits the programmer to use a method name for some abstract concept and than provide instantiations of that concept for different parameter types through different overloaded methods with the same name. The name `search` is overloaded in Table 2.1, for instance.

Access control The qualifiers `private, public, protected` may be used to limit access to a method. `private` methods may be accessed only by code in the same class. `public` methods can be accessed from any code. `protected` methods can only be accessed in the same class or its subclasses. If no qualifier is provided, a method can be accessed by code in the same package (it is said to be *package protected*).

Constructors

A class may specify zero or more *constructors*. For instance:

```
def this() { this(8);}
def this(N:Int) { this.N=N;}
```

Constructors may be overloaded just like methods. The qualifiers `private, public, protected` may be used to limit access to a constructor.

Instances of a class are created by invoking constructors using the `new` expression:

```
new Board(q, k)
```

The compiler declares an error if it cannot find a constructor for the specified class with the same number of formal arguments such that the formal type of each argument is a supertype of the type of each actual argument.

If a class does not have a constructor an *implicit* constructor is created. If the class has no properties then the implicit constructor is of the form:

```
def this() {}
```

If the class has properties `x1:T1`, ..., `xn:Tn` then the implicit constructor is of the form:

```
def this(x1:T1,...,xn:Tn) {
    property(x1,...,xn);
}
```

2.1.2 Structs

An object is typically represented through *handles*, indirect references to a contiguous chunk of data on the heap. The space allocated for an object typically consists of space for its fields and space for a *header*. A header contains some bytes of data that represent meta-information about the object, such as a reference to the table of methods used to implement virtual dispatching on the object. X10 also permits the definition of *structs*, which are different.

A `struct` is a "header-less" object. It can be represented by exactly as much memory as is necessary to represent the fields of the struct (modulo alignment constraints) and with its methods compiled to "static" methods. This is accomplished by imposing certain restrictions on structs (compared to objects). A `struct` definition is just like a `class` definition except that it is introduced by the keyword `struct` rather than by `class`. Like a class, a struct may define zero or more fields, constructors and methods and may implement zero or more interfaces. However, a struct does not support inheritance; it may not have an `extends` clause. A struct may not be recursive: that is, there can be no cycles in the graph whose nodes are struct definitions and whose edges from a to b record that the struct a has a field of type (struct) b.

For instance the struct:

```
package examples;
public struct Complex(re:Double, im:Double){}
```

defines a `Complex` struct, instances of which can be represented by exactly two doubles. An `Array[Complex]` with `N` elements can be implemented (in the Native backend) with a block of memory containing `2*N` doubles, with each `Complex` instance inlined.

X10 does not support any notion of a reference to a struct. Structs are passed by value. Equality == on structs is defined as point-wise equality on the fields of the struct. Further, structs are immutable; no syntax is provided to update fields of a struct in place. While these decisions simplify the language, they make it awkward to express certain kinds of code, hence mutability and references to structs may be introduced in future versions of the language.

2.1.3 Function literals

X10 permits the definition of functions via literals. Such a literal consists of a parameter list, followed optionally by a return type, followed by =>, followed by the body (an expression).

For instance the function that takes an argument `i` that is an `Int` and returns `old(i)` if `i < n` and `newItem` otherwise, is written as:

```
(i:Int) => (i < n ? old(i) : newItem)
```

The type of this value is `(Int)=>Int`. Note that the return type is inferred in the above definition. We could also have written it explicitly as:

```
(i:Int):Int => (i < n ? old(i) : newItem)
```

For instance:

```
(a:Int) => a // the identity function on Int's
(a:Int, b:Int):Int => a < b ? a : b // min function
```

Above, the type of the first value is `(Int)=>Int`, the type of functions from `Int` to `Int`. The type of the second is `(Int,Int)=>Int`.

A function is permitted to access immutable variables defined outside its body (such functions are sometimes called closures). Note that the immutable variable may itself contain an object with mutable state.

The return type can often be omitted – it is inferred to be the type of the return expression.

If `x` is a function value of type `(S1, ..., Sn) => T`, and `s1, ..., sn` are values of the given types, then `x(s1,...,sn)` is of type `T`.

2.2 Statements

The sequential statements of X10 should be familiar from Java and C++.

Assignment.

The statement `x=e` evaluates the expression `e` and assigns it to the local variable `x`.

The statement `d.f=e` evaluates `d` to determine an object `o`. This object must have a field `f` that is mutable. This value is changed to the value obtained by evaluating `e`.

In both cases the compiler checks that the type of the right hand side expression is a subtype of the declared type.[1]

Conditionals.

`if (c) then S else S`. The first branch is executed if the condition `c` evaluates to true, else the second branch is executed. One-armed conditionals are also permitted (the `else S` may be dropped).

While loops.

`while (c) S`. The condition `c` is evaluated. If it is true, the body `S` is executed, and the control returns to the top of the loop and the cycle of condition evluation and body execution repeats. Thus the loop terminates only when the condition evaluates to false.

A while loop may be labeled `l: while (c) S` (just as any other statement). The body of a `while` loop may contain the statements `continue l;` and `break l;`. The first causes control to jump to the top of the loop labeled `l`, i.e. the rest of the code in the body of the loop following the `continue l;` statement is not executed. `break l;` causes control to jump after the loop labeled `l` (that is, control exits from that loop).

For loops.

X10 supports the usual sort of `for` loop. The body of a `for` loop may contain `break` and `continue` statements just like `while` loops.

Here is an example of an explicitly enumerated for loop:

```
def sum0(a:Rail[Int]):Int {
var result:Int=0;
for (var x:Int=0; x < a.size; ++x)
    result += a(x);
return result;
}
```

X10 also supports enhanced for loops. The for loop may take an index specifier `v in r`, where `r` is any value that implements `x10.lang.Iterable[T]` for some type `T`. The body of the loop is executed once for every value generated by `r`, with the value bound to `v`.

[1]If it is not, the compiler tries to first check for user-defined coercions. Only if it cannot find one does it declare an error. See the Language Manual for more details.

```
def sum1(a:Rail[Int]):Int {
var result:Int=0;
for (v in a.values())
    result += v;
return result;
}
```

Of particular interest is `IntRange`. The expression `e1 .. e2` produces an instance of `IntRange` from `l` to `r` if `e1` evaluates to `l` and `e2` evaluates to `r`. On iteration it enumerates all the values (if any) from `l` to `r` (inclusive). Thus we can sum the numbers from `0` to `N`:

```
def sum2(N:Int):Int {
var result:Int=0;
for (v in 0..N) result +=v;
return result;
}
```

One can iterate over multiple dimensions at the same time using `Region`. A `Region` is a data-structure that compactly represents a set of *points*. For instance, the region `(0..5)*(1..6)` is a 2-d region of points `(x,y)` where `x` ranges over `0..5` and `y` over `1..6`. (The bounds are inclusive.) The natural iteration order for a region is lexicographic. Thus one can sum the coordinates of all points in a given rectangle:

```
def sum3(M:Int, N:Int):Int {
var result:Int=0;
for ([x,y] in (0..M)*(0..N))
    result += x+y;
    return result;
};
```

Here the syntax `[x,y]` is said to be *destructuring syntax*; it destructures a 2-d point into its coordinates `x` and `y`. One can write `p[x,y]` to bind `p` to the entire point.

Throw statement.

X10 has a non-resumptive exception model. This means that an exception can be thrown at any point in the code. In X10 2.2.3, all exceptions are *unchecked*. This means that methods do not need to explictly declare the set of exceptions that they may raise.

Throwing an exception causes the call stack to be unwound until a catcher can be found (see below). If no catcher can be found the exception is thrown to the closest dyamically enclosing `finish` surrounding the throw point.

Try catch statement.

A `try/catch/finally` statement works just as in Java. It permits exceptions to be caught (through a `catch` clause) and computation resumed from the point of capture. A `finally` clause specifies code that must be run whenever control exits the `try` block – regardless of whether the return is normal or exceptional.

Return statement.

The statement `return;` causes control to return from the current method. The method must be a `void` method. The statement `return e;` causes control to return from the current method with the value `e`.

A description of the other control constructs may be found in [26].

2.3 Types

X10 is a statically type-checked language. This means that the compiler checks the type of each expression, and ensures that the operations performed on an expression are those that are permitted by the type of the expression.

The name `C` of a class or interface is the most basic form of a type in X10. X10 also supports

- *generic types* (e.g. `Array[Int]`) that take types as parameters,
- *constrained types* (e.g. `Array{self.rank==1}`) that specify an additional constraint on the properties of the base type,
- *nested types* (e.g. `MyArray[T].Reducer`
- *functional type* (e.g. `(Int)=>Int`.

A type specifies a set of members (fields, methods, constructors) that are defined on expressions of that type.

The availability of members of a type also depends on the accessibility relation. That is, a member of a type may be defined but not accessible (e.g. it is marked `private`).

A variable of a type `S` can only be assigned values whose static type `T` is the same as `S` or is a *subtype* of `S`.

A type `C` is a subtype of type `E` if there is some type `D` such that `C` is a subtype of `D` and `D` is a subtype of `E`.

If (class or interface) `C[X1,...,Xm]` `extends` or `implements` type `D[S1,..., Sn]` (for `m` and `n` non-negative integers) then for every mapping `A` from type variables to types, the type obtained by applying `A` to `C[X1,...,Xm]` is a subtype of the type obtained by applying `A` to `D[S1,...,Sn]`.

A constrained type `C{c}` is a subtype of `D{d}` if `C` is a subtype of `D` and `c` implies `d`.

A nested type `A.B` is a subtype of `C.D` if `A` is the same as `C` and `B` is a subtype of `D`.

A function type `(A1,...,Am)=>B` is a subtype of `(D1,...,Dn)=>E` if `m=n`, `Di` is a subtype of `Ai` (for `i` in `1..n`) and `B` is a subtype of `E`.

2.3.1 Generic types

X10 permits *generic types*. That is a class or interface may be declared parametrized by types:

```
class List[T] {
var item:T;
    var tail:List[T]=null;
    def this(t:T){item=t;}
}
```

This specifies that the class definition actually specifies an infinite family of classes, namely those obtained by replacing the type parameter `T` with any concrete type. For instance, `List[Int]` is the class obtained by replacing `T` in the body of the class with `Int`:

```
class List_Int {
var item:Int;
    var tail:List_Int=null;
    def this(t:Int){item=t;}
}
```

Clearly generic types are very convenient – after all they let you figure out the code of the class once, and then use it an unbounded number of times ... in a type-safe fashion.

X10 types are available at runtime, unlike Java (which erases them). Therefore, Array[Int] and Array[Float] can be overloaded. X10 does not have primitive types and Int etc. can be used to instantiate type parameters. Bounds may be specified on type parameters and methods can be invoked on variables whose type is that parameter, as long as those methods are defined and accessible on the bound.

2.3.2 Constrained types

Constrained types are a key innovation of X10.

A constrained type is of the form `T{c}` where `T` is a type and `c` is a `Boolean` expression of a restricted kind. `c` may contain references to the special variable `self`, and to any final variables visible at its point of definition.

Such a type is understood as the set of all entities `o` which are of type `T` and satisfy the constraint `c` when `self` is bound to `o`.

The permitted constraints include the predicates == and `!=`. These predicates may be applied to constraint terms. A constraint term is either a final variable visible at the point of definition of the constraint, or the special variable `self` or of the form `t.f` where `f` names a field, and `t` is (recursively) a constraint term. In a term `self.p` `p` must name a property of `self`.

Examples:

```
Matrix[Int]{self.I==100, self.J==200} // 100 x 200 matrix
Matrix[Int]{self.I==self.J} // the type of square matrices.
```

Constraints may also include occurrences of user-defined predicates. See [26] for details.

2.3.3 Type definitions

A type definition permits a simple name to be supplied for a complicated type, and for type aliases to be defined. A type definition consists of a name, an optional list of type parameters, an optional list of (typed arguments), an "=" symbol, followed by a type in which the type parameters and arguments may be used.

For instance the type definition:

```
public static type boolean(b:Boolean) = Boolean{self==b};
```

permits the expression `boolean(true)` to be used as shorthand the type `Boolean{self==true}`.

X10 permits top-level type definitions. A type definition with principal name `N` must be defined in the file `N.x10`. Example:

```

package examples;
public class Matrix[T](I:Int,J:Int) {

}
type Matrix[T](I:Int,J:Int)=Matrix[T]{self.I==I,self.J==J};
```

Note that a file may contain a type definition all by itself. This is the case for `Rail.x10`:

```
public type Rail[T]=Array[T]{self.rank==1,self.zeroBased,self.rect,self.rail};
```

3 The APGAS model

In this chapter, we cover the APGAS constructs in X10 v2.2.3.

3.1 Async

The fundamental concurrency construct in X10 is `async`:

```
Stmt ::= async Stmt
```

The execution of `async S` may be thought of as creating a new *activity* to execute `S` and returning immediately. The newly created activity runs in parallel with the current activity and has access to the same heap of objects as the current activity. Thus an X10 computation may have many concurrent activities “in flight“ at the same time. Activities communicate with each other by reading and writing shared variables, e.g. fields of objects that both have access to.

An activity may be thought of as a very light-weight thread of execution. This means that the statement `S` may in fact contain just a few instructions, e.g. reading the value of a variable, performing some computation and then writing the value of a variable. There is no restriction on the statement `S` – it may contain any other construct (including other `async`s). In particular, activities may be long-running – indeed they may run for ever. In particular they make invoke recursive methods. Hence an activity is associated with its own control stack.

Activities (unlike threads in Java) are not named. There is no runtime object corresponding to an activity that is visible to user programs. This permits the implementation the freedom to actually *not* create a separate activity for each `async` at runtime as long as the semantics of the language are not violated. For instance, the compiler may decide to translate the program fragment:

```
 async { f.x=1;}
f.y=2;
```

to

```
 f.x=1;
f.y=2;
```

This is called “inlining” an activity. It becomes much harder for a compiler to inline if activities can have names – for, to inline the activity the compiler will need to figure out that the name is not used later.

Activities cannot be interrupted or aborted once they are in flight. They must proceed to completion.

In `async S`, the code in `S` is permitted to refer to immutable variables defined in the lexically enclosing scope. This is extremely convenient when writing code.

Local vs Global termination Because an activity may spawn further activities, we distinguish between *local termination* and *global termination*. We say that an activity `async S` has locally terminated if `S` has terminated. We say that it has *globally terminated* if it has locally terminated and further any activities spawned during its execution have themselves (recursively) globally terminated.

3.2 Finish

`finish` is a construct that converts global termination to local termination.

Using `async` we can specify that the elements of a rail should be doubled in parallel by:

```
// Initialize the i'th element to i.
val a:Rail[Int] = new Rail[Int](N, (i:Int)=>i);
// Asynchronously double every element of the array
for(i in 0..(a.size-1))
  async a(i) *= 2;
}
```

Consider now what happens if we attempt to read the value of a location:

```
// Initialize the i'th element to i.
val a:Rail[Int] = new Rail[Int](N, (i:Int)=>i);
// Asynchronously double every element of the array
for(i in 0..(a.size-1))
  async a(i) *= 2;
Console.OUT.println("a(1)=" + a(1));
}
```

Will it print out `1` or `2`? We cannot say! The activity executing `a(1) *= 2` may not have terminated by the time the current activity starts executing the print statement!

This is a fundamental problem. The programmer needs a mechanism for specifying *ordering* of computations. To this end we introduce the `finish` statement:

```
Stmt ::= finish Stmt
```

An activity executes `finish S` by executing `S` and then waiting until all activities spawned during the execution of `S` (transitively) terminate. Simple, but powerful!

To ensure proper termination of an X10 program, the `main` method is executed within a `finish`. Thus the program terminates only when the `main` method globally terminates. This property ensures that for every activity created during program execution there is a corresponding `finish` statement that will be notified of its termination. We say the activity tree is *rooted*.

We can now write our program as follows:

```
// Initialize the i'th element to i.
val a:Rail[Int] = new Rail[Int](N, (i:Int)=>i);
// Asynchronously double every element of the array
finish for(i in 0..(a.size-1))
  async a(i) *= 2;
Console.OUT.println("a(1)=" + a(1));
}
```

and be guaranteed that the output will be `2`. Notice that little needs to change in the program – we just add `finish` and `async` at the right place!

Table 3.1 shows how the Fibonacci program can be written in parallel. It uses an idiom that is of interest in many other settings. The natural functional way to write `fib` is through a recursive function call:

```
def fib(n:Int):Int = (n <=2)? 1: fib(n-1)+fib(n-2);
```

The value is returned on the activity's stack. However, we are interested in running `fib` in parallel, and hence we will want a way by which multiple activities can invoke multiple `fib` calls, each in their own stack. X10 does not permit the call stack of one activity to be shared by another, or the code running during the execution of an activity to read/write the local variables in a stack frame of another activity.[1]

Table 3.1 presents a slightly more verbose Fibonacci program that makes explicit the interaction of activities through objects on the heap. A heap object is created for each recursive invocation of `fib`. The object has a single field which initially contains the input argument to the call, and on return contains the result of the call.

Note that the program illustrates that during execution `finish` and `async` may be scoped arbitrarily: the body of an `async` can contain `finish` statements, and the body of a `finish` can contain `async` statements. This interplay is at the heart of the expressiveness of the `async`/`finish` model.

[1]Why? Because in general this is not a safe operation. The stack frame being referred to may not exist any more!

```
public class Fib {
  var n:Int=0;

  def this(n:Int) {this.n = n;}

  def fib() {
    if (n <= 2) {
        n=1;
        return;
    }
    val f1=new Fib(n−1);
    val f2=new Fib(n−2);
    finish {
        async f1.fib();
        f2.fib();
    }
    n=f1.n+f2.n;
  }

  public static def main(args:Rail[String]) {
    if (args.size < 1) {
      Console.OUT.println("Usage: Fib <n>");
      return;
    }
    val n = Int.parseInt(args(0));
    val f = new Fib(n);
    f.fib();
    Console.OUT.println("fib(" + n + ")= " + f.n);
  }
}
```

Table 3.1: Fib

3.2.1 The rooted exception model

X10 supports a *rooted exception model*. Any exception thrown inside an activity can be caught and handled within the activity by executing a `try/catch` statement. What happens if there is no `try/catch`?

The rooted model offers an answer. Since every activity has a governing `finish`, we let the exception propagate up from the activity to the governing `finish`. When executing a `finish S` statement, all exceptions thrown by activities spawned during the execution of `S` are accumulated at the `finish` statement. If at least one exception has been received at the `finish` statement, then it throws a `MultipleExceptions` exception with the set of exceptions as an argument. This ensures that no exception gets dropped on the floor (unlike Java).

3.3 Atomic

Consider a parallel version of the Histogram program:

```
def hist(a:Rail[Int], b: Rail[Int]) {
  finish for(i in 0..(a.size-1)) async {
    val bin = a(i)% b.size;
    b(bin)++;
  }
}
```

However, this program is *incorrect*! Why? Multiple activities executing `b(bin)++` may interfere with each other! The operation `d++` is not an *atomic* operation.

An atomic operation is an operation that is performed in a single step with respect to all other activities in the system (even though the operation itself might involve the execution of multiple statements). X10 provides the *conditional atomic statement* as a basic statement:

```
Stmt ::= when (c) Stmt
```

Here `c` is a condition, called the *guard* of the statement. An activity executes `when (c) S` atomically – in a single step – provided that the condition `c` is satisfied. Otherwise it blocks (waiting for the condition to be true).

The conditional atomic statement is an extremely powerful construct. It was introduced in the 1970s by Per Brinch Hansen and Tony Hoare under the name “conditional critical region”. This is the only construct in X10 that permits one activity to block waiting for some other set of activities to establish some condition on shared variables.

This construct has an important special case. The statement `atomic S` is just shorthand for `when (true) S`.

Since the construct is so powerful, it is subject to several conditions for ease of implementation.

- The condition `c` must be *sequential*, *non-blocking*, and *local*. That is, it must not spawn new activities (contain a `async` or `finish`), it must not itself call a `when`, and it must not access remote locations (contain an `at`).
- The statement `S` must be also be *sequential*, *non-blocking*, and *local*.
- The condition `c` may be evaluated an arbitrary number of times by the X10 scheduler to determine if it returns true. Therefore it should be side-effect free and as simple as possible.

The first two restrictions are enforced dynamically: while executing either the condition or the statement if the program attempts to execute a `async`, `at`, `finish`, or `when` then an `IllegalOperationException` will be raised. The last restriction is currently not enforced by X10.

The Histogram problem can now be solved correctly:

```
def hist(a:Rail[Int], b: Rail[Int]) {
  finish for(i in 0..(a.size−1)) async {
    val bin = a(i)% b.size;
    atomic b(bin)++;
  }
}
```

A parallel N-Queens program is given in Table 3.2

3.4 Places

We come now to a central innovation of X10, the notion of *places*. Places permit the programmer to explicitly deal with notions of *locality*.

3.4.1 Motivation

Locality issues arise in three primary ways.

First, consider you are writing a program to deal with enormous amounts of data – say terabytes of data, i.e. thousands of gigabytes. Now you may not have enough main memory on a single node to store all this data – a single node will typically have tens of gigabytes of main storage. So therefore you will need to run your computation on a *cluster* of nodes: a collection of nodes connected to each other through some (possibly high-speed) interconnect. That is, your single computation will actually involve the execution of several operating system level processes, one on each node. Unfortunately, acccessing a memory location across a network is typically orders of magnitude slower (i.e. has higher *latency*) than accessing it from a register on the local core. Further, the rate at which data can be transferred to local memory (*bandwidth*) is orders of

```
public class NQueensPar {
    var nSolutions:Int = 0;
    public static val expectedSolutions =
        [0, 1, 0, 0, 2, 10, 4, 40, 92, 352, 724, 2680, 14200, 73712, 365596, 2279184, 14772512];
    val N:Int;
    def this(N:Int) { this.N=N;}
    def start() {
        finish new Board().search();
    }
    class Board {
        val q: Rail[Int];
        def this() {
            q = new Rail[Int](0, (Int)=>0);
        }
        def this(old: Rail[Int], newItem:Int) {
            val n = old.size;
            q = new Rail[Int](n+1, (i:Int)=> (i < n? old(i) : newItem));
        }
        def safe(j: Int) {
            val n = q.size;
            for ([k] in 0..(n−1)) {
                if (j == q(k) || Math.abs(n−k) == Math.abs(j−q(k)))
                    return false;
            }
            return true;
        }
        def search(R: IntRange) {
            for (k in R) async
                if (safe(k))
                    new Board(q, k).search();
        }

        def search() {
            if (q.size == N) {
                atomic nSolutions++;
                return;
            }
            this.search(0..(N−1));
        }
    }
    public static def main(args: Rail[String]) {
        val n = args.size > 0 ? Int.parseInt(args(0)) : 8;
        println("N=" + n);

        val nq = new NQueensPar(n);
        var start:Long = −System.nanoTime();
        nq.start();
        val result = nq.nSolutions==expectedSolutions(nq.N);
        start += System.nanoTime();
        start /= 1000000;
        println("NQueens " + nq.N + " has " + nq.nSolutions + " solutions" +
            (result? " (ok)" : " (wrong)") + " (t=" + start + "ms).");
    }

    static def println(s:String) {
        Console.OUT.println(s);
    }
}
```

Table 3.2: NQueens

magnitude higher than the rate at which it can be transferred to memory across the cluster.

As with implicit parallelism, one could try to write extremely clever compilers and runtimes that try to deal with this memory wall implicitly. Indeed, this is the idea behind *distributed shared memory* (DSM). The entire memory of a collection of processes is presented to the programmer as a single shared heap. Any activity can read and write any location in shared memory. However, there are no efficient implementations of DSM available today. The primary conceptual issue is that the programmer has lost control over decisions that can have orders of magnitude impact on performance of their code. When looking at a single assignment statement `a.f=e`, the programmer has no way of knowing whether this is going to take dozens of cycles or millions of cycles.

A second primary motivation arises from heterogeneity. Computer architects are looking to boost computational power by designing different kinds of specialized cores, each very good at particular kinds of computations. In general, these *accelerators* interact with the main processor at arm's length.

Two primary cases in point are the Toshiba-Sony-IBM Cell Broadband Engine ("Cell processor" for short), and general-purpose graphical processing engines (GPGPUs for short), from vendors such as NVidia and AMD. These provide an enormous boost in computational power for particular kinds of regular loops at the cost of introducing specialized hardware.

For instance, the Cell provides eight specialized processors (SPEs) on a single chip, connected to each other through a high-speed (on-chip) bus. These processors may execute many instructions in parallel (they have "single instruction multiple data", SIMD, instructions). However, data needs to be explicitly transfered from main memory to a local cache on each SPE, operated upon, and then transfered back.

The third motivation is similar to the second, but involves only homogeneous cores. Multiple cores may share precious resources, such as L1 and L2 cache. To improve performance, it may make sense to *bind* activities to particular cores, in particular to force certain groups of activities to work on the same cores so that they can amortize the cost of cache misses (because they are operating on the same data). Or it may make sense to bind them to *different* cores that do not share an L2 cache so that the execution of one does not pollute the cache lines of the other.

3.4.2 The `at` construct

A *place* in X10 is a collection of data and activities that operate on that data. A program is run on a fixed number of places. The binding of places to hardware resources (e.g. nodes in a cluster, accelerators) is provided externally by a configuration file, independent of the program.

Programs are typically written to operate on any number of places. The number of places in a particular run of the program can be queried through `Place.MAX_PLACES`.

In X10 v2.2.3 all places are uniform. In future versions of the language we will support heterogeneity by permitting different kinds of places, with the ability to check the attributes of a place statically, and hence write code that depends on the *kind* of place it is running on.

The primary construct for exposing places to the programmer is:

```
Stmt ::= at (p) Stmt
```

An activity executing `at (p) S` suspends execution in the current place. The object graph G at the current place whose roots are all the variables V used in `S` is serialized, and transmitted to place `p`, deserialized (creating a graph G' isomorphic to G), an environment is created with the variables V bound to the corresponding roots in G', and `S` executed at `p` in this environment. On local termination of `S`, computation resumes after `at (p) S` in the original location. The object graph is not automatically transferred back to the originating place when `S` terminates: *any updates made to objects copied by an* `at` *will not be reflected in the original object graph.*

The `at` statement is a reminder to the programmer that at this point communication may potentially happen across the network. Therefore the programmer should be aware of what values will be transmitted from the source to the destination place and also be aware that any mutations to objects during the execution of `S` will only be visible at `p`

Because of this ability to shift the current place, `at` is said to be a *place-shifting* operation. It is the only control construct related to places in X10.

The indexical constant `here` can be used to determine the current place.

It is also possible to use `at` as an expression to do a computation and return a value to the originating place.

```
Expr ::= at (p) { S; E }
Expr ::= at (p) E
```

In this variant of `at`, the value of `E` is serialized from `p` back to the originating place and used as the value of the `at` expression.

3.4.3 GlobalRef

Of course the abstraction of a partitioned global address space without some means of actually referring to objects in a different partition would not be of much use. Therefore the X10 standard library provides the `GlobalRef` struct in the `x10.lang` package. Using GlobalRef, the programmer can easily create cross-place references by capturing a GlobalRef within an `at` statement.

The core API of `GlobalRef` is shown below. It has two operations: a constructor to create a `GlobalRef` to encapsulate an object and a method to access the encapsulated object, which is constrained via a method guard to only be applicable at the home `Place` of the `GlobalRef`.

```
public struct GlobalRef[T](home:Place) {T <: Object} {

    /**
```

```
     * Create a GlobalRef encapsulating the given object of type T.
     */
    public native def this(t:T):GlobalRef[T]{self.home==here};

    /**
     * Return the object encapsulated in the GlobalRef.
     * This method can only be invoked at the Place at which
     * the GlobalRef was created.
     */
    public native operator this(){here == this.home}:T;

}
```

To access the encapsulated object, the programmer uses `at` to place-shift to the ref's `home`:

```
at (gr.home) {
  val obj = gr();
  // ...compute using obj...
}
```

In X10, the programmer must explicitly place-shift using `at` to access an object that lives in another `Place`. Requiring an explicit `at` is a design choice that was made to highlight for the programmer that at this point communication may potentially happen across the network.

Table 3.3 shows some additional operations defined on the `GlobalRef` struct. These functions encapsulate more efficient implemenetations of common idiomatic usages of `GlobalRef`.

3.4.4 PlaceLocalHandle

Another useful abstraction for a partitioned global address space is that of an abstract reference that can be resolved to a different object in each partition of the address space. Such a reference should be both efficiently transmittable from one `Place` to another and also be efficiently resolvable to its target object in each `Place`. A primary use of such an abstraction is the definition of distributed data structures that need to keep a portion of their state in each `Place`. In X10, the `PlaceLocalHandle` struct in the `x10.lang` package provides such an abstraction. Examples of its usage can be found in virtually all of the programming examples in the next part of this book.

```
/**
 * Unsafe native method to get value.
 * Assumes here == this.home; however this is not enforced
 * by a constraint because it would entail dynamic checks.
 * Must only be called at this.home!
 */
private native def localApply():T;

/**
 * Evaluates the given closure at (this.home), passing as a
 * parameter the object that is encapsulated by this GlobalRef.
 * This is equivalent to the following idiom:
 * if (here == this.home)
 * return eval(this());
 * else
 * return at (this.home) eval(this());
 * However, because it does not use a place constraint on the
 * method, it avoids a dynamic place check on the first branch.
 */
public native def evalAtHome[U](eval:(T)=> U):U;

/**
 * If (this.home == here), returns the object that is
 * encapsulated by this GlobalRef. If (this.home != here),
 * returns a copy at the current place.
 * This is equivalent to the following idiom:
 * if (here == this.home)
 * return this();
 * else
 * return at (this.home) this();
 * However, because it does not use a place constraint on the
 * method, it avoids a dynamic place check on the first branch.
 */
public native def getLocalOrCopy():T;
```

Table 3.3: Advanced features of GlobalRef API

4 The X10 Performance Model

Programmers need an intuitive understanding of the performance characteristics of the core constructs of their programming language to be able to write applications with predictable performance. We will call this understanding a *performance model* for the language. Desirable characteristics of a performance model include simplicity, predictive ability, and stability across different implementations of the language. The performance model should abstract away all non-essential details of the language and its implementation, while still enabling reasoning about those details that do have significant performance impact. Languages with straightforward mappings of language constructs to machine instructions usually have fairly straightforward performance models. As the degree of abstraction provided by the language's constructs and/or the sophistication of its implementation increase, its performance model also tends to become more complex.

In this chapter, we present an abridged version[1] of the performance model for X10 v2.2.3 focusing on those aspects that are most relevant for the HPC programmer. Although the rate of change of the X10 language has significantly decreased from earlier stages of the project, the language specification and its implementations are still evolving more rapidly than production languages such as C++ and Java. Therefore, we break this chapter into two logical sections: aspects that we believe are fundamental to the language itself and aspects that are more closely tied to specific choices embodied in the X10 v2.2.3 implementation and thus more likely to change in future versions. The second logical section is presented simultaneously with a discussion of some of the central implementation decisions embodied in the X10 v2.2.3 runtime system and compiler.

4.1 Fundamental X10 Performance Model

The core language model of X10 is that of a type-safe object-oriented language. Thus much of the core performance model is intended to be similar to that of Java. We believe the Java performance model is generally well-understood. Therefore in this section we focus on areas where the performance models for X10 and Java diverge or where X10 has new constructs that do not trivially map to Java constructs.

[1] based on the discussion in [14]

4.1.1 X10 Type System

The type systems of X10 and Java differ in three ways that have important consequences for the X10 performance model. First, although X10 classes are very similar to Java's, X10 adds two additional kinds of program values: functions and structs. Second, X10's generic type system does not have the same erasure semantics as Java's generic types do. Third, X10's type system includes *constrained types*, the ability to enhance type declarations with boolean expressions that more precisely specify the acceptable values of a type.

Functions in X10 can be understood by analogy to closures in functional languages or local classes in Java. They encapsulate a captured lexical environment and a code block into a single object such that the code block can be applied multiple times on different argument values. X10 does not restrict the lifetime of function values; in particular they may escape their defining lexical environment. Thus, the language implementation must ensure that the necessary portions of the lexical environment are available for the lifetime of the function object. In terms of the performance model, the programmer should expect that an unoptimized creation of a function value will entail the heap allocation of a closure object and the copying of the needed lexical environment into that object. The programmer should also expect that trivial usage of closures (closures that do not escape and are created and applied solely within the same code block) will be completely eliminated by the language implementation via inlining of the function body at the application site.

Structs in X10 are designed to be space-efficient alternatives to full-fledged classes. Structs may implement interfaces and define methods and fields, but do not support inheritance. Furthermore structs are immutable: a struct's instance fields cannot be modified outside of its constructor. This particular design point was chosen specifically for its implications for the performance model. Structs can be implemented with no per-object meta-data and can be freely inlined into their containing context (stack frame, containing struct/object, or array). Programmers can consider structs as user-definable primitive types, that carry none of the space or indirection overheads normally associated with objects.

The X10 generic type system differs from Java's primarily because it was designed to fully support the instantiation of generic types on X10 structs without losing any of the performance characteristics of structs. For example, `x10.lang.Complex` is a struct type containing two double fields; `x10.util.ArrayList[T]` is a generic class that provides a standard list abstraction implemented by storing elements of type `T` in a backing array that is resized as needed. In Java, a `java.util.ArrayList[Complex]`, would have a backing array of type `Object[]` that contained pointers to heap-allocated `Complex` objects. In contrast, the backing storage for X10's `x10.util.ArrayList[Complex]` is an array of inline Complex structs without any indirection or other object-induced space overheads. This design point has a number of consequences for the language implementations and their performance model. Much of the details are implementation-specific so we defer them to Section 4.4 and to the paper by Takeuchi et al [32]. However, one high-level consequence of this design is generally true: to implement the desired semantics the language implementation's runtime type infrastructure must be able to distinguish between different instantiations of a generic class (since instantiations on different struct types will have different memory layouts).

Constrained types are an integral part of the X10 type system and therefore are intended to be fully supported by the runtime type infrastructure. Although we expect many operations on constrained types can be checked completely at compile time (and thus will not have a direct runtime overhead), there are cases where dynamic checks may be required. Furthermore, constrained types can be used in dynamic type checking operations (`as` and `instanceof`). We have also found that some programmers prefer to incrementally add constraints to their program, especially while they are still actively prototyping it. Therefore, the X10 compiler supports a compilation mode where instead of rejecting programs that contain type constraints that cannot be statically entailed it, will generate code to check the non-entailed constraint at runtime (in effect, the compiler will inject a cast to the required constrained type). When required, these dynamic checks do have a performance impact. Therefore part of performance tuning an application as it moves from development to production is reducing the reliance on dynamic checking of constraints in frequently executed portions of the program.

4.1.2 Distribution

An understanding of X10's distributed object model is a key component to the performance model of any multi-place X10 computation. In particular, understanding how to control what objects are serialized as the result of an `at` can be critical to performance understanding.

Intuitively, executing an `at` statement entails copying the necessary program state from the current place to the destination place. The body of the `at` is then executed using this fresh copy of the program state. What is necessary program state is precisely defined by treating each upwardly exposed variable as a root of an object graph. Starting with these roots, the transitive closure of all objects reachable by properties and non-transient instance fields is serialized and an isomorphic copy is created in the destination place. Furthermore, if the `at` occurs in an instance method of a class or struct and the body of the `at` refers to an instance field or calls an instance method, `this` is also implicitly captured by the `at` and will be serialized. It is important to note that an isomorphic copy of the object graph is created even if the destination place is the same as the current place. This design point was chosen to avoid a discontinuity between running a program using a single place and with multiple places.

Serialization of the reachable object graph can be controlled by the programmer primarily through injection of transient modifiers on instance fields and/or GlobalRefs. It is also possible to have a class implement a custom serialization protocol[2] to gain even more precise control. An X10 implementation may be able to eliminate or otherwise optimize some of this serialization, but it must ensure that any program visible side-effects caused by user-defined custom serialization routines happen just as they would have in an unoptimized program. Thus, the potential of user-defined custom

[2] `x10.io.CustomSerialization`

serialization makes automatic optimization of serialization behavior a fairly complex global analysis problem. Therefore, the base performance model for object serialization should not assume that the implementation will be able to apply serialization reducing optimizations to complex object graphs with polymorphic or generic types.

The X10 standard library provides the `GlobalRef`, `RemoteArray` and `RemoteIndexedMemoryChunk` types as the primitive mechanisms for communicating object references across places. Because of a strong requirement for type safety, the implementation must ensure that once an object has been encapsulated in one of these types and sent to a remote place via an `at`, the object will be available if the remote place ever attempts to spawn an activity to return to the object's home place and access it. For the performance model, this implies that cross-place object references should be managed carefully as they have the potential for creating long-lived objects. Even in the presence of a sophisticated distributed garbage collector [17] [3], the programmer should expect that collection of cross-place references may take a significant amount of time and incur communication costs and other overheads.

Closely related to the remote pointer facility provided by `GlobalRef` is the `PlaceLocalHandle` functionality. This standard library class provides a place local storage facility in which a key (the `PlaceLocalHandle` instance) can be used to look up a value, which may be different in different places. The library implementation provides a collective operation for key creation and for initializing the value associated with the key in each place. Creation and initialization of a `PlaceLocalHandle` is an inherently expensive operation as it involves a collective operation. On the other hand cross-place serialization of a `PlaceLocalHandle` value and the local lookup operation to access its value in the current place are relatively cheap operations.

4.1.3 Async and Finish

The `async` and `finish` constructs are intended to allow the application programmer to explicitly identify potentially concurrent computations and to easily synchronize them to coordinate their interactions. The underlying assumption of this aspect of the language design is that by making it easy to specify concurrent work, the programmer will be able to express most or all of the useful fine-grained concurrency in their application. In many cases, they may end up expressing more concurrency than can be profitably exploited by the implementation. Therefore, the primary role of the language implementation is to manage the efficient scheduling of all the potentially concurrent work onto a smaller number of actually concurrent execution resources. The language implementation is not expected to automatically discover more concurrency than was expressed by the programmer. In terms of the performance model, the programmer should be aware that an `async` statement is likely to entail some modest runtime cost, but should think of it as being a much lighter weight operation than a thread creation.

As discussed in more detail in Section 4.3, the most general form of `finish` involves the implementation of a distributed termination algorithm. Although programmers can

[3] which is not available in Native X10 v2.2.3

assume that the language implementation will apply a number of static and dynamic optimizations, they should expect that if a `finish` needs to detect the termination of activities across multiple places, then it will entail communication costs and latency that will increase with the number of places involved in the `finish`.

4.1.4 Exceptions

The X10 exception model differs from Java's in two significant ways. First, X10 defines a "rooted" exception model in which a `finish` acts as a collection point for any exceptions thrown by activities that are executing under the control of the `finish`. Only after all such activities have terminated (normally or abnormally) does the `finish` propagate exceptions to its enclosing environment by collecting them into a single `MultipleExceptions` object which it will then throw. Second, the current X10 language specification does not specify the exception semantics expected within a single activity. Current implementations of X10 assume a non-precise exception model that enables the implementation to more freely reorder operations and increases the potential for compiler optimizations.

4.2 X10 v2.2.3 Implementation Overview

X10 v2.2.3 is implemented via source-to-source compilation to another language, which is then compiled and executed using existing platform-specific tools. The rationale for this implementation strategy is that it allows us to achieve critical portability, performance, and interoperability objectives. More concretely, X10 v2.2.3 can be either compiled to C++ or Java. The resulting C++ or Java program is then compiled by either a platform C++ compiler to produce an executable or compiled to class files and then executed on a cluster of JVMs. We term these two implementation paths *Native X10* and *Managed X10* respectively.

Portability is important because we desire implementations of X10 to be available on as many platforms (hardware/operating system combinations) as possible. Wide platform coverage both increases the odds of language adoption and supports productivity goals by allowing programmers to easily prototype code on their laptops or small development servers before deploying to larger cluster-based systems for production.

X10 programs need to be capable of achieving close to peak hardware performance on compute intensive kernels. Therefore some form of platform-specific optimizing compilation is required. Neither interpretation nor unoptimized compilation is sufficient. However, by taking a source-to-source compilation approach we can focus our optimization efforts on implementing a smaller set of high-level, X10-specific optimizations with significant payoff while still leveraging all of the classical and platform-specific optimization found in optimizing C++ compilers and JVMs.

Finally, X10 needs to be able to co-exist with existing libraries and application frameworks. For scientific computing, these libraries are typically available via C APIs;

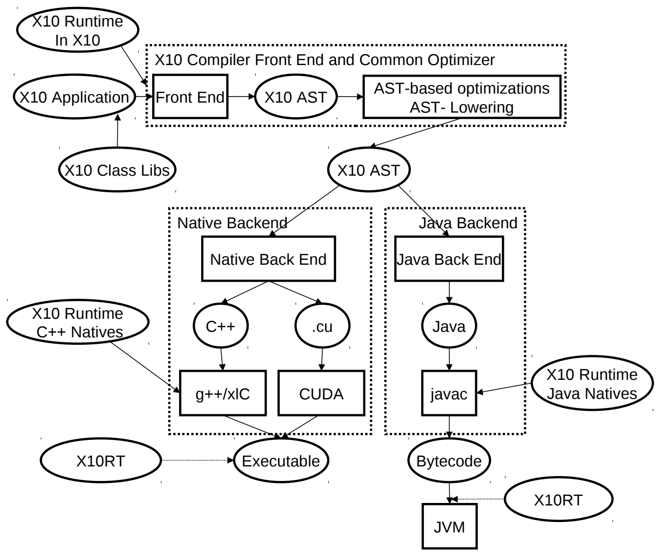

Figure 4.1: X10 Compiler Architecture

therefore Native X10 is the best choice. However, for more commercial application domains existing code is often written in Java; therefore Managed X10 is also an essential part of the X10 implementation strategy.

The overall architecture of the X10 compiler is depicted in Figure 4.1. This compiler is composed of two main parts: an AST-based front-end and optimizer that parses X10 source code and performs AST based program transformation; Native/Java backends that translate the X10 AST into C++/Java source code and invokes a post compilation process that either uses a C++ compiler to produce an executable binary or a Java compiler to produce bytecode.

Using source-to-source compilation to bootstrap the optimizing compilation of a new programming language is a very common approach. A multitude of languages are implemented via compilation to either C/C++ and subsequent post-compilation to native code or via compilation to Java/C# (source or bytecodes) and subsequent execution on a managed runtime with an optimizing JIT compiler. An unusual aspect of the X10 implementation effort is that it is pursuing both of these paths simultaneously. This decision has both influenced and constrained aspects of the X10 language design (consideration of how well/poorly a language feature can be implemented on both backends is required) and provided for an interesting comparison between the strengths and limitations of each approach. It also creates some unfortunate complexity in the X10 performance model because the performance characteristics of C++ and Java implementations are noticeably different.

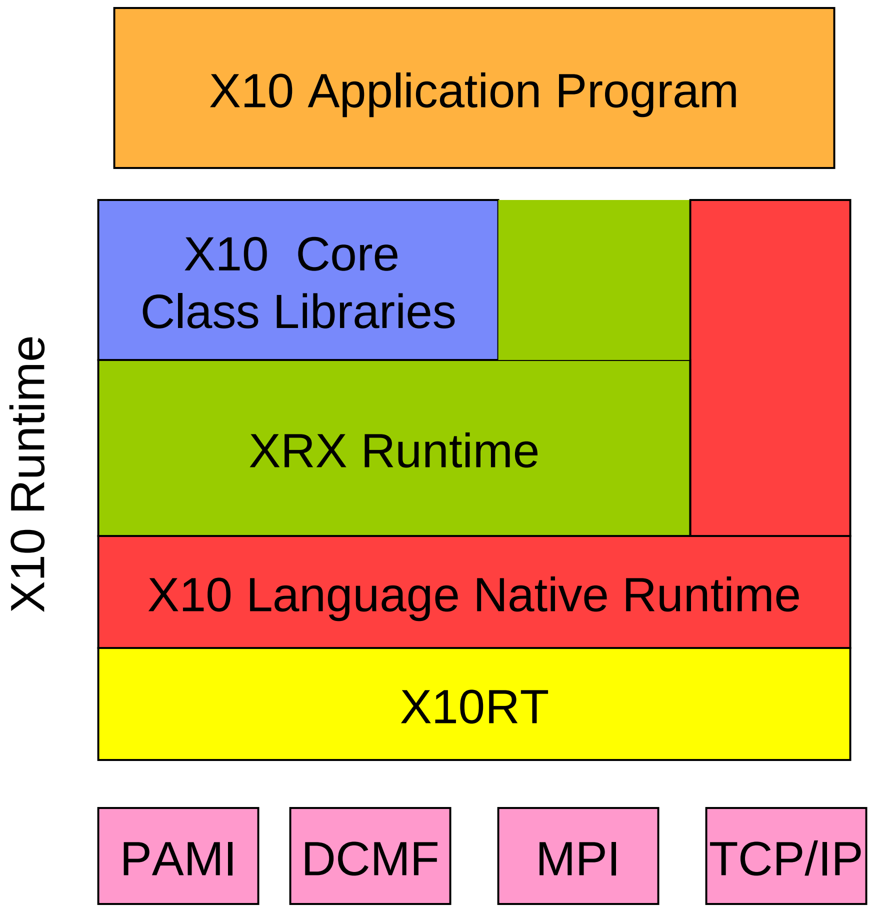

Figure 4.2: X10 Runtime Architecture

4.3 X10 v2.2.3 Runtime

Figure 4.2 depicts the major software components of the X10 runtime. The runtime bridges the gap between X10 application code and low-level facilities provided by the network transports (PAMI *etc.*) and the operating system. The lowest level of the X10 runtime is X10RT which abstracts and unifies the capabilities of the various network layers to provide core functionality such as active messages, collectives, and bulk data transfer.

The core of the runtime is *XRX*, the *X*10 *R*untime in *X*10. It implements the primitive X10 constructs for concurrency and distribution (*async*, *at*, *finish*, *atomic*, and *when*). The X10 compiler replaces these constructs with calls to the corresponding runtime services. The XRX runtime is primarily written in X10 on top of a series of low-level APIs that provide a platform-independent view of processes, threads, primitive synchronization mechanisms (e.g., locks), and inter-process communication. For instance, the *x10.lang.Lock* class is mapped to *pthread_mutex* (resp. *java.util.concurrent.locks.ReentrantLock*) by Native X10 (resp. Managed X10).

The X10 Language Native Runtime layer implements the object-oriented features of the sequential X10 language (dynamic type checking, interface invocation, memory management, *etc.*) and is written in either C++ (Native X10) or Java (Managed X10).

The runtime also provides a set of core class libraries that provide fundamental data types, basic collections, and key APIs for concurrency and distribution such as *x10.util.Team* for multi-point communication or *x10.array.Array.asyncCopy* for large data transfers.

In this section, we review the specifics of the X10 v2.2.3 runtime implementation focusing on performance aspects.

4.3.1 Distribution

The X10 v2.2.3 runtime maps each place in the application to one process.[4] Each process runs the exact same executable (binary or bytecode).

Upon launch, the process for place 0 starts executing the main activity.

```
finish { main(args); }
```

Static fields. Static fields are lazily initialized in each place when they are first accessed. Both X10 v2.2.3 backend compilers map static fields initialized with compile time constants to static fields of the target language. Other static fields are mapped to method calls. The method checks to see if the field has already been initialized in the current place, evaluates the initialization expression if it has not, and then returns the value of the field. Therefore accessing a static field with a non-trivial initialization expression will be more modestly more expensive in X10 than the corresponding operations in either C++ or Java.

X10RT. The X10 v2.2.3 distribution comes with a series of pluggable libraries for inter-process communication referred to as X10RT libraries [43, 44]. The default X10RT library—*sockets*—relies on POSIX TCP/IP connections. The *standalone* implementation supports SMPs via shared memory communication. The *mpi* implementation maps X10RT APIs to MPI [19]. Other implementations support various IBM transport protocols (DCMF, PAMI).

Each X10RT library has its own performance profile—latency, throughput, etc. For instance, the X10 v2.2.3 *standalone* library is significantly faster than the *sockets* library used on a single host.

The performance of X10RT can be tuned via the configuration of the underlying transport implementation. For instance, the *mpi* implementation honors the usual MPI settings for task affinity, fifo sizes, etc.

[4]The X10 v2.2.3 runtime may launch additional processes to monitor the application processes. These helper processes are idle most of the time.

Teams. The *at* construct only permits point-to-point messaging. The X10 v2.2.3 runtime provides the *x10.util.Team* API for efficient multi-point communication.

Multi-point communication primitives—a.k.a. *collectives*—provided by the *x10.util.Team* API are hardware-accelerated when possible, e.g., *broadcast* on BlueGene/P. When no hardware support is available, the Team implementation is intended to make a reasonable effort at minimizing communication and contention using standard techniques such as butterfly barriers and broadcast trees.

AsyncCopy. The X10 v2.2.3 tool chain implements *at* constructs via serialization. The captured environment gets encoded before transmission and is decoded afterwards. Although such an encoding is required to correctly transfer object graphs with aliasing, it has unnecessary overhead when transmitting immediate data, such as arrays of primitives.

As a work around, the X10 v2.2.3 *x10.array.Array* class provides specific methods—*asyncCopy*—for transferring array contents across places with lower overhead. These methods guarantee the raw data is transmitted as efficiently as permitted by the underlying transport with no redundant packing, unpacking, or copying. Hardware permitting, they initiate a direct copy from the source array to the destination array using RDMAs.[5]

4.3.2 Concurrency

The cornerstone of the X10 runtime is the scheduler. The X10 programming model requires the programmer to specify the place of each activity. Therefore, the X10 scheduler makes per-place decisions, leaving the burden of inter-place load balancing to the library writer and ultimately the programmer.

The X10 v2.2.3 scheduler assumes a symmetrical, fixed number of concurrent execution units (CPU cores) per process for the duration of the execution. This assumption is consistent with the HPCS context—job controllers typically assign concurrently running applications to static partitions of the available computing resources—but will be relaxed in subsequent releases of X10.

Work-Stealing scheduler. The X10 v2.2.3 scheduler belongs to the family of *work-stealing* schedulers [3, 11] with a *help-first* scheduling policy [15]. It uses a pool of worker threads to execute activities. Each worker thread owns a double-ended queue of pending activities. A worker pushes one activity for each *async* construct it encounters. When a worker completes one activity, it pops the next activity to run from its deque. If the deque is empty, the worker attempts to *steal* a pending activity from the deque of a randomly selected worker.

Since each worker primarily interacts with its own deque, contention is minimal and only arises with load imbalance. Moreover, a *thief* tries to grab an activity from the

[5]RDMA: remote direct memory access.

top of the deque whereas the *victim* always pushes and pops from the bottom, further reducing contention.

In X10 v2.2.3, the *thief* initially chooses a *victim* at random then inspects the deque of every worker in a cyclic manner until it manages to steal a pending activity.

The X10 scheduler borrows the deque implementation of Doug Lea's Fork/Join framework [18].

Life cycle. A worker may be in one of four states:

running one activity,

searching for an activity to execute,

suspended because the activity it is running has executed a blocking construct, such as *finish* or *when*, or method, such as *System.sleep* or *x10.util.Team.barrier*,

stopped because there are already enough workers running or searching.

Suspended and stopped workers are idle. Starting in X10 v2.2.3, the runtime will mostly suspend excess idle workers, but even if the place is entirely inactive the runtime still requires one worker to be polling the network (i.e., busy waiting) to respond to incoming messages. We expect that at least for some x10rt implementations that in later X10 release it will be possible to eliminate the need for busy waiting entirely.

Cooperative scheduler. The X10 v2.2.3 scheduler never preempts a worker running user code. The X10 v2.2.3 runtime is designed to enable achieving the highest possible performance on MPI-like distributed X10 applications where the programmer use matching send and receive instructions to achieve total control over the communication and execution schedule. If the user code never yields to the runtime then pending activities (local or remote) are not processed. In other words, the runtime does not make any fairness guarantee.

A worker may yield either by executing a blocking statement or by invoking the *Runtime.probe* method. The latter executes all the pending activities at the time of the call before returning to the caller. This includes all the pending remote activities—activities spawned here from other places—and all the activities already in this worker deque, but does not include activities in other deques.

Parallelism. The user can specify the number of workers in the pool in each place using the `X10_NTHREADS` environment variable.[6] The X10 v2.2.3 scheduler may create additional threads during the execution. But it strives to maintain the number of *non-idle* workers close to the requested value.

[6]Some X10RT libraries may internally use additional threads for network management. See documentation.

- If a worker suspends, the scheduler wakes a stopped worker if available or allocates and starts a new worker if not.
- If a suspended worker resumes, the scheduler preempts and stops a searching worker if any.
- If there are more than `X10_NTHREADS` workers running then the scheduler preempts and stops the first one who empties its deque.

As a result, the current scheduler guarantees the following properties that are intended to hold for any X10 implementation.

1. If there are `X10_NTHREADS` pending activities or more then there are `X10_NTHREADS` or more workers processing them, that is, running them or searching for them.
2. If there are "$n <$ `X10_NTHREADS`" workers running user code then there are "`X10_NTHREADS` $- n$" workers searching for pending activities.
3. If there are `X10_NTHREADS` or more workers running then there are no workers spinning.

Property 1 is the goal of any work-stealing scheduler: assuming the effort of finding pending activities is negligible, parallel activities are processed in parallel using `X10_NTHREADS` parallel processing units.

Property 2 guarantees that available cores are used to find pending activities quickly.

Property 3 mitigates the penalty of busy waiting in the current implementation: spinning workers are never getting in the way of the application provided the user makes sure that `X10_NTHREADS` is at most equal to the number of hardware cores available to the runtime for each place. For instance, if running 8 places on a 32-core node, `X10_NTHREADS` must not be larger than 4 workers per place.

Joining. In order to minimize pool size adjustments, the scheduler implements one key optimization. If a worker blocks on a *finish* construct but its deque is not empty, it does not suspend but instead processes the pending activities from its deque. It only eventually suspends if its deque becomes empty or if it reaches some other blocking construct (different from *finish*). By design, the pending activities that get processed by the worker in this phase must have been spawned from the blocked *finish* body. In the X10 v2.2.3 implementation, the worker will not attempt to steal activities from others if the *finish* construct is still waiting for spawned activities to terminate when the deque gets empty as this would require to carefully pick activities the *finish* construct is waiting for.

Thanks to this behavior, *finish* has much less scheduling overhead than other synchronization mechanisms, e.g., *when* constructs, and should be preferred when possible.

While this optimization is typically very effective at improving performance without observable drawbacks, it may lead to unbounded stack growth for pathological programs. Therefore, it may be disabled by setting the environment variable `X10_NO_STEALS`.[7]

[7]The `X10_NO_STEALS` flag essentially turns deep stacks into large collections of mostly-idle threads

Overhead. For each *async* statement, the current worker must make work available to other workers. In the best implementation and best case scenario (no contention) this requires at least one CAS instruction[8] per *async*. As a result, *async* constructs should only be used to guard computations that require (significantly) more resources than a CAS.

The X10 v2.2.3 runtime also allocates one small heap object per *async*. Again, anything smaller than that should be executed sequentially rather than wrapped with an *async*. Moreover, memory allocation and garbage collection can become a bottleneck if vast amounts of activities are created concurrently. The runtime therefore exposes the *Runtime.surplusActivityCount* method that returns the current size of the current worker deque. Application and library code may invoke this method to decide whether or not to create more asynchronous activities, as in:

```
if (Runtime.surplusActivityCount() >= 3) m(); else async m();
```

4.3.3 Synchronization

Finish. Within a place, one only needs to count activity creation and termination events to decide the completion of a *finish* construct. The story is different across places as inter-process communication channels are likely to reorder messages so that termination events may be observed ahead of the corresponding creation events. The X10 v2.2.3 implementation of *finish* keeps track of these events on an unambiguous, per-place basis.

In the worst-case scenario, with p places, there will be p counters in each place, that is, $p \times p$ counters for each *finish*. Moreover, there could be one inter-process message for each activity termination event. Messages could contain up to p data elements.

In practice however much fewer counters, fewer messages, and smaller messages are necessary thanks to various optimizations embedded in the X10 v2.2.3 implementation. In particular, events are accumulated locally and only transmitted to the *finish* place when local quiescence is detected—all local activities for this *finish* have completed. Counters are allocated lazily. Messages use sparse encodings.

To complement these runtime mechanisms, the programmer may also specify finish *pragmas*, which inform the runtime system about the kind of concurrent tasks that the finish will wait for, as in:

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

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

Currently, the runtime system supports five finish pragmas:

with smaller stacks, avoiding stack overflow errors. But ultimately, this only matters to unscalable programs of little practical relevance.

[8]CAS: compare-and-swap.

FINISH_ASYNC A finish for a unique async possibly remote.

FINISH_LOCAL A finish with no remote activity.

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

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

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

For now, neither the compiler nor the runtime makes any attempt at checking the validity of the pragma. Therefore a pragma, if misused, may result in the spurious (early) termination of the annotated finish or in a deadlock.

Uncounted Async. There are some situations in which an `async` may be annotated with `@Uncounted` (from the `x10.compiler` package). This annotation tells the compiler and runtime not to perform any of the book-keeping necessary to ensure that the governing `finish` progresses only after this `async` has terminated.

There are two principle cases in which the use of `Uncounted` is recommended. First, the programmer may be able to establish that the lifetime of this `async` and all `asyncs` spawned in its dynamic extent is contained within the lifetime of the current activity. For instance one situation in which this happens is if the `async` corresponds to a remote message send, and on the remote side the body of the message executes some local operations and responds with a message send to the originator. In the meantime, the originator sits waiting in a loop for the return message (e.g. by executing `Runtime.probe()`. This is a safe use of `Uncounted` (see §7).

The second situation is one in which the `async` corresponds to a message send directly implemented in the hardware, and some other reasoning is used to establish that these messages complete in time (see §6.2.3).

Atomic and When. The X10 v2.2.3 implementation of the *atomic* construct uses a place-wide lock. The lock is acquired for the duration of the atomic section. The *when* construct is implemented using the same lock. Moreover, every suspended *when* statement is notified on every exit from an atomic section, irrespective of condition.

The per-place lock effectively serializes all atomic operation in a place whether they might inerfere or not. This implementation does not scale well beyond a few worker threads. Similarly, the *when* implementation does not scale well beyond a few occurrences (distinct condition variables).

The X10 standard library provides various atomic classes and locks that enable better scaling. Both the *collecting finish* idiom and the *x10.util.WorkerLocalStorage* API may be also used to minimize contention.

4.4 X10 v2.2.3 Compilation

When an application programmer writes X10 code that they are intending to execute using Native X10, their base performance model should be that of C++. Unless discussed below, the expected performance of an X10 language construct in Native X10 is the same as the corresponding C++ construct.

4.4.1 Classes and Interfaces

X10 classes are mapped to C++ classes and the compiler directly uses the C++ object model to implement inheritance, instance fields, and instance methods. Interfaces are also mapped to C++ classes to support method overloading, but the X10 implements relationship is not implemented using the C++ object model. Instead, additional interface dispatch tables (akin to ITables in Java[9]) are generated by the X10 compiler "outside" of the core C++ object model. The motivation for this design decision was to stay within the simpler, single-inheritance subset of C++ that minimizes per-object space overheads and also preserves the useful property that a pointer to an object always points to the first word of the object and that no "this pointer adjustment" needs to be performed on assignments or during the virtual call sequence.

Non-interface method dispatch corresponds directly to a C++ virtual function call. Interface method dispatch will involve additional table lookups and empirically is 3 to 5 times slower than a virtual function call. C++ compilers typically do not aggressively optimize virtual calls, and will certainly not be able to optimize away the dispatch table lookup used to implement interface dispatch. Therefore, as a general rule, non-final and interface method invocations will not perform as well in Native X10 as they will in Managed X10.

Unless specially annotated, all class instances will be heap allocated and fields/variables of class types will contain a pointer to the heap allocated object.

4.4.2 Primitives and Structs

The dozen X10 struct types that directly correspond to the built-in C primitive types (int, float, etc.) are implemented by directly mapping them to the matching primitive type. Any X10 level functions defined on this types are implemented via static inline

[9]see the description of "searched ITables" in Alpern et al. [1]

methods. The performance characteristics of the primitive C++ types is exactly the performance of their X10 counterparts.

All other X10 structs are mapped to C++ classes. However, all of the methods of the C++ class are declared to be non-virtual. Therefore, the C++ class for a struct will not include a vtable word. Unlike object instances, struct instances are not heap allocated. They are instead embedded directly in their containing object or stack-allocated in the case of local variables. When passed as a parameter to a function, a struct is passed by value, not by reference. In C++ terms, a variable or field of some struct type S is declared to be of type S, not S*.

This implementation strategy optimizes the space usage for structs and avoids indirections. Programmers can correctly think of structs as taking only the space directly implied by their instance fields (modulo alignment constraints). However, passing structs, especially large structs, as method parameters or return value is significantly more expensive than passing/returning a class instance. In future versions of X10 we hope to be able to pass structs by reference (at the implementation level) and thus ameliorate this overhead.

4.4.3 Closures and Function Types

An X10 function type is implemented exactly the same as other X10 interface types. An X10 closure literal is mapped to a C++ class whose instance fields are the captured lexical environment of the closure. The closure body is implemented by an instance method of the C++ class. The generated closure class implements the appropriate function type interface. Closure instances are heap allocated. If the optimizer is able to propagate a closure literal to a program point where it is evaluated, the closure literal's body is unconditionally inlined. In many cases this means that the closure itself is completely eliminated as well.

4.4.4 Generics

Generic types in X10 are implemented by using C++'s template mechanism. Compilation of a generic class or struct results in the definition of a templatized C++ class. When the generic type is instantiated in the X10 source, a template instantiation happens in the generated C++ code.

The performance of an X10 generic class is very similar to that of a similar C++ templatized class. In particular, instantiation based generics enable X10 generic types instantiated on primitives and structs to be space efficient in the same way that a C++ template instantiated on a primitive type would be.

4.4.5 Memory Management

On most platforms Native X10 uses the Boehm-Demers-Weiser conservative garbage collector as its memory manager. A runtime interface to explicitly free an object is also available to the X10 programmer. The garbage collector is only used to automatically reclaim memory within a single place. The BDWGC does not yield the same level of memory management performance as that of the memory management subsystem of a modern managed runtime. Therefore, when targeting Native X10 the application programmer may need to be more conscious of avoiding short-lived objects and generally reducing the application's allocation rate.

Because the X10 v2.2.3 implementation does not include a distributed garbage collector, if a `GlobalRef` to an object is sent to a remote place, then the object (and therefore all objects that it transitively refers to) become uncollectable. The life-time of all multi-place storage must currently be explicitly managed by the programmer. This is an area of the implementation that needs further investigation to determine what mix of automatic distributed garbage collection and additional runtime interfaces for explicit storage control will result in the best balance of productivity and performance while still maintaining memory safety.

4.4.6 Other Considerations

In general, Native X10 inherits many of the strengths and weaknesses of the C++ performance model. C++ compilers may have aggressive optimization levels available, but rarely utilize profile-directed feedback. C++ compilers are generally ineffective at optimizing non statically-bound virtual function calls. Over use of object-oriented features, interfaces, and runtime type information is likely to reduce application performance more in Native X10 than it does in Managed X10.

The C++ compilation model is generally file-based, rather than program-based. In particular, cross-file inlining (from one .cc file to another) is performed fairly rarely and only at unusually high optimization levels. Since the method bodies of non-generic X10 classes are mostly generated into .cc files, this implies that they are not easily available to be inlined except within their own compilation unit (X10 file). Although for small programs, this could be mitigated by generating the entire X10 application into a single .cc file, this single-file approach is not viable for the scale of applications we need Native X10 to support.

4.5 Final Thoughts

Clearly, the performance models described in this chapter are not the final and definitive X10 performance model. However, we do believe that the language specification and its implementations are well-enough understood that it is possible for significant X10 programs to be written and for programmers to obtain predictable and understandable performance behavior from those programs. As the X10 implementations continue to mature, we expect to be able to eliminate some of the less desirable features of the X10 v2.2.3 performance models.

We hope that the open discussion of our design decisions in implementing X10 and their implications for its performance will be useful to the X10 programmer community and to the broader research community that is engaged in similar language design and implementation projects.

Part II

Programming Examples

In this part we discuss several of the PERCS benchmark and their implementation in X10. The code for the benchmarks discussed here is a slight cosmetic variation of the code actually run on the PERCS machine. Both the benchmark code [42] and the X10 release [41] that was used to execute them on the PERCS machine have been released. We discuss scalability issues, and outline performance considerations for various idioms which motivate one way of realizing the problem in X10 versus another.

We are primarily interested in *parallel efficiency*. The parallel efficiency at P cores is the number of operations performed per unit time by P cores, divided by the product of P and the number of operations performed per unit time by a single core. Ideally the ratio would be 1, in practice a ratio above 0.95 is considered good.

The benchmarks are discussed in two groups. §5 discusses the "Hello World" problem and what it takes to write the code in such a way that it scales to tens of thousands of cores. Also discussed is the HPC Streams benchmark. This tests that an X10 computation can achieve the expected memory bandwidth. The implementation of this benchmark discussed here uses X10 teams, and tests X10 support for huge pages and for team broadcast.

§6 discusses several programs best expressed in the "Single Program Multiple Data" (SPMD) style, and shows how this style can be expressed in X10. Discussed are:

1. RandomAccess (also called the "GUPS" benchmark): This benchmark measures the capability of the tool-chain to saturate the network with read-modify-write operations performed on remote memory. This is a valuable basic benchmark that approximates the behavior of various distributed graph algorithms. The implementation uses X10 teams, broadcasts, huge pages, and also uses custom memory allocation for "congruent" distributed arrays (arrays scattered across multiple places with the same local virtual memory address across all places).

2. KMeans: This is an unsupervised learning benchmark that can be expressed with teams and broadcasts in X10.

3. FT: This is the 2-d "Fast Fourier Transform" benchmark. It requires significant local computation and all-to-all communication. The benchmark is expressed in X10 using teams, broadcasts and custom memory allocation.

4. LU: This benchmark implements the LU decomposition algorithm for matrices. It uses teams, broadcasts, custom memory allocation, huge pages, also RDMA transfer (through async copy), and pragmas.

Finally §7 discusses the Unbalanced Tree Search (UTS) benchmark. This measures the ability of X10 to balance highly irregular work across many places.

The remaining two benchmarks – SSCA1 and SSCA2 – are not discussed since they use essentially the same techniques discussed in the other benchmarks.

5 Basic Multi-Place Idioms

5.1 HelloWholeWorld

We will start by describing the X10 implementation of the APGAS extension to the classic HelloWorld program: print one message to the console from every Place in the computation. The code below shows how one can write this in X10:

```
/** An APGAS Hello; print a message at every place */
public class HelloWholeWorld {
  /** //
   * writes first argument to the console at each Place
   * @param args the command line arguments
   */ //
  public static def main(args:Rail[String]) {
    finish
      for (p in Place.places())
        at (p)
          async
            Console.OUT.println("(At " + p + ") : " + args(0));
  }
}
```

The program may be read quite simply as: "At each place asynchronously print the given string, and wait until they are all done." In more detail, when this program is executed, a single activity at place 0 starts executing the body of the `main` method, in an environment in which `args` is bound to the array of strings read in from the command line. This activity sets up a `finish` scope, and for each place `p` launches an activity at that place. The body of this activity prints out a string formed from `p` and the first argument passed in to the method.

Note that `args` is defined at place 0 but accessed from all places. The compiler and run-time determined that `args` is a value at place 0 that is referenced within the body of an `at`; hence the `at` is translated into a message with a payload that carries the value of `args`. Similarly, the compiler will also determine that the value of `p` will need to be included in the message payload.

One can improve this program by noticing that it is not necessary to transmit the entire array `args` to other places; only its zero'th value is needed. Hence it is better to extract this value into a local variable and reference this variable from the body of the `at`. One can also avoid the transmission of `p` entirely by replacing its use with the expression `here`, which simply evaluates to the Place in which the activity is executing.

```
  public static def main(args:Rail[String]) {
    finish
      for (p in Place.places()) {
        val arg = args(0);
        at (p)
          async
            Console.OUT.println("(At " + here + ") : " + arg);
      }
  }
}
```

The improved version of HelloWorld shown above will work well at moderate scale (hundreds of places), but at larger scale will start to suffer reduced scalability because:

- the main activity running at place 0 is sequentially sending point-to-point messages to each other place. As the number of places increases, this activity will become a sequential bottleneck.
- the runtime will by default use a general implementation of a multi-place `finish`. However this particular code fragment is using a restricted form of parallelism in which exactly one activity is being created at each place. This allows a specialized and significantly more scalable implementation of `finish` to be used instead.

The simplest way of rewriting the program to scale well is to use some of the APIs provided by the standard library class `PlaceGroup` as shown below:

```
// Option 1: Use built-in broadcast facility in PlaceGroup
{
val arg = args(0);
PlaceGroup.WORLD.broadcastFlat(()=>{
  Console.OUT.println("(At " + here + ") : " + arg);
});
```

In this program, the work to be performed is bundled into a closure and passed as an argument to the `broadcastFlat` method of the `WORLD` `PlaceGroup`. The implementation of this method will ensure that the closure is executed exactly once on each place in the `PlaceGroup` before it returns. Although using this built-in facility is quite convenient, there may be reasons for the programmer to want to more explicitly express the computation. Controlling the implementation of `finish` can be done via a pragma mechanism, as shown below:

```
// Option 2: Use pragma to optimize finish implementation for better scalablity
val arg = args(0);
@Pragma(Pragma.FINISH_SPMD) finish
```

```
  for (p in Place.places())
    at (p)
      async
        Console.OUT.println("(At " + here + ") : " + arg);
```

Here the pragma indicates that a `finish` implementation optimized for the special case of a single activity per place should be used. To make the program fully scalable, it is also necessary to parallalize the initial spawning of the activities, as shown by the chunked loop below.

```
// Option 3: Parallelize initiation by chunking and use a pragma
// to optimize finish implementation for better scalablity.
// This is exactly how PlaceGroup.WORLD.broadcastFlat is implemented.
val arg = args(0);
@Pragma(Pragma.FINISH_SPMD) finish
  for(var i:Int=Place.numPlaces()−1; i>=0; i−=32) {
    at (Place(i))
      async {
        val max = Runtime.hereInt();
        val min = Math.max(max−31, 0);
        @Pragma(Pragma.FINISH_SPMD) finish
          for (var j:Int=min; j<=max; ++j) {
            at (Place(j))
              async
                Console.OUT.println("(At " + here + ") : " + arg);
          }
      }
  }
```

This nested loop with two levels of finishes scales well. In fact this is exactly how the `broadcastFlat` method of `WORLD` is implemented in the X10 standard library. However, it is significantly more complex than any of the previous variants of HelloWorld. This complexity makes it harder to separate the core computation from the boilerplate code needed to ensure scalable task initiation. Therefore we recommend using X10's support for function types and closures to define utility methods like `broadcastFlat` to encapsulate scalable communication and concurrency patterns for use in application general code.

5.2 Stream

5.2.1 Problem

The purpose of the Stream benchmark is to measure the sustainable memory bandwidth and corresponding computation rate for simple vector kernels. It does this by performing the caclulation `a[i] = b[i] + Beta * c[i]` over large vectors `a`, `b`, and `c`.

5.2.2 Solution design

The implementation of this benchmark in X10 follows a straight-forward SPMD style of programming. The main activity launches an activity at every place using the `broadcastFlat` utility method described in the previous section. These activities then allocate and initialize the arrays, perform the computation, and verify the results. To maximize memory system performance, the backing storage for the arrays should be allocated using huge pages to enable efficient usage of TLB entries.

5.2.3 Code

The core of the X10 implementation of Stream is shown below; the utility methods `now` for timing and `printStats` are elided.

```
static MEG = 1024*1024;
static alpha = 3.0D;
static NUM_TIMES = 10;
static DEFAULT_SIZE = 64 * MEG;
static NUM_PLACES = Place.MAX_PLACES;

static def makeHugeArray(size:Int)=
    new Array[Double](IndexedMemoryChunk.allocateZeroed[Double](size,8,
    IndexedMemoryChunk.hugePages()));

public static def main(args:Array[String](1)){here == Place.FIRST_PLACE} {
    val verified = new Cell[Boolean](true);
    val times = GlobalRef[Array[Double](1)](new Array[Double](NUM_TIMES));
    val N0 = args.size>0? Int.parse(args(0)) : DEFAULT_SIZE;
    val N = (N0 as Long) * NUM_PLACES;
    val localSize = N0;

    Console.OUT.println("localSize=" + localSize);

    PlaceGroup.WORLD.broadcastFlat(()=>{
            val p = here.id;
            val a = makeHugeArray(localSize);
            val b = makeHugeArray(localSize);
            val c = makeHugeArray(localSize);
            for (var i:Int=0; i<localSize; i++) {
                b(i) = 1.5 * (p*localSize+i);
                c(i) = 2.5 * (p*localSize+i);
            }

            val beta = alpha;

            for (var j:Int=0; j<NUM_TIMES; j++) {
                if (p==0) {
                    val t = times as GlobalRef[Array[Double](1)]{self.home==here};
                    t()(j) = -now();
                }
                for (var i:Int=0; i<localSize; i++)
                    a(i) = b(i) + beta*c(i);
                Team.WORLD.barrier(here.id);
                if (p==0) {
                    val t = times as GlobalRef[Array[Double](1)]{self.home==here};
                    t()(j) += now();
                }
            }


            for (var i:Int=0; i<localSize; i++)
                if (a(i) != b(i) + alpha*c(i))
                    verified.set(false);
        });

    var min:Double = 1000000;
53        for (var j:Int=1; j<NUM_TIMES; j++)
54            if (times()(j) < min)
55                min = times()(j);
56        printStats(N, min, verified());
57    }
```

Between lines 20 and 50 is the definition of the SPMD code that will execute at every place. The main points of interest in this code are are:

Array allocation (lines 7): The three arrays are allocated using a constructor of the `Array` class that allows the caller to provide the backing memory (the `IndexedMemoryChunk`) that should be used to store the array's data. This memory was itself allocated using the `allocateZeroed` method of `IndexedMemoryChunk` with the value of the third argument indicating whether or not the native memory should be allocated using huge pages.

Vector computation (lines 37 – 38): These two lines are the inner loop that actually performs the required vector operations.

Barrier (line 39): The `barrier` method of `Team` is used to synchronize the activities running in each place after the computational kernel is completed. Teams will be discussed in more detail in Section 6.1.3.

6 Optimizing Communication

6.1 Memory allocation and network hardware

6.1.1 Enabling RDMA and Asynchronous Copy

RDMA (Remote Direct Memory Access) hardware, such as InfiniBand, enables the transfer of segments of memory from one machine to another without the involvement of the CPU or operating system. This technology significantly reduces latency of data transfers, and frees the CPU to do other work while the transfer is taking place. To use RDMA hardware, the application needs to register the memory segments eligible for transfer with the network hardware, and issue transfer requests as background tasks with a completion handler to signal when the transfer is complete. In X10, the main mechanism to do this is via the `Array.asyncCopy()` method, which performs these operations for you, if RDMA hardware is available.

6.1.2 Customized Memory Allocation

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

6.1.3 Teams and other hardware optimizations

X10 teams offer capabilities similar to HPC collectives, such as Barrier, All-Reduce, Broadcast, All-To-All, etc. Some networks support these well-known communication patterns in hardware, and some simple calculations on the data is supported as well.

When the X10 runtime is configured for these systems, the X10 team operations will map directly to the hardware implementations available, offering performance that can not be matched with simple point-to-point messaging.

6.2 Random Access

6.2.1 Problem

The Random Access (RA) benchmark measures small updates to random memory locations within a giant table, by performing XOR operations on those locations. Because the table is spread across many places, any update to a random location in that table is likely to be an update to memory located at a remote place, not the local place. The more places, the more likely this is. Performance is measured by how many GUPS (Giga Updates Per Second) the system can sustain.

6.2.2 Solution Design

Our implementation takes advantage of the RDMA and customized memory allocation described above, allocating memory buffers at the same location in all places, and registering this with the RDMA hardware. We also make use of the calculation abilities within the network hardware, to perform the XOR operations not on the CPU, but instead in the network interface of the remote machine. These optimizations make the Random Access benchmark run primarily on the network, leaving the CPU free to spend its time generating the next random address to update.

6.2.3 Code

The code below shows how to allocate congruent memory that can later be used in remote memory operations, via the `IndexedMemoryChunk.allocateZeroed(_ ,_ , true)` method call.

```
// create congruent array (same address at each place)
val plhimc = PlaceLocalHandle.makeFlat(PlaceGroup.WORLD,
        () => new Box(IndexedMemoryChunk.allocateZeroed[Long](localTableSize, 8, true))
        as Box[IndexedMemoryChunk[Long]]{self!=null});
PlaceGroup.WORLD.broadcastFlat(()=>{
    for ([i] in 0..(localTableSize−1)) plhimc()()(i) = i as Long;
});
```

The code below shows the core computation.

```
static def runBenchmark(plhimc: PlaceLocalHandle[Box[IndexedMemoryChunk[Long]]
            {self!=null}], logLocalTableSize: Int, numUpdates: Long) {
    val mask = (1<<logLocalTableSize)−1;
    val local_updates = numUpdates / Place.MAX_PLACES;
```

```
 5      val max = Place.MAX_PLACES;
 6
 7      PlaceGroup.WORLD.broadcastFlat(()=>{
 8          val jj = Runtime.hereInt();
 9          var ran:Long = HPCC_starts(jj*(numUpdates/Place.MAX_PLACES));
10          val imc = plhimc()();
11          val size = logLocalTableSize;
12          val mask1 = mask;
13          val mask2 = Place.MAX_PLACES - 1;
14          val poly = POLY;
15          val lu = local_updates;
16          for (var k:Long=0 ; k<lu ; k+=1L) {
17              val place_id = ((ran>>size) as Int) & mask2;
18              val index = (ran as Int) & mask1;
19              val update = ran;
20              if (place_id==jj) {
21                  imc(index) ^= update;
22              } else {
23                  imc.getCongruentSibling(Place(place_id)).remoteXor(index, update);
24              }
25              ran = (ran << 1) ^ (ran<0L ? poly : 0L);
26          }
27      });
28  }
```

The key line is 23. `getCongruentSibling` returns a `RemoteIndexedMemoryChunk` at the given place. The `remoteXor` method call performs an asynchronous, atomic, remote `xor` operation on this location, but in "uncounted" mode (see §4.3.3), i.e. as if with the code:

On the PERCS machine, this kind of remote update is supported directly in hardware, through reliable messaging. This method call is implemented natively, with a call to the PAMI function that exposes this hardware functionality. Key to making this work is the congruent array functionality. Note that the code above does not implement a local update atomically, but it does use HFI hardware for remote updates, and the hardware will perform those updates atomically. To perform the local updates atomically, the HFI should be used for local updates as well.

Note that the timed portion of the code measures the cost of local updates and the cost of issuing the remote atomic updates. It does not measure time to completion of these issued updates. In principle, it is possible to examine some hardware counters and measure how many updates completed successfully, from each place, and aggregate these numbers to get an accurate count. The reference UPC code does not perform such a measurement, and neither does the X10 code.

There is aother plausible way to check how many operations are completed. One starts with a known configuration, times the run of two rounds of updates, and then runs a third round of reads and counts how many cells have a value different from the value in the known configuration. If no updates were lost, and all updates were completed, the result would be zero. The UPC team (led by George Almasi) has performed these tests on this hardware and obtained a very small number, significantly less than the 1%

permitted by the benchmark. Further, the time taken by the UPC code for two rounds was almost precisely twice the time taken by the UPC code for one round.

Given that the UPC code and the X10 code perform exactly the same operation at the hardware level – issue remote atomic update operations to the HFI – we believe that in practice this is evidence that (almost) all operations are successfully completed within the timed portion of the code. (Note that in principle it is possible that the second update could happen after the termination of the second round of issuance, and before the read in the third round. We do not believe this potential discrepency is significant.)

6.3 KMeans: Unsupervised clustering

6.3.1 Problem

K-Means Clustering is a common benchmark, which sorts a large number of data points into clusters, by minimizing the sum of squares of distances between data and the corresponding cluster centroids. We use an iterative implementation, where the number of points, clusters, dimensions, and iterations are provided at the start, and initial point values and cluster means are chosen at random. In each iteration, all points will be sorted into whichever cluster centroid is closest to that point's value, the resulting cluster centroid is recalculated, and this repeats.

6.3.2 Solution design

Our implementation spreads the points evenly across all X10 places, and each place is responsible for assigning its set of points to clusters. To do this, every place needs to know what the existing cluster mean values are at the beginning of an iteration, and after assigning points to clusters, the new cluster means need to be calculated from the new assignments made at every place. We use the AllReduce team operation before and between each iteration of the algorithm to do this multi-place calculation of the mean of each cluster. The AllReduce operation allows a simple calculation to be made across values stored in many places, and the result is provided to all places. Our implementation performs an ADD operation to count number of points in each cluster, and an ADD to sum up the values of all points in each cluster.

6.3.3 Code

KMeans is written in SPMD-style, where every place executes the same sequence of operations, on its own local set of data. The initialization phase, after reading in command-line arguments, uses `PlaceGroup.WORLD.broadcastFlat(()=>{..})` to mark the point where computation changes from place 0 to all places. We initialize data structures and random values for all of the points, and blindly assign them to clusters. We also initialize `Team.WORLD`, which is a pre-defined team consisting of all X10 places. Finally, we use the `team.allreduce()` operation to `ADD` the values of the points that every place has put into the same cluster. The `team.allreduce()` also acts as a barrier, ensuring that every place has reached this point in the algorithm, and is ready to begin the iterations.

```
PlaceGroup.WORLD.broadcastFlat(()=>{
    val role = Runtime.hereInt();
    val random = new Random(role);
    val host_points = new Rail[Float](num_slice_points*dim,
            (Int)=>random.next());

    val host_clusters = new Rail[Float](num_clusters*dim);
    val host_cluster_counts = new Rail[Int](num_clusters);

    val team = Team.WORLD;

    if (role == 0) {
        for (var k:Int=0; k<num_clusters; ++k) {
            for (var d:Int=0; d<dim; ++d) {
                host_clusters(k*dim+d) = host_points(k+d*num_slice_points);
            }
        }
    }

    val old_clusters = new Rail[Float](num_clusters*dim);

    var compute_time:Long = 0;
    var comm_time:Long = 0;

    team.allreduce(role, host_clusters, 0, host_clusters, 0, host_clusters.size,
            Team.ADD);
// ...
});
```

Within each iteration, every place will compare the value of each point with the mean of every cluster (calculated previously), and possibly re-assign a point from one cluster to another. After this reshuffle is done locally at every place, each place enters a pair of `team.allreduce()` calls, which re-calculates the sum of all point values per cluster, and the number of points per cluster across all places. These values provide the new means, and the iteration repeats until computation is complete.s

```
for (var iter:Int=0; iter<iterations; ++iter) {

    Array.copy(host_clusters, old_clusters);
    host_clusters.clear();
    host_cluster_counts.clear();

    val compute_start = System.nanoTime();
    for (var p:Int=0; p<num_slice_points; p+=8) {
        val closest = Vec.make[Int](8);
        val closest_dist = Vec.make[Float](8);
        for (var w:Int=0; w<8; ++w) closest(w) = -1;
        for (var w:Int=0; w<8; ++w) closest_dist(w) = 1e37f;
        for (var k:Int=0; k<num_clusters; ++k) {
            val dist = Vec.make[Float](8);
            for (var w:Int=0; w<8; ++w) dist(w) = 0.0f;
            for (var d:Int=0; d<dim; ++d) {
                val tmp = Vec.make[Float](8);
                for (var w:Int=0; w<8; ++w) {
                    tmp(w) = host_points(p+w+d*num_slice_points) − old_clusters(k*dim+d);
                }
                for (var w:Int=0; w<8; ++w) {
                    dist(w) = dist(w) + tmp(w) * tmp(w);
                }
            }
            for (var w:Int=0; w<8; ++w) {
                if (dist(w) < closest_dist(w)) {
                    closest_dist(w) = dist(w);
                    closest(w) = k;
                }
            }
        }
        for (var d:Int=0; d<dim; ++d) {
            for (var w:Int=0; w<8; ++w) {
                val index = closest(w)*dim+d;
                host_clusters(index) += host_points(p+w+d*num_slice_points);
            }
        }
        for (var w:Int=0; w<8; ++w) ++host_cluster_counts(closest(w));
    }
    compute_time += System.nanoTime() − compute_start;

    val comm_start = System.nanoTime();
    team.allreduce(role, host_clusters, 0, host_clusters, 0, host_clusters.size,
            Team.ADD);
    team.allreduce(role, host_cluster_counts, 0, host_cluster_counts, 0,
            host_cluster_counts.size, Team.ADD);
    comm_time += System.nanoTime() − comm_start;

    for (var k:Int=0; k<num_clusters; ++k) {
        for (var d:Int=0; d<dim; ++d) host_clusters(k*dim+d) /= host_cluster_counts(k);
    }

    if (role == 0) {
        Console.OUT.println("Iteration: " + iter);
        if (verbose) printClusters(host_clusters, dim);
    }
}
```

6.4 Fourier Transform

6.4.1 Problem

We implement the Fast Fourier Transform algorithm, which transforms time-based data into frequency-based data. For example, the conversion of a measured sound wave into the set of individual frequencies that make up that sound wave.

6.4.2 Solution design

Our implementation uses a 2D parallel algorithm, where the data is stored in a 2D array, with each place handling a single row of the array. Each place performs FFT calculations on its local row, then transposes the row into a form suitable for an All-To-All team operation, to swap results with other places. After the All-To-All, the data is transposed back into the form necessary for FFT, and more calculations are made on the local row. This pattern is repeated until all of the steps of FFT are completed.

The 2D array allows us to make use of parallelism, but it also means that a large amount of data needs to be exchanged after each phase. So we pay particular attention to optimizing the movement of that data, both by using an All-To-All team operation, and by optimizing for RDMA within the All-To-All. Using the All-To-All is straightforward – it means that we organize the data in the buffers into the form required by the All-To-All operation. But to optimize for RDMA, we need to make use of the congruent memory allocator described in the RA program above. As a reminder, the congruent memory allocator allocates memory buffers at the same location in all places, so that memory addresses do not need to be exchanged, and it registers those memory locations with the RDMA hardware. Depending on the system, an All-To-All operation may benefit from an initial warm-up operation, to initialize any data structures or other components of the network before the main calculation. So we include a warmup in our program as well.

6.4.3 Code

Initialization is straightforward, using the `makeFlat()` and `broadcastFlat()` idioms to initialize each place, and start a SPMD-style execution via the `FT.run()` method.

```
val plh = PlaceLocalHandle.makeFlat[FT](PlaceGroup.WORLD, ()=>new FT(nRows,
        localSize, N, SQRTN, verify));

PlaceGroup.WORLD.broadcastFlat(()=>{plh().run();});
```

Memory allocation uses our congruent memory allocator via the last `true` argument to `IndexedMemoryChunk.allocateZeroed()`, which allocates the A array at the same memory location in every place. Same for the B array. Internally, this also registers the arrays with the RDMA hardware, so that their contents can be transmitted via RDMA.

```
A = new Rail[Double](IndexedMemoryChunk.allocateZeroed[Double](localSize,
        8, true));
B = new Rail[Double](IndexedMemoryChunk.allocateZeroed[Double](localSize,
        8, true));
```

The `FT.alltoall()` method uses the `Team.WORLD.alltoall()` team with those A and B arrays, swapping them after the call.

```
def alltoall() {
    Team.WORLD.alltoall(I, B, 0, A, 0, 2 * nRows * nRows);
    val C = B;
    B = A;
    A = C;
}
```

6.5 LU Decomposition

6.5.1 Problem

LU stands for Lower-Upper, and this benchmark is also called High Performance Linpack (HPL). The implementation is a LU factorization of a large square matrix, which is broken into tiles across X10 places. It uses alternating phases of local matrix multiplication with All-To-All communications to distribute changes at the tile-edges that cross place boundries.

6.5.2 Solution design

LU is primarily a CPU-bound benchmark, where the matrix multiplication calculation dominates, and network communication time is secondary. Our implementation makes use of IBM's ESSL library, which is similar to the more familiar BLAS libraries, which enables maximum CPU performance with well-known math patterns. The ESSL library is provided as a C-style header and library, which gets linked into the X10 binary through `@NativeCPPExtern` method annotations, and the use of X10 primitives that map directly onto C-style primitives.

For network communications, LU differs from the above benchmarks in that it uses many parallel X10 teams made up of a subset of places, one team per row and column of the overall matrix, instead of a world team. So each place, handling a single tile, is a member of two teams. The communication pattern has phases of calculation, parallel row swaps (with all row teams communicating at the same time), and parallel column swaps. This benchmark shows examples of several new Team operations worth highlighting, as well as optimizations such as the congruent memory allocation and RDMA-enabled `Array.asyncCopy()` which have already been covered in previous benchmarks.

6.5.3 Code

Our benchmark starts out demonstrating the import and wrapping of the ESSL library, through the `@NativeCPP` family of annotations. We also show the use of the `blockTriSolve` further down in the program.

```
@NativeCPPInclude("essl_natives.h")
@NativeCPPCompilationUnit("essl_natives.cc")
class LU {

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

    @NativeCPPExtern
        native static def blockTriSolveDiag(diag:Rail[Double],
                min:Int, max:Int, B:Int):void;
    // ...

    def triSolve(J:Int, timer:Timer) {
        if (A_here.hasRow(J)) {
            var tmp:Rail[Double];
            if (A_here.hasCol(J)) tmp = A_here.block(J, J).raw;
            else tmp = colBuffer;
            val diag = tmp;
            timer.start(10);
            row.bcast(rowRole, J%py, diag, 0, diag, 0, diag.size);
            timer.stop(10);
            for (var cj:Int = J + 1; cj <= NB; ++cj) {
                if (A_here.hasCol(cj)) {
                    blockTriSolve(A_here.block(J, cj).raw, diag, B);
                }
            }
        }
    }
    // ...
}
```

We create our large matrix as a set of tiles allocated via the congruent memory allocator at each place, named `buffers`. The constructor, which is invoked in every place as a part of the `PlaceLocalHandle.makeFlat()` call, shows the creation of the local X10 teams named `col` and `row`. These teams are created via splitting the `WORLD` team into segments.

```
public static def main(args:Rail[String]) {
    if (args.size < 4) {
        Console.OUT.println("Usage: LU M B px py bk");
        Console.OUT.println("M = Matrix size,");
        Console.OUT.println("B = Block size, where B should perfectly divide M");
        Console.OUT.println("px py = Processor grid, where px*py = nplaces");
        Console.OUT.println("bk = block size for panel, where bk should divide B");
        return;
    }
    // ...
    val A = BlockedArray.make(M, N, B, B, px, py);
    val buffers = PlaceLocalHandle.makeFlat[Rail[Double]{self!=null}]
            (Dist.makeUnique(), ()=>new Rail[Double]
            (IndexedMemoryChunk.allocateZeroed[Double](N,
            8, IndexedMemoryChunk.hugePages())));
    val lus = PlaceLocalHandle.makeFlat[LU](Dist.makeUnique(),
            ()=>new LU(M, N, B, px, py, bk, A, buffers));
    Console.OUT.println ("LU: M " + M + " B " + B + " px " + px + " py " + py);
    start(lus);
}

def this(M:Int, N:Int, B:Int, px:Int, py:Int, bk:Int,
        A:PlaceLocalHandle[BlockedArray],
        buffers:PlaceLocalHandle[Rail[Double]{self!=null}]) {
    this.M = M; this.N = N; this.B = B; this.px = px; this.py = py; this.bk = bk;
    this.A = A; A_here = A();
    this.buffers = buffers; buffer = buffers();
    remoteBuffer = new RemoteArray(buffer);
    MB = M / B - 1;
    NB = N / B - 1;
    colRole = here.id % px;
    rowRole = here.id / px;
    col = Team.WORLD.split(here.id, rowRole, colRole);
    row = Team.WORLD.split(here.id, colRole, rowRole);
    pivot = new Rail[Int](B);
    rowForBroadcast = new Rail[Double](B);
    val rowBuffers = new Rail[Rail[Double]{self!=null}](M / B / px + 1,
            (Int)=>new Rail[Double](IndexedMemoryChunk.allocateZeroed[Double](B*B,
            8, IndexedMemoryChunk.hugePages())));
    val colBuffers = new Rail[Rail[Double]{self!=null}](N / B / py + 1,
            (Int)=>new Rail[Double](IndexedMemoryChunk.allocateZeroed[Double](B*B,
            8, IndexedMemoryChunk.hugePages())));
    this.rowBuffers = rowBuffers;
    this.colBuffers = colBuffers;
    rowBuffer = rowBuffers(0);
    remoteRowBuffer = new RemoteArray(rowBuffer);
    colBuffer = colBuffers(0);
}
```

7 Unbalanced computations

Hitherto we have considered problems that can be partitioned statically across multiple places.

We shall now consider a class of problems for which such a static partitioning is not feasible. Many problems can be cast in the form of a *state-space search*. Examples can be found in many areas – such as game-playing, planning, problem-solving, puzzle-solving etc.

Consider for example the N-Queens problem discussed earlier. We can conceptualize it as a certain kind of a search process. The state of the search can be summarized in a data-structure called a *configuration*. In the case of N-Queens a configuration is simply the current state of the chess board. This can be summarized by the current placement of queens – a sequence of at most N integers, the ith integer representing the column in which the ith queen is placed. A configuration can be classified as *good* or *bad*. A good configuration satisfies all the application-specific invariants, a bad one does not. In the case of N-Queens each configuration must be such that no two assigned queens should be on the same column or left or right diagonal. If a configuration is good, it may generate zero or more additional configurations. For N-queens, if the configuration already represents N queens placed on a board then the configuration is considered final, i.e. generates zero additional configurations. Otherwise it generates as many new configurations as the number of squares on which queens can be placed in the next row without violating the stated constraints.

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

Practical applications typically have a few additional complications. Sometimes the configuration generated from a good configuration may already have been discovered, i.e. the space of configurations is a graph and not a tree. Hence it is necessary to determine whether the newly generated configuration has already been encountered or not. Sometimes – as in the case of state-space searches performed during game play, such as chess – one is looking for the "best" next move, not necessarily all possible moves. So some form of min/max search technique may be used to prune good configurations that are not as good as other configurations currently being considered.

Regardless of these complications, state-space search problems typically satisfy a few properties:

- Each configuration can be compactly represented (say in a few kilo-bytes or less).
- Usually, some non-trivial amount of computation is necessary to determine the next configurations from a good configuration.
- The number of configurations generated from a good configuration may be very difficult to predict statically.

The challenge at hand, therefore, is to parallelize the computation across a potentially very large number of places while achieving high parallel efficiency. The computation is typically initiated at a single place (usually place 0) with the *root* configuration. A good solution must look to quickly divide up the work across all available places. It must ensure that once a place runs out of work it is able to quickly find work, if in fact work exists at any place in the system. That is, a good solution must solve the *global load-balancing* problem.

The material in this chapter is excerpted from [28].

7.1 Unbalanced Tree Search

In this section we shall focus on a particular problem, the Unbalanced Tree Search (UTS) problem that was designed specifically as a benchmark for global load balancing [22, 6]. The details in the next section are taken from these references.

7.1.1 Problem

The problem is to find the number of nodes in a unbalanced tree. Given is the state of the root node in the tree and various parameters that control the behavior of the tree generation process. Two kinds of trees are specified *binomial trees* and *geometric trees* as follows:

> A node in a binomial tree has m children with probability q and has no children with probability $1 - 1$, where m and q are parameters ... When $qm < 1$, this process generates a finite tree with expected size $1/(1-qm)$. Since all nodes follow the same distribution, the trees generated are self-similar and the distribution of tree sizes and depths follow a power law ... The variation of subtree sizes increases dramatically as qm approaches 1. This is the source of the tree's imbalance. A binomial tree is an optimal adversary for load balancing strategies, since there is no advantage to be gained by choosing to move one node over another for load balance: the expected work at all nodes is identical.

On the other hand geometric trees have somewhat more regular structure. Their depth is bounded by a parameter d.

> The nodes in a geometric tree have a branching factor that follows a geometric distribution with an expected value that is specified by the parameter $b_0 > 1$. The parameter d specifies its maximum depth cut-off, beyond which the tree is not allowed to grow ... The expected size of these trees is $(b_0)^d$, but since the geometric distribution has a long tail, some nodes will have significantly more than b_0 children, yielding unbalanced trees ... the expected size of the subtree rooted at a node increases with proximity to the root.

7.1.2 Solution Design

The central problem to be solved is load balancing.

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

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

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

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

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

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

The only question left is to determine the assignment of buddies to a place. We are looking for a directed graph that is fully connected (so work can flow from any vertex to any other vertex) and that has a low diameter (so latency for work distribution is low) and has a low degree (so the number of buddies potentially sendng work to a dead vertex is low). z-dimensional hyper-cubes satisfy these properties and have been implemented.

7.1.3 Code

Utilities The UTS code uses several fixed size stacks (a simple abstraction over `IndexedMemoryChunk`, IMC), and a queue also built over IMCs.

`APlaceLocalHandle` is created with a `UTS` object at each place. This carries the state needed for a worker to execute `UTS` code, namely the fields:

```
val queue:Queue;
val lifelineThieves:FixedSizeStack[Int];
val thieves:FixedSizeStack[Int];
val lifelines:Rail[Int];
val lifelinesActivated:Rail[Boolean];

val n:Int;
val w:Int;
val m:Int;

val random = new Random();
val victims:Rail[Int];
val logger:Logger;

@x10.compiler.Volatile transient var active:Boolean = false;
@x10.compiler.Volatile transient var empty:Boolean;
@x10.compiler.Volatile transient var waiting:Boolean;
```

At every place `h`, `lifelineThieves` is pre-populated as if lifeline visits had already been made by the places `3*h+i` for `i` in `1..3`. This permits the work from place `0` to be rapidly disseminated to all places in an initial wave of outdegree 3 (therefore taking $log_3(P)$ to get to all places).

The main entry point for the timed portion of the program is:

```
def main(st:PlaceLocalHandle[UTS], seed:Int) {
    @Pragma(Pragma.FINISH_DENSE) finish {
        try {
            active = true;
            logger.startLive();
            queue.init(seed);
            processStack(st);
            logger.stopLive();
            active = false;
            logger.nodesCount = queue.count;
        } catch (v:Throwable) {
            error(v);
        }
    }
}
```

It starts the computation within a `finish` and initializes work at place `0` – this immediately triggers the first propagation wave that distributes work over the entire graph, as discussed above. The heart of the work is done by `processStack`:

```
@Inline final def processAtMostN() {
    var i:Int=0;
    for (; (i<n) && (queue.size>0); ++i) {
        queue.expand();
    }
    queue.count += i;
    return queue.size > 0;
}

@Inline static def min(i:Int,j:Int) = i < j ? i : j;

final def processStack(st:PlaceLocalHandle[UTS]) {
    do {
        while (processAtMostN()) {
            Runtime.probe();
            distribute(st);
        }
        reject(st);
    } while (steal(st));
}
```

This expands upto `n` nodes (taken from the nodes stored in `queue`), then calls `probe()` to check for (and process) incoming events, and then checks to see if work needs to be distributed. If no more work is left (as determined by termination of the `while` loop), any visiting thieves are told there is no more work, and the activity starts steal processing.

Let us discuss the steal path first.

```
def steal(st:PlaceLocalHandle[UTS]) {
    if (P == 1) return false;
    val p = Runtime.hereInt();
    empty = true;
    for (var i:Int=0; i < w && empty; ++i) {
        ++logger.stealsAttempted;
        waiting = true;
        logger.stopLive();
        at (Place(victims(random.nextInt(m)))) @Uncounted async st().request(st, p, false);
        while (waiting) Runtime.probe();
        logger.startLive();
    }
    for (var i:Int=0; (i<lifelines.size) && empty && (0<=lifelines(i)); ++i) {
        val lifeline = lifelines(i);
        if (!lifelinesActivated(lifeline)) {
            ++logger.lifelineStealsAttempted;
            lifelinesActivated(lifeline) = true;
            waiting = true;
            logger.stopLive();
            at (Place(lifeline)) @Uncounted async st().request(st, p, true);
            while (waiting) Runtime.probe();
            logger.startLive();
        }
    }
    return !empty;
}
```

Up to `w` attempts are made (in sequence) to steal from other places. A steal is executed by sending a message (through an `at`) statement. The message can be marked as `Uncounted` because the only `async` it triggers will be one coming back with a reply, and the current activity will remain alive until this reply is processed. After sending the message, the activity unters a `probe` loop waiting for a response. The response (see `reject` processing below) will set `waiting` to `false`, releasing the activity from this loop.

If no work is received from these random steals, the activity visits the lifelines in sequence. The logic here is exactly the same as with random stealing, except that the activation of each lifeline is marked locally (so that at most one request is outstanding on the lifeline), and the target is told this is a lifeline steal.

If any work is found as a result of these attempts, the activity returns to `processStack` and continues processing. Otherwise the activity terminates. Since the life-lines have been set up, it is now up to others to activate work at this place, through a distribution.

Let us now discuss the distribution path. This is relatively simple. The current work is divided up amongst the activated lifelines and active thieves. Note that a normal theft can be responded to with an uncounted `async` (because the thief already has an activity running), but on Line linenumber the `async` must be uncounted because it is not known whether the place being woken up already has an activity running or not.

```
@Inline def give(st:PlaceLocalHandle[UTS], loot:Queue.Fragment) {
    val victim = Runtime.hereInt();
    logger.nodesGiven += loot.hash.length();
    if (thieves.size() > 0) {
        val thief = thieves.pop();
        if (thief >= 0) {
            ++logger.lifelineStealsSuffered;
            at (Place(thief)) @Uncounted async { st().deal(st, loot, victim); st().waiting = false; }
        } else {
            ++logger.stealsSuffered;
            at (Place(−thief−1)) @Uncounted async { st().deal(st, loot, −1); st().waiting = false; }
        }
    } else {
        ++logger.lifelineStealsSuffered;
        val thief = lifelineThieves.pop();
        at (Place(thief)) async st().deal(st, loot, victim);
    }
}
@Inline def distribute(st:PlaceLocalHandle[UTS]) {
    var loot:Queue.Fragment;
    while ((lifelineThieves.size() + thieves.size() > 0) && (loot = queue.grab()) != null) {
        give(st, loot);
    }
    reject(st);
}
```

The code for dealing is:

```
def deal(st:PlaceLocalHandle[UTS], loot:Queue.Fragment, source:Int) {
    try {
        val lifeline = source >= 0;
        if (lifeline) lifelinesActivated(source) = false;
        if (active) {
            empty = false;
            processLoot(loot, lifeline);
        } else {
            active = true;
            logger.startLive();
            processLoot(loot, lifeline);

            processStack(st);
            logger.stopLive();
            active = false;
            logger.nodesCount = queue.count;
        }
    } catch (v:Throwable) {
        error(v);
    }
}
```

Part III

Conclusion and Appendices

8 Conclusion

This book presents a brief overview of the X10 language, and shows how to express various concurrency idioms in such a way that the code can scale to tens of thousands of places.

This book does not illustrate the use of multiple `async`s in a given place, and hence the properties of work-stealing or “finish/join” scheduler.

Also it has focused on the Native runtime and does not discuss interoperability with Java, an important concept.

We leave these discussions for future work.

References

[1] Bowen Alpern, Anthony Cocchi, Stephen Fink, and David Grove. Efficient implementation of Java interfaces: Invokeinterface considered harmless. *ACM SIGPLAN Notices*, 36(11):108–124, November 2001. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

[2] Ganesh Bikshandi, Jose G. Castanos, Sreedhar B. Kodali, V. Krishna Nandivada, Igor Peshansky, Vijay A. Saraswat, Sayantan Sur, Pradeep Varma, and Tong Wen. Efficient, portable implementation of asynchronous multi-place programs. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 271–282, New York, NY, USA, 2009. ACM.

[3] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46:720–748, September 1999.

[4] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, October 2005.

[5] C. Danis and C. Halverson. The value derived from the observational component in an integrated methodology for the study of HPC programmer productivity. In *Proceedings of the Third Workshop on Productivity and Performance in High-End Computing*, HPCA '06, pages 11–21, 2006.

[6] James Dinan, Stephen Olivier, Gerald Sabin, Jan Prins, P Sadayappan, and Chau-Wen Tseng. Dynamic Load Balancing of Unbalanced Computations Using Message Passing. In *IPDPS 07: Parallel and Distributed Processing Symposium*, pages 1–8, Long Beach, CA, March 2007. IEEE International.

[7] Jack Dongarra, Robert Graybill, William Harrod, Robert Lucas, Ewing Lusk, Piotr Luszczek, Janice Mcmahon, Allan Snavely, Jeffrey Vetter, Katherine Yelick, Sadaf Alam, Roy Campbell, Laura Carrington, Tzu-Yi Chen, Omid Khalili, Jeremy Meredith, and Mustafa Tikir. DARPA's HPCS program: History, models, tools, languages. In Marvin V. Zelkowitz, editor, *Advances in COMPUTERS High Performance Computing*, volume 72 of *Advances in Computers*, pages 1 – 100. Elsevier, 2008.

[8] K. Ebcioglu, V. Sarkar, T. El-Ghazawi, and J. Urbanic. An Experiment in Measuring the Productivity of Three Parallel Programming Languages. In *Proceedings of the Third Workshop on Productivity and Performance in High-End Computing*, HPCA '06, pages 30–36, 2006.

[9] T. El-Ghazawi, B. Carlson, and J. Draper. Upc Language Specifications v1.1, 2003.

[10] John Richards *et al.* IBM PERCS productivity assessment report 4, 2012. Internal IBM memo.

[11] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[12] Amol Ghoting, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 231–242, Washington, DC, USA, 2011. IEEE Computer Society.

[13] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[14] David Grove, Olivier Tardieu, David Cunningham, Ben Herta, Igor Peshansky, and Vijay Saraswat. A performance model for X10 applications: what's going on under the hood? In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, X10 '11, pages 1:1–1:8, New York, NY, USA, 2011. ACM.

[15] Yi Guo, Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[16] C. Halverson and C. Danis. Towards an Ecologically Valid Study of Programmer Behavior for Scientific Computing. In *Proceedings of the First Workshop on Software Engineering for Computational Science and Engineering*, ICSE '08, 2008.

[17] Kiyokuni Kawachiya, Mikio Takeuchi, Salikh Zakirov, and Tamiya Onodera. Distributed garbage collection for managed X10. In *Proceedings of the 2012 ACM SIGPLAN X10 Workshop*, X10 '12, pages 5:1–5:11, New York, NY, USA, 2012. ACM.

[18] Doug Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. ACM.

[19] MPI Forum. MPI: A Message-Passing Interface Standard. Version 2.2. `http://www.mpi-forum.org`, September 4th 2009.

[20] Rajesh Nishtala, Paul H. Hargrove, Dan O. Bonachea, and Katherine A. Yelick. Scaling communication-intensive applications on BlueGene/P using one-sided communication and overlap. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[21] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: an extensible compiler framework for Java. In *Proceedings of the 12th international conference on Compiler construction*, CC'03, pages 138–152, Berlin, Heidelberg, 2003. Springer-Verlag.

[22] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. UTS: an unbalanced tree search benchmark. In *Proceedings of the 19th international conference on Languages and compilers for parallel computing*, LCPC'06, pages 235–250, Berlin, Heidelberg, 2007. Springer-Verlag.

[23] Jens Palsberg, editor. *X10 '12: Proceedings of the 2012 ACM SIGPLAN X10 Workshop*, New York, NY, USA, 2012. ACM.

[24] Vijay Saraswat, editor. *X10 '11: Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, New York, NY, USA, 2011. ACM.

[25] Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky, and Olivier Tardieu. The asynchronous partitioned global address space model. Technical report, Computer Science Department, U Rochester, Toronto, Canada, June 2010.

[26] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 language specification. `http://x10.sourceforge.net/documentation/languagespec/x10-223.pdf`.

[27] Vijay Saraswat and Radha Jagadeesan. Concurrent clustered programming. In *CONCUR 2005 - Concurrency Theory*, pages 353–367, London, UK, 2005. Springer-Verlag.

[28] Vijay A. Saraswat, Prabhanjan Kambadur, Sreedhar Kodali, David Grove, and Sriram Krishnamoorthy. Lifeline-based global load balancing. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 201–212, New York, NY, USA, 2011. ACM.

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

[30] A. Skjellum, E. Lusk, and W. Gropp. *Using MPI: Portable Parallel Programming with the Message Passing Iinterface*. MIT Press, 1999.

[31] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.

[32] Mikio Takeuchi, Yuki Makino, Kiyokuni Kawachiya, Hiroshi Horii, Toyotaro Suzumura, Toshio Suganuma, and Tamiya Onodera. Compiling X10 to Java. In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, X10 '11, pages 3:1–3:10, New York, NY, USA, 2011. ACM.

[33] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[34] Wikipedia. HPC Challenge Awards: 2007 Awards – Most Productive Research Implementation: X10, 2007.

[35] Wikipedia. Percs, 2011.

[36] Wikipedia. C# (programming language), 2012.

[37] Wikipedia. Eclipse (software), 2012.

[38] Wikipedia. Pig (programming_tool), 2012.

[39] Wikipedia. Publications using X10, 2012.

[40] Wikipedia. Universities using X10, 2012.

[41] X10 2.2.3 release on SourceForge. `http://sourceforge.net/projects/x10/files/x10/2.2.3/`.

[42] X10 PERCS benchmarks version 2.2.3. `http://sourceforge.net/projects/x10/files/x10/2.2.3/x10-benchmarks-2.2.3.tar.bz2`.

[43] X10RT API specification. `http://x10.sourceforge.net/x10rt/`.

[44] X10RT implementations. `http://x10-lang.org/documentation/practical-x10-programming/x10rt-implementations.html`.

[45] X10 website. `http://x10-lang.org`.

A Building X10

The X10 runtime has several build-time configuration options, which enable specific communications protocols, and reduce error checking within the X10 libraries at runtime for better performance. There is a similar set of options available when compiling your own program with the x10c++ compiler. When and how to use these is detailed here.

A.1 Network transport implementations

For communications, the X10 runtime has the following transport implementations:

- Sockets: Uses TCP/IP sockets to support multiple places on one or more hosts. This is the default implementation, and is the only option when using managed X10. The sockets transport uses SSH to launch the binaries across the network. If you have a simple cluster of machines that support Ethernet and SSH, the sockets transport is a good choice.

- Standalone: Supports multiple places on a single host, using shared memory between places. Standalone has high bandwidth, but limited message sizes and only supports places running on a single machine.

- MPI: An implementation of X10RT on top of MPI-2. This supports all the hardware that your MPI implementation supports, such as Infiniband and Ethernet, and should be used for systems where MPI is preferred. It does not (currently) use MPI's collective implementations, but instead uses our own collective implementations.

- PAMI: An IBM communications API that comes with the IBM Parallel Environment: http://www-03.ibm.com/systems/software/parallel/index.html. PAMI supports high-end networks such HFI (Host Fabric Interface), BlueGene, Infiniband, shared memory, and also Ethernet. The PAMI implementation uses PAMI's collectives, and is X10's best-performing transport. If your system has the IBM Parallel Environment installed, you'll want to use PAMI.

The X10 runtime will always be built with Sockets and Standalone transports, as no special libraries need to be available on the system to compile them. But if you wish to make use of MPI or PAMI in your program, you must build the X10 runtime from source, specifying that you want to build in support for one or both of these transports. Similarly, when building your own program, you can choose which transport to use and other options via arguments to x10c++.

A.2 Building the X10 runtime

The X10 runtime is built using ant (http://ant.apache.org/). When you are satisfied that your program is operating correctly, for best performance in your programs, build the X10 runtime with the `optimize` and `NO_CHECKS` options turned on. These will reduce the error checking within the X10 library classes at runtime. If you want to enable support for PAMI or MPI transports, turn on the `X10RT_PAMI` and/or `X10RT_MPI` options. For example: "`ant -DX10RT_PAMI=true -Doptimize=true -DNO_CHECKS=true dist`"

A.3 Building your program

You use the x10c++ compiler for building your own programs. The X10 compiler takes arguments to enable runtime optimizations, reduced error checking, and transport selection of your program source, at build-time. Use the `-O`, `-NO_CHECKS`, and `-STATIC_CHECKS` flags for the fastest performance, after you are satisfied with the correctness of your program. To choose a network transport other than sockets, specify `pami`, `mpi`, or `standalone` as the value of the `-x10rt` flag. For example: "`x10c++ -O -NO_CHECKS -STATIC_CHECKS -x10rt pami YourProgram.x10`".

B Running X10

The output of the x10c++ compiler is a standard binary executable program. Depending on which network transport chosen, different launchers and libraries will have been linked in.

Details on how to launch your program, and which environment variables control where the X10 places execute in a multi-host environment differs slightly between transports, but in general the binary can simply be executed.

Here we provide more details on the environment variables which control X10 compilation and execution, in general. We also provide information about specific environment variables for the PAMI transport, for a system using the IBM Parallel Environment. Full details can be found on the X10 website [45].

B.1 X10 Variables

The environment variables of interest are:

`X10_NTHREADS=1`: Specify the number of worker threads used in each place to run the application code. Default to 1. Set to the number of cores available per place. For instance, when running 8 places on a 32-core host, set to 4.

`X10_STATIC_THREADS=true`: Specify that the runtime should not create threads dynamically. Default to `false`, that is, permit dynamic thread creation. Set to `true` for a small performance boost for SPMD-style code with single-threaded places.

`X10RT_CPUMAP=cpumap`: Specify a file describing the mapping from places to cores. Optional. Default to none. Override the `MP_TASK_AFFINITY` setting if set. Like the host file, the cpumap file contains one line per X10 place. Line n specifies the core id for place n.

B.1.1 Congruent Memory Allocator

The X10 runtime provides a congruent memory allocator. It permits allocating memory at the same address in every place, registers this memory with PAMI, and optionally uses large pages thus enabling fast data access and transfer.

If used, the congruent memory allocator must be configured using the following environment variables:

`X10_CONGRUENT_HUGE=true`: Specify that the runtime should use large pages for the congruent memory allocator ($16M$ pages). Default to false ($64K$ pages).

`X10_CONGRUENT_BASE=0x3000000000`: Specify the base memory address for the congruent memory region.

`X10_CONGRUENT_SIZE=0x800000000`: Specify the size of the congruent memory region.

B.1.2 Optimized Collectives

On the Hurcules system, fast collective operations employing shared memory and the Collective Acceleration Unit (CAU) require both `MP_SHARED_MEMORY=yes` and `MP_COLLECTIVE_GROUPS=4`.

B.1.3 X10 compiler options for performance tuning

Programmers may find it convenient to adopt the following methodology.

When initially developing the application, run with no compiler flags set. No optimizations will be performed. Dynamic checks will silently be generated for those constraint type checks which cannot be satisfied by the compiler.

When the code is running satisfactorily, you may wish to improve performance. Setting the `-STATIC_CHECKS` option will cause the compiler to print out errors if constrained types cannot be verified statically. These can typically be fixed by examining the offending code, and adding missing constraint clauses at variable/parameter declaration sites. The advantage of doing this is that the code records more precisely the constraints that hold at run-time, and no run-time code is generated.

To enable full optimizations with the x10c++ compiler specify additionally the `-O`, `-NO_CHECKS` options. The first option turns on compiler optimization and the second turns off array bound checks.

To enable use of the PAMI transport with the x10c++ compiler specify the option `-x10rt pami`.

B.2 POE Settings

We recommend setting the following POE environment variables for optimal performance on most X10 benchmarks:

```
MP_RESD=poe
MP_EUILIB=us
MP_EUIDEVICE=sn_single
MP_DEVTYPE=hfi
MP_MSG_API=X10
MP_SHARED_MEMORY=yes
MP_CPU_USE=unique
MP_ADAPTER_USE=dedicated
MP_USE_BULK_XFER=yes
MP_TASK_AFFINITY=core
MP_COLLECTIVE_GROUPS=4
X10_STATIC_THREADS=1
X10_NTHREADS=1
X10_CONGRUENT_HUGE=1
X10RTTRANSPORT=‘‘-x10rt pami’’
```

To bind X10 places to CPU cores (to prevent the OS from migrating places to other cores). use the `X10RT_CPUMAP` environment variable. This points to a CPU map file, which like the `MP_HOSTFILE`, depends on the layout of X10 places.

X10 programs can be launched directly using POE, or can be batched using LoadLeveler. The necessary POE libraries are linked in, so the binary can be launched directly, e.g. `./a.out`, or via the POE command with `poe ./a.out`. When launching with POE, the `MP_PROCS` and `MP_HOSTFILE` environment variables should be set, to control the distribution of the program on the system. The value for `MP_PROCS` is the number of X10 places. When using LoadLeveler, the values for the number of places and layout is defined in your LoadLeveler script.

Details of exact command-line arguments for running with POE at various scales are provided in the `run` script included with every benchmark.