

X10: Programming for Hierarchical Parallelism and Non-Uniform Data Access

(Extended Abstract)

Kemal Ebcioğlu
IBM TJ Watson Research
Center
PO Box 704
Yorktown Heights, NY 10598
kemal@us.ibm.com

Vijay Saraswat
IBM TJ Watson Research
Center
PO Box 704
Yorktown Heights, NY 10598
vsaraswa@us.ibm.com

Vivek Sarkar
IBM TJ Watson Research
Center
PO Box 704
Yorktown Heights, NY 10598
vsarkar@us.ibm.com

ABSTRACT

The challenges faced by current and future-generation large-scale systems include: 1) *Frequency wall*: inability to follow past frequency scaling trends, 2) *Memory wall*: inability to support a coherent uniform-memory access model with reasonable performance, and 3) *Scalability wall*: inability to utilize all levels of available parallelism in the system. These challenges manifest themselves as both *performance* and *productivity* issues in the use of large-scale systems. X10 is an experimental modern object-oriented programming language being developed to help address the second and third of these three challenges. It is intended for high-performance, high-productivity programming of large-scale computer systems with non-uniformities in data access and coherence, as well as multiple heterogeneous levels of parallelism. This paper provides a summary of the X10 language and programming model, and discusses its applicability to future processor architectures and runtime systems.

1. INTRODUCTION AND MOTIVATION

The challenges faced by current and future-generation large-scale systems include: 1) *Frequency wall*: inability to follow past frequency scaling trends due to power and thermal limitations, 2) *Memory wall*: inability to support a coherent uniform-memory access model with reasonable performance thereby leading to severe nonuniformities in latency and bandwidth for accessing data in different parts of the system, and 3) *Scalability wall*: inability to utilize all levels of available parallelism in the system, e.g., clusters, SMPs, multiple cores on a chip, co-processors, SMT, and SIMD levels. The focus of this paper is on language, compiler, and runtime techniques to address the second and third of these three challenges, which have increased in urgency given the paucity of techniques to address the first challenge. It is now common wisdom that the ongoing increase in complex-

ity of large-scale parallel systems to address these challenges has been accompanied by a *decrease in software productivity* for developing, debugging, and maintaining applications for such machines [13]. This is a serious problem because current trends for next generation systems, including SMP-on-a-chip and tightly coupled “blade” servers, indicate that these complexities will be faced not just by programmers for large-scale parallel systems, but also by mainstream application developers.

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

During the same period, significant experience has also been gained with the design and use of commercial object-oriented languages, such as JAVA and C#. These languages, along with their accompanying libraries, frameworks and tools, have enjoyed much success in improving *productivity* for commercial applications. The object-oriented programming model has also been well-served by the emergence of *managed runtime environments*, capable of providing portability of programs across a variety of architectures. Such an environment is typically organized around a *virtual machine* and provides a number of features aimed at improving software productivity, such as type safety, value safety, and automatic memory management.

While the productivity benefits of commercial object-oriented languages are well established for single-threaded applications, the results for concurrent applications have been decidedly mixed. Despite much effort [18], attempts to pre-

Preprint: To be presented at the 3rd International Workshop on Language Runtimes: "Impact of Next Generation Processor Architectures On Virtual Machine Technologies", co-located with the ACM OOPSLA 2004 conference, Vancouver, October 2004. Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

cisely define the semantics of the memory model for JAVA – that is, the concurrent interactions between multiple threads and a single shared global heap – continue to be very complicated technically and arguably beyond the reach of most practicing programmers. With some notable exceptions (e.g. JSR 166 [15]), research on concurrency in such languages has not addressed the issue of delivering the scalable performance that is required by large-scale parallel systems.

X10 is an experimental new language currently under development at IBM in collaboration with academic partners. The X10 effort is part of the IBM PERCS project (Productive Easy-to-use Reliable Computer Systems) whose goal is to design adaptable scalable systems for the 2010 timeframe, with a technical agenda focused on hardware-software co-design that combines advances in chip technology, computer architecture, operating systems, compilers, programming environments and programming language design. The main role of X10 is to simplify the programming model so as to increase the programming productivity for future systems like PERCS, without degrading performance. Combined with the PERCS Programming Tools agenda [23], the ultimate goal is to use a new programming model and a new set of tools to deliver a 10× improvement in development productivity for large-scale parallel applications by 2010.

To increase programmer productivity, X10 starts with a state-of-the-art object-oriented programming model, and then raises the level of abstraction for constructs that are expected to be amenable to automatic static and dynamic optimizations by 2010 — specifically, X10 introduces *atomic sections* in lieu of locks, *clocks* in lieu of barriers, and *asynchronous operations* in lieu of threads. To increase performance transparency, X10 integrates new constructs (notably, *places*, *regions* and *distributions*) to model hierarchical parallelism and non-uniform data access.

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

The rest of the paper is organized as follows. In Section 2, we introduce the core features of the X10 programming model. In Section 3, we discuss how programs expressed in X10 can take advantage (through a suitable compiler and runtime system) of various architectural features. In Section 4, we outline compiler optimizations that we expect will be important in an X10 context. Finally, Section 5 contains our conclusions.

2. X10 PROGRAMMING MODEL OVERVIEW

This section provides a brief summary of the X10 language, focusing on the core features that are most relevant to locality and parallelism. Figure 1 contains a schematic overview of the main X10 constructs for concurrency and data access.

A central concept in X10 is that of a *place*. A place is a collection of resident light-weight threads (called *activities*)

and *data*, and is intended to map to a data-coherent unit in a large scale system such as an SMP node or a single co-processor. It contains a bounded, though perhaps dynamically varying, number of *activities* and a bounded amount of storage. Cluster-level parallelism can be exploited in an X10 program by creating multiple places.

There are four storage classes in an X10 program:

1. *Activity-local* — this storage class is private to the activity, and is located in the place where the activity executes. The activity's stack and thread-local data are allocated in this storage class.
2. *Place-local* — this storage class is local to a place, but can be accessed coherently by all activities executing in the same place.
3. *Partitioned-global* — this storage class represents a *unified or global address space*. Each element in this storage class has a unique place that serves as its home location, but the element is accessible by both local activities (activities in the same place) and remote activities (activities in a different place).
4. *Values* — Instances of *value classes* (value objects), are immutable and stateless in X10, following the example of Kava[2]. Such value objects are in this storage class. Since value objects do not contain any updatable locations, they can be freely copied from place to place. (The choice of when to clone, cache or share value objects is left to the implementation.) In addition, methods may be invoked on such an object from any place.

X10 activities operate on two kinds of *data objects*. A *scalar* object has a small, statically fixed set of fields, each of which has a distinct name. The mutable state of a scalar object is located at a single place. An *aggregate* (array) object has many elements (the number may be known only when the object is created), uniformly accessed through an index (e.g. an integer) and may be distributed across many places. Specifically, an X10 array specifies 1) a set of indices (called a *region*) for which the array has values, 2) a *distribution mapping* from indices in this region to places, and 3) the usual array mapping from each index in this region to a value of the given base type (which may itself be an array type). Operations are provided to construct regions (distributions) from other regions (distributions), and to iterate over regions (distributions). These operations include standard set-based operations such as unions and intersections, some of which are available in modern languages such as ZPL [4]. It is also worth noting that commonly-used basic types such as `int`, `float`, `complex` and `string` are defined as *value classes* in the `x10.lang` standard library, rather than as primitive types in the language.

Throughout its lifetime an activity executes at the same place, and has direct access only to data stored at that place. Remote data may be accessed by spawning asynchronous activities at the places at which data is resident. An asynchronous activity is a statement of the form `async (P) S` where `S` is a statement and `P` is a place expression. Such a statement is executed by spawning an activity at the place designated by `P` to execute statement `S`. As a convenient means of identifying the place of a datum in the partitioned-global storage class, when the expression `P` specifies an array

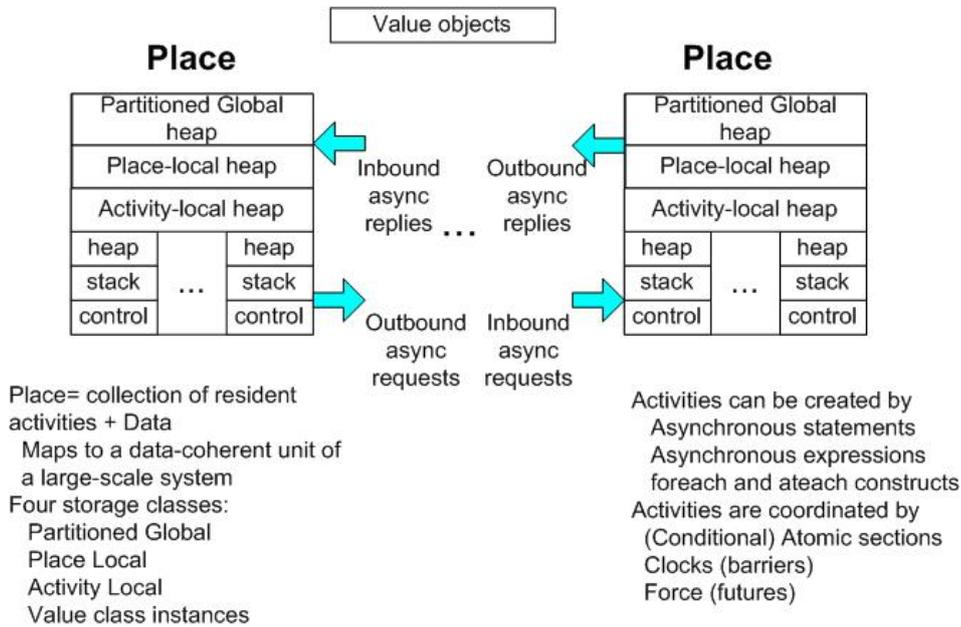


Figure 1: Overview of X10 constructs for concurrency and data access.

element or object, it evaluates to the place containing that array element or object. (P) can also be omitted, in which case, it is inferred to be the place of the data accessed by statement S (provide that a single place can be unambiguously inferred). Asynchronous activities that may return a value to the invoking activity are called *futures* and are discussed further below. Activities can also be spawned in the local place as a high-level abstraction of multithreading. The *foreach* construct serves as a convenient mechanism for spawning local activities across a specified index set (*region*). The *ateach* (pronounced “at each”) construct serves as a convenient mechanism for spawning activities across a set of local/remote places or objects.

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

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

A clock is defined as a special value class instance, on which only a restricted set of operations can be performed. At any given time an activity is *registered* with zero or more clocks. The activity that creates a clock, is automatically registered with this clock. An activity may register other activities with a clock, or may un-register itself with a clock. An activity may also *quiesce* on the clocks it is registered with and suspend until all of them have advanced (by executing the blocking *next* operation), or require that a statement (possibly involving execution of new *async* activities) be executed to completion before a clock can advance. At

any given step of the execution a clock is in a given *phase*. The first phase of the clock starts when the clock is created. The clock *advances* to its next phase only when all its currently registered activities have quiesced, and all statements scheduled for execution in this phase have terminated. In this manner, clocks serve as *barriers* for a dynamically varying collection of activities.

Force Operations. When an activity *A* executes the statement, $F = \text{future}(P) E$, it asynchronously spawns an activity *B* at the place designed by P to evaluate the expression E. Execution of the expression in *A* terminates immediately, yielding a *future* [9] in F, thereby enabling *A* to perform other computations in parallel with the evaluation of E. *A* may also choose to make the future stored in F accessible to other activities. When any activity wishes to examine the value of the expression E, it invokes a *force* operation on F. This operation blocks until *B* has completed the evaluation of E, and returns with the value thus computed.

X10 does not allow the invoking activity, *A*, to register the spawned activity *B* with any of the clocks *A* is registered with. Further, E is not allowed to invoke a conditional atomic section.

These restrictions are sufficient to ensure that the evaluation of a future can never deadlock even if the expression E creates its own clocks and uses futures and unconditional atomic sections.

Unconditional Atomic Sections. A statement block or method that is qualified as *atomic* has the semantics of being executed by an activity as if in a single step, during which all other activities are frozen¹. Thus atomic sections may

¹The implementation may of course allow concurrent execution of atomic sections, using techniques such as optimistic concurrency, as long as atomic sections are made to appear

be thought of as executing in some global sequential order, even though this order is indeterminate. An atomic section is a generalization of user-controlled locking, so that the user only needs to specify that a collection of statements should execute atomically and can leave the responsibility of lock management and other mechanisms for enforcing atomicity to the language implementation. Primitives such as fetch-and-add, updates to histogram tables, updates to a bucket in a hash table, airline seat reservations arriving at an online data base, and many others, are a natural fit for synchronization using atomic sections. X10 also requires that all accesses to shared mutable data be inside an atomic section, which can ease the constraints on the memory consistency model.

It is important to avoid including long-running or blocking operations in an atomic section. To this end, we distinguish between *intra-place* and *inter-place* atomic sections as follows: all data accessed in an intra-place atomic section reside in the same place as the activity executing the atomic section, whereas data accessed in an inter-place atomic section may reside in a remote place. Since inter-place data accesses can result in long-running or blocking operations, we expect intra-place atomic sections to be used more frequently than inter-place atomic sections. In addition, we call an atomic section *analyzable* if the locations and places of all data to be accessed in the atomic section can be computed on entry to the atomic section [21]. An X10 programmer can reasonably expect that the overhead of supporting atomic sections on current and future systems will increase progressively for the following four classes of atomic sections: 1) analyzable intra-place, 2) non-analyzable intra-place, 3) analyzable inter-place, 4) non-analyzable inter-place, with 1) being the most efficient and 4) being the most expensive. We expect implementations of 1) and 2) to be comparable to implementations of *wait-free* data-structures.

Conditional atomic sections. Conditional atomic sections in X10 are akin to conditional critical regions [12], and have the form `when (c) S`. If the guard `c` is false in the current state, the activity executing the statement blocks until `c` becomes *true*. Otherwise, as far as any other concurrently executing activity is concerned, the statement is executed *in a single step* which begins with the evaluation of `c = true`, and ends with the completion of statement `S`. This implies that `c` is not allowed to change between the time it is detected to be true and the time `S` begins execution. X10 currently does not permit the statement `S` to contain or invoke a nested conditional atomic section.

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

A number of other features in X10 are not mentioned here due to space limitations. These include generic interfaces, generic classes, type parameters, sub-distributions, array constructors, exceptions, place casts and the nullable type constructor.

Figure 2 contains an overview of our implementation plan for the compiler and runtime environment for X10 programs. A static high-level optimizer is used to perform the optimizations outlined in Section 4. The X10 runtime system and dynamic compiler is part of the X10 virtual machine,

to execute in a “single step” to the programmer.

which interfaces with the PERCS Continuous Program Optimization (CPO) system. Both the static optimizer and the virtual machine use profile feedback and information on the target hardware to make their optimization decisions.

3. EXPLOITING ARCHITECTURAL FEATURES

In this section we will briefly discuss how X10 can be used to exploit parallel hardware architectures. The following ideas are presented only as a general discussion on X10 and its interaction with architectures, and are not intended to represent an implementation plan for a specific architecture.

3.1 Scalable Parallelism with a Large Number of Places

A paramount concern in scalable parallel computation is *memory address to physical location mapping*. In very early, cacheless computer systems, a memory address used to denote the actual physical location of a datum. In modern ILP, SMP, and/or SMT architectures, a datum is moved from physical location to physical location (registers, any cache level in any processor, memory modules), using caching assumptions and heuristics (such as spatial and temporal locality, lack of false sharing) as well as register allocation algorithms that allow memory locations to reside in registers temporarily. With coherent cache memory systems, the address-to-location mapping can get arbitrarily complex, and may involve an expensive broadcast query. This may look up all the L2 cache directories in the system, consume a lot of bandwidth and power, and can effectively cause serialization because of resource contention. Alternatively (with a directory-based cache protocol) one can implement a slower, up to three-hop lookup that first asks a fixed “home memory module” of a datum for the current owner of the datum, and then can read the datum from the current owner processor’s L2 cache. Despite a number of optimizations that can be implemented, such coherent cache designs are difficult to scale to very high degrees of parallelism, and the caching heuristics and assumptions are not always reliable.

In the MPI programming model, the problem of address-to-location mapping is left to the programmer, which can be an error-prone task. The burden of guaranteeing coherence also falls on the programmer, thereby introducing another major source of programming complexity and errors. Following the paradigm of programming languages such as UPC and Titanium[7, 11], X10 simplifies the problem of address-to-location space through the introduction of a *partitioned global address space*. Through distribution mappings, the X10 programmer assigns a unique place to each mutable shared datum in the partitioned-global space. During the execution of any X10 activity, the hardware/compiler can map any X10 global memory address to a *simplified*, easy-to-compute specification of its physical location, of the form: `<placeId, address within place>`. In many cases the lookup can be performed statically by the compiler. No system wide query is necessary to determine the current physical location of a global memory address. This approach offers a high degree of scalability and high performance in large scale-out systems with a high speed interconnect.

To avoid slowdown in the case of repeated communication between places that happen to be physically far apart, a second level place-to-physical-node mapping could be used

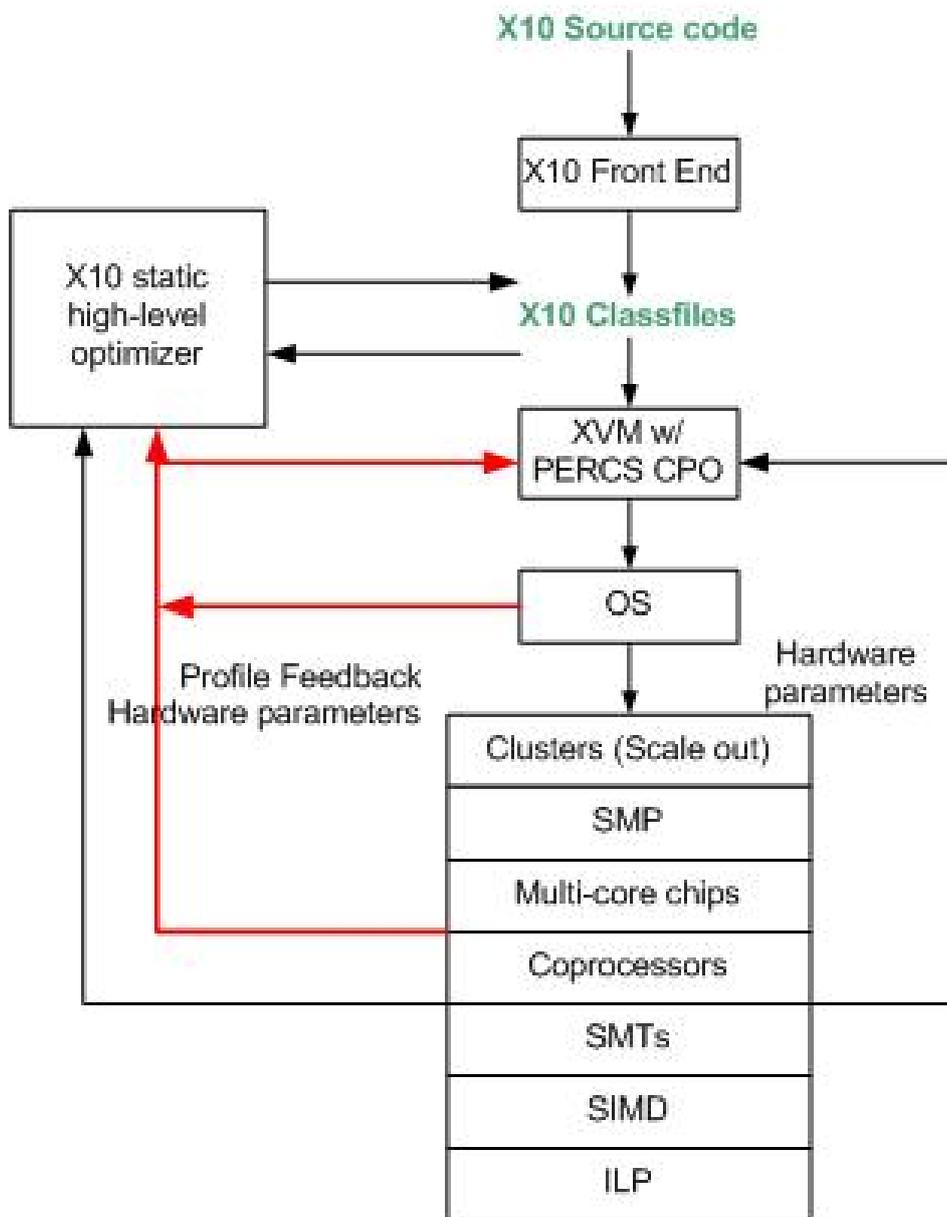


Figure 2: Schematic design of X10 implementation

(with the help of a runtime dynamic optimization system) to remap such frequently communicating places to physically closer nodes, while taking load balancing into consideration.

A key goal in the design of distributed parallel systems and software is locality, i.e., the minimization of communication between remote nodes. By making the programmer aware of the cost of remote communication, through the use of explicit language constructs for remote accesses, the X10 design is well-suited to future-generation processor implementations where inter-node communication and synchronization are expected to be significantly slower and more expensive compared to intra-node communication and synchronization.

3.2 Support for Heterogeneous Computing

With an ultra light-weight VM implementation that ensures code portability across heterogeneous platforms, and the ability to define new simple data types and operations as value classes (following the example of Kava[2]), X10 can directly execute special hardware operations (e.g. graphics functions) natively on the platforms that have the proper native hardware, while using efficient JIT compilation on other machines that do not have the native hardware. Thus, an X10 place and its activities can be mapped to heterogeneous physical nodes, taking advantage of the native hardware features of each node, while retaining portability of code across the heterogeneous platforms.

Heterogeneous hardware configurations that can benefit from the X10 programming model include specialized co-processors in a system on-a-chip environment that are not memory coherent, PIM-like multithreaded processors capable of tolerating high DRAM latencies, and dedicated storage and I/O processors which are close to the disks and to each other but are far from the main computation processors.

3.3 Implementing Fast Communication

X10's remote activity-spawning constructs may be supported very efficiently by high speed/highly pipelinable messaging hardware, and/or in software, for communication-intensive applications. A remote activity invocation from place i to place j can be achieved by sending a work queue item (a message) from i to j containing the starting address of the activity to spawn as a new activity on j , along with the parameters to be used by the new activity. The message is inserted in a single work queue in j in the order the messages are sent from i to j . The new activity at j will cease when an end-of-activity primitive is executed, returning a result if the activity was a `future`. For the case of a `future`, j must send a return message with the return value. i will in turn dequeue items from the future work queue and wake up the light-weight threads in i that may be waiting for the future result to arrive.

An important class of X10 applications can be transformed by compiler techniques into long running activities in each place, which communicate by fine grain, data flow style send-receive primitives. High performance synchronized send-receive communication hardware can be used to implement such applications.

3.4 Achieving ILP and Thread-level Parallelism in a Single Place

A key obstacle to achieving strong scalability in current

applications is that it is hard to uncover multiple levels of parallelism in an MPI programming model based on *domain decomposition*. Future systems that require 10^5 -way parallelism will exhibit poor performance on current MPI applications that can only over 10^3 -way useful parallelism (say), even after scaling up the data size. Also, since the original MPI algorithm is often fine-tuned to minimize inter-task communication, if one were able to break up the original MPI tasks one could also expose unusually high communication needs among the new parallel sub-tasks, thus requiring an architecture paradigm outside of the slow message passing infrastructure normally connecting processors/memories. A second obstacle to achieving strong scalability is Amdahl's law. For example, if the original program spent, say, 0.1% of its time in unparallelized code, this would limit performance to only 1000X, even if the remaining 99.9% of the program took zero time. There is therefore a potential benefit in fine-grain parallelization of serial sections and MPI tasks in general.

X10 can offer a solution for increasing instruction-level and thread level parallelism within a place. At the lower levels of the parallelism hierarchy, within a single place, X10 offers explicit parallelism constructs of the form:

```
future T x = future(P){B}
A;
T y = x.force();
```

which specifies that B and A have no dependences (except for any atomic sections in them) and can be executed in parallel. Assume that the place P is known to be the same place as the current one i.e., $P = \text{here}$. To implement this construct, an SMT architecture may start executing the new activity body B with an independent program counter and also continue the current activity with A. An SMP architecture may spawn the new activity B on a currently idle processor, and continue the current activity with A. In a statically scheduled architecture such as a VLIW[6] or EPIC[5] machine, or in the semi-statically scheduled TRIPS architecture[19], the compiler may "snip off" the dependence edges going from B to A in its conservatively constructed dependence graph, and schedule operations as usual. For a scalable superscalar architecture with a wide issue window [8], the compiler may insert a special instruction just before A that forces instructions in A not to depend on instructions in B, but to possibly depend on instructions that precede `future{B}`. The explicit parallelism constructs are very useful for achieving aggressive, uninhibited parallel execution, which would not always be possible through automatic parallelization, since the latter would need to (1) automatically group the original code into B and A and (2) prove the independence of B and A, at compile time.

Higher level X10 parallelism constructs (such as parallel loops) can be expressed as multiple invocations of simpler primitives that spawn a single activity, so the observations here can be generalized to higher level constructs as well.

3.5 Implementing Synchronization Efficiently

Sometimes one would like to write a consumer operation without having to guarantee that the producer operation is executed first. An individual cell in a mesh computation where the cell expects inputs from some of its sides, and produces outputs at its remaining sides, is a typical example, where one would like to write the code for the behavior of the cell, without knowing if the cell inputs have already

been computed or not. For example, a data flow buffer implemented with a conditional atomic section as well as data structures built using `future`'s are all suitable for efficiently expressing this kind of producer-consumer synchronization in X10. The implementation techniques range from hardware-supported global P/V semaphores to software simulated I-structures (involving a ready bit, a value and a waiting threads queue for each future datum). I-structures were previously used in data flow machines.

Techniques such as thread-level speculation [24] may be used to efficiently implement atomic sections, with hardware support. This approach maintains “speculatively written” bits in cache lines written during the execution of an atomic code fragment, and either rolls back by discarding the cache line when a data race is detected, or commits by removing the speculative state from the cache line. Other hardware/software approaches to atomic sections have recently been described in [10, 1].

Classical barrier network hardware (a simple AND gate) can be used to implement a barrier when the compiler can statically establish the set of activities that will participate in a clock synchronization. The general case is implemented through a global counting semaphore. Combining networks could be used (as in the RP3 parallel computer, [17]) to update counting semaphores efficiently, in order to prevent the semaphores from becoming high-contention hot-spots.

Overall, we believe that X10 will fit today’s distributed architectures (as well as their enhanced future versions) well. It has the potential to provide high degrees of cluster level, thread level, and instruction level parallelism. However, there are still many challenges in implementing atomic sections, fast asynchronous thread spawning, fast transmission of results, and fast activity group synchronization in a scalable manner.

4. COMPILER OPTIMIZATIONS

The compiler plays a crucial role in X10’s productivity goals. State-of-the-art parallelizing compiler techniques can in many cases transform an X10 program that provides a high level specification of *what* should be done, to a more efficient program (perhaps another X10 program), that describes *how* it should be done. Figure 3 shows an example of such a transformation. Version 1 of the code describes a parallel array copy operation (between arrays with unequal distributions) at a high level; this X10 program (albeit inefficient if implemented literally) is clear and easy to code and understand. Version 3 of the code (which can also be derived from version 1 by a state-of-the-art compiler) describes a more efficient form of array copying, which aggregates messages, and minimizes the total number of inter-node communication events. Version 3 is also a clear and easily comprehensible code fragment, thanks to the very high level parallelization constructs in X10. However, for achieving productivity goals, such as reducing the time for obtaining the first correct parallel program with reasonable performance, or facilitating the rapid teaching of X10 to a group of non-expert programmers, it is important to provide state-of-the-art compiler techniques among the productivity arsenal for X10, that can perform code transformations as in Figure 3 automatically. Note that the best production-level performance most probably will still be achieved by manual optimization of a parallel algorithm, and we have striven to make this manual optimization process as productive as

possible for X10, by design.

In the present section, we will describe a subset of state-of-the-art compiler techniques applicable to X10. Due to space limitations, we outline two main classes of optimizations — inter-place optimizations, and optimization of clocks and atomic sections. In addition, we briefly discuss the benefits that accrue to compiler optimization from X10’s strong typing and single-assignment (functional) programming features. There are several more opportunities for optimization of X10 programs that are beyond the scope of this paper e.g., optimization of value classes, optimization of data distributions and redistributions, optimization of streams and arrays, intra-place optimizations, and runtime optimizations

4.1 Inter-place optimizations

The X10 programming model supports the notion of a global address space distributed across places. A single place is assumed to map to a data-coherent hardware unit of a large scale parallel system, such as a Symmetric Multi-Processor (SMP) chip or a single processor core with support for thread-level and instruction-level parallelism. In contrast, some degree of software support is necessary to support a consistency model for data accesses across places. In this section, we outline important optimizations for X10 programs that use multiple places.

One of the strengths of X10 is that it can be used as a single language that supports multiple programming paradigms. Table 1 highlights the range of programming paradigms that can be followed when using multiple places in X10, and the important optimizations that need to be performed in each case. Many of these optimizations have been developed in past work on specific programming paradigms, and can be leveraged for X10 as well.

The most convenient mechanism for inter-place interactions in X10 is through fine-grained *asynchronous operations*, which include *asynchronous statements* (one-way messages) and *asynchronous expressions* (futures). While this capability can aid programmer productivity, typical hardware support for inter-place interactions is better suited to fewer large-grained communications than many fine-grained communications. It is therefore very important to provide compiler optimizations for aggregation of async operations.

The basic rule for aggregation is that two dynamic async operations can be combined if they have the same source and destination places, and if program dependences do not require that some other intervening statement be performed between the two operations. There are two special cases of aggregation that are important to exploit for X10:

1. *Loop-level aggregation* — in this case a single *foreach* or *for* loop is strip-mined/tiled so as to ensure that each tile executes in the same place, and all instances of the same async operation in a tile have the same destination. In the absence of additional program dependence constraints, it should then be possible to combine all instances of that async operation within the tile into a single coarse-grained async operation. As described above, Figure 3 shows three equivalent versions of a simple Array Copy example, where versions 2 and 3 are obtained by performing loop-level aggregation on version 1.
2. *Object-level aggregation* — in this case, certain object reference fields are analyzed as being “local” i.e., the

Programming Paradigm	Activities	Storage Classes	Important Optimizations
Message-passing e.g. MPI	Single activity per place	Place-local	Domain decomposition, message aggregation, optimization of barriers & reductions
Data parallel e.g. HPF	Single global program	Partitioned-global	SPMDization, communication & synchronization optimizations
PGAS e.g. Titanium, UPC	Single activity per place	Partitioned-global, place-local	Localization, SPMDization, communication & synchronization optimizations
DSM e.g. TreadMarks	Multiple (w/ implicit place affinity)	Partitioned-global, activity-local	Data Layout, page locality optimizations
NUMA	Multiple (w/ explicit place affinity)	Partitioned-global, activity-local	Data distribution, consistency & synchronization optimizations
Co-processor e.g. STI Cell	Multiple (w/ implicit & explicit place affinity)	Partitioned-global, place-local	Consistency & synchronization optimizations
Active messages / futures	Multiple (w/ explicit place affinity)	Place-local, Activity-local	Message aggregation, synchronization optimizations
Full X10	Multiple	Partitioned-global, place-local, activity-local	All of the above

Table 1: Important Optimizations for Programming Paradigms that use Multiple Places

target of the object reference is guaranteed to be in the same place as the object containing the field. Once again, it should be possible to combine all async operations from the same place performed on objects reachable through local fields.

These rules form the basis for a task-partitioning algorithm that enforces the *acyclicity constraint* [22, 20] so as to ensure that each thread is non-blocking.

4.2 Optimization of Clocks and Atomic Sections

X10 *activities* are created by using asynchronous operations and *foreach/ateach* concurrent statements. *Clocks* and *atomic sections* are the two primary mechanisms used to coordinate activities. The usage of clocks and atomic sections helps improve programmer productivity, but can potentially lead to large overheads when used across multiple places. Past approaches (e.g., [25]) have highlighted opportunities for compiler optimization of barriers either by replacing them by equivalent weaker coordination primitives such as pair-wise or one-to-all synchronizations, or by eliminating the barriers outright. These approaches can be extended to optimization of clocks.

A promising opportunity for optimizing atomic sections across places is to replace them by *user-defined reduction operations*. There are two key restrictions that need to be imposed on atomic sections to enable this transformation: 1) the atomic section should be *analyzable* as defined in [21], and 2) multiple instances of the same atomic section should be commutative and associative.

4.3 Optimization Benefits of Strong Typing and Single-Assignment features in X10

The strongly typed X10 language, along with interprocedural type inference techniques that can further identify

the run-time type of an object, permit better static memory address disambiguation of object references (especially compared to C/C++ and Fortran). Better static memory disambiguation can in turn result in fewer perceived dependences between memory accesses and higher instruction-level, thread-level and cluster-level parallelism. Also, the use of *ownership types* [3] in X10 can simplify *escape analysis* and determination of which data accesses are local.

While allowing the user all the benefits of an imperative, multiple assignment-based, object-oriented language with mutable data structures, X10 also makes it easy to write methods and other code fragments in a functional, single-assignment style. Some of the X10 language features that facilitate the single-assignment style include immutable value classes, final and clocked final variables, and array constructors. Also, even when a code fragment written by a programmer contains mutable data structures with multiple assignment, it is sometimes possible for a compiler to convert a code fragment to a pure functional form through static single assignment transformations design for arrays and other aggregate data structures [14].

The functional style is easy to recognize for a compiler, and has a number of well-known benefits for parallel execution. When a method or code fragment written in the functional style is examined through inter-procedural analysis, it is possible to prove the lack of side effects, which can enable higher levels of redundancy elimination and parallelism transformations than in traditional imperative languages: for example, a parallelizing compiler can spawn functional method calls as independent parallel activities, or can eliminate entire function calls from frequently executed paths when they are proved to be partially redundant or partially dead. Also, in an SMP multiprocessor context, when the last assignment to a (clocked) final variable is executed on one processor, it can be broadcast to the other processors that use its value as a shared read-only variable from that

<pre> // VERSION 1: // Creates a total of 2*N // activities, one for each // index in 0..N-1, // and one for each future. int N = ... ; region R = [N] ; distribution D1 = block(R); distribution D2 = cyclic(R); // D1 and D2 have the // same region, R . . . // a and b have same // element type T T[D1] a; T[D2] b; . . . ateach (i:a) { // for each index i // of a do in parallel: // In the place of a[i], // remotely read b[i], // write its value to a[i] . . . // Note: // As a convenience, // a[i]=!~b[i]; or // a[i]=b[i]; // can serve as a // shorthand for // the synchronous remote // read: // a[i] = future(b[i]) // .force(); a[i] = b[i]; } </pre>	<pre> // VERSION 2: // Creates a total of P+N // activities, one for each // of the P places, and one // for each future. int N = ... ; region R = [N] ; distribution D1 = block(R); distribution D2 = cyclic(R); // D1 and D2 have the // the same region, R . . . // a and b have same // element type T T[D1] a; T[D2] b; . . . ateach(distribution .unique(D1.places)) { // At each place // which contains elems // of a, do in parallel: // Current place="here" . . . // for all indices j such // a[j] is in "here" // (serial for loop) for (j : D1 here) { // copy b[j] from // whatever place it is // in, to a[j] in "here" a[j] = future(b[j]) .force(); } } </pre>	<pre> // VERSION 3: // Creates a total of P+P*P // activities, one for each // <src,dest> place pair int N = ... ; region R = [N] ; distribution D1 = block(R); distribution D2 = cyclic(R); // D1 and D2 have the // same region, R. . . . // a and b have same // element type T T[D1] a; T[D2] b; . . . ateach(distribution .unique(D1.places)) { // At each place // which contains elems // of a, do in parallel: // Current place="here" // LD1={i a[i] in "here"} region LD1 = (D1 here).region; // for each place p // containing any b elems // with indices in LD1 for(p:D2[LD1]) { region LD2 = (D2 p).region; region Common = LD1 && LD2; // copy all the b elems // with indices in LD1 // from place p, to the // corresponding a elems // in place "here" // (message aggregation) a[Common]= future(b[Common]) .force(); } } </pre>
---	--	---

Figure 3: Example of loop-level aggregation for Array Copy example

point on. Immutable variables that are assigned once, eliminate the overhead of the usual coherence issues of keeping multiple replicated copies of a datum in different caches or memory modules and can create several opportunities for additional optimization [16].

5. CONCLUSION

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

We expect the next version of the language to be significantly informed by experience in implementing and using the language, and through productivity studies comparing X10 with existing programming models such as MPI, OpenMP, and UPC. Our directions for future work include constructs that support inter-place activity migration, continuous program optimization, and fault tolerance.

Acknowledgments

X10 is being developed in the context of the IBM PERCS (Productive Easy-to-use Reliable Computing Systems) project, which is funded by DARPA. We are grateful to the following people for their feedback and contributions to the design and implementation of X10: David Bacon, Bob Blainey, Philippe Charles, Perry Cheng, Julian Dolby, Guang Gao, Christian Grothoff, Allan Kielstra, Robert O'Callahan, Filip Pizlo, Christoph von Praun, V.T. Rajan, Lawrence Rauchwerger, Mandana Vaziri, and Jan Vitek. We are also grateful to John McCalpin, Mootaz Elnozahy and Lisa Spainhower for helpful discussions.

6. REFERENCES

- [1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *11th International Symposium on High Performance Computer Architecture (HPCA-11)*, San Francisco, CA, February 2005. To appear.
- [2] D. Bacon. Kava: A Java dialect with a uniform object model for lightweight classes. *Concurrency – Practice and Experience*, 15:185–206, 2003.
- [3] C. Boyapati, B. Liskov, and L. Shriram. Ownership types for object encapsulation. In *POPL '03. Proceedings of the 30th ACM SIGPLAN-SIGACT on Principles of programming languages*, New York, NY, USA, 2003. ACM Press.
- [4] B. L. Chamberlain, S.-E. Choi, S. J. Deitz, and L. Snyder. The high-level parallel language ZPL improves productivity and performance. In *Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing*, 2004.
- [5] C. Dulong. The IA-64 architecture at work. *IEEE Computer*, 31(7):24–32, July 1998.
- [6] K. Ebcioglu. Some design ideas for a VLIW architecture for sequential natured software. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Processing*, pages 3–21. North Holland, 1988. Michel Cosnard et al., editors.
- [7] T. El-Ghazawi, W. Carlsson, and J. Draper. UPC Language Specification v1.1.1, October 2003.
- [8] M. Galluzzi, V. Puente, A. Cristal, R. Beivide, J.-A. Gregorio, and M. Valero. A first glance at kilo-instruction based multiprocessors. In *Proceedings of the first conference on computing frontiers on Computing frontiers*, pages 212–221. ACM Press, 2004.
- [9] R. Halstead. MULTILISP: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7:501–538, 1985.
- [10] L. Hammond, V. Wong, M. Chen, B. Hertzberg, B. Carlstrom, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31th Annual International Symposium on Computer Architecture (ISCA-31)*, Munich, June 2004.
- [11] P. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, and K. Yelick. Titanium Language Reference Manual. Technical Report CSD-01-1163, University of California at Berkeley, Berkeley, Ca, USA, 2001.
- [12] C. Hoare. Monitors: An operating system structuring concept. *CACM*, 17(10):549–557, October 1974.
- [13] HPL Workshop on High Productivity Programming Models and Languages, May 2004. <http://hplws.jpl.nasa.gov/>.
- [14] K. Knobe and V. Sarkar. Array SSA form and its use in Parallelization. In *POPL '98. Proceedings of the 25th ACM SIGPLAN-SIGACT on Principles of programming languages*, Jan. 1998.
- [15] D. Lea. The Concurrency Utilities, 2001. JSR 166, <http://www.jcp.org/en/jsr/detail?id=166>.
- [16] I. Pechtchanski and V. Sarkar. Immutability Specification and its Applications. *Proceedings of the ACM 2002 Java Grande/ISCOPE Conference*, October 2002.
- [17] G. Pfister, W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, V. Norton, and J. Weiss. The Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771. IEEE Press, August 1985.
- [18] W. Pugh. Java Memory Model and Thread Specification Revision, 2004. JSR 133, <http://www.jcp.org/en/jsr/detail?id=133>.
- [19] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of*

the 30th International Symposium on Computer Architecture, 2003.

- [20] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989. In the series, Research Monographs in Parallel and Distributed Computing.
- [21] V. Sarkar and G. R. Gao. Analyzable atomic sections: Integrating fine-grained synchronization and weak consistency models for scalable parallelism. Technical report, CAPSL Technical Memo 52, February 2004.
- [22] V. Sarkar and J. Hennessy. Partitioning Parallel Programs for Macro-Dataflow. *ACM Conference on Lisp and Functional Programming*, pages 202–211, August 1986.
- [23] V. Sarkar, C. Williams, and K. Ebcioglu. Application development productivity challenges for high-end computing. In *Proceedings of Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, February 2004.
<http://www.research.ibm.com/ar1/pphec/pphec2004-proceedings.pdf>.
- [24] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.
- [25] C.-W. Tseng. Compiler optimizations for eliminating barrier synchronization. *Proceedings of the fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 144–155, August 1995.