Experiences with an SMP Implementation for X10 based on the Java Concurrency Utilities (Extended Abstract)

Rajkishore Barik
IBM India Research Lab.
rajbarik@in.ibm.com

Vincent Cave
IBM T.J. Watson Res. Ctr.
vcave@us.ibm.com

Christopher Donawa
IBM Toronto Laboratory
donawa@ca.ibm.com

Allan Kielstra
IBM Toronto Laboratory
kielstra@ca.ibm.com

Igor Peshansky
IBM T.J. Watson Res. Ctr.
igorp@us.ibm.com

Vivek Sarkar
IBM T.J. Watson Res. Ctr.
vsarkar@us.ibm.com

1. INTRODUCTION

It is now well established that future computer systems are rapidly moving from uniprocessor to multi-core configurations, and using parallelism instead of frequency scaling as the foundation for increased compute capacity. Unlike previous generations of hardware evolution, this shift towards ubiquitous parallelism will have a major impact on software. Modern OO languages, such as JAVA™ and C#, together with their runtime environments, libraries, frameworks and tools, have made concurrent programming accessible to mainstream application developers, rather than just system programmers. However, the facilities for concurrent programming in these languages have also revealed several drawbacks of parallel programming at the level of unstructured threads with lock-based synchronization.

X10 [1] is an experimental new language currently under development at IBM in collaboration with academic partners. The goal of the X10 project is to build a full language system – including a new language, compiler, optimizer, runtime system, and programming environment — that provides a 10× improvement in development productivity for parallel applications. As part of this agenda, we have developed an SMP implementation for X10 built on the Java Concurrency Utilities, also known as `java.util.concurrent` or the JUC. This paper summarizes our experiences to date with the SMP implementation based on the JCU, and outlines directions for future work.

2. OVERVIEW OF X10

This section provides a brief summary of v0.41 of the X10 programming language, as described in [1]. The goal of X10 is to introduce a core set of new language constructs that address the fundamental requirements for high productivity programming of parallel systems at all scales — multi-core processors, symmetric shared-memory multiprocessors (SMPs), commodity clusters, high end supercomputers like Blue Gene, and even embedded processors like Cell.

With a view to ubiquitous adoption, X10 uses a serial subset of the Java language as its foundation, but replaces the Java language's current support for concurrency by new constructs that are motivated by high-productivity high-performance parallel programming.

Perhaps the simplest introduction to X10 is to examine how three X10 constructs — async, atomic, finish — can be used to write a large class of single-place parallel programs. In a single-place X10 program, all activities execute in the same place, and have uniform read and write access to all shared data, as in multithreaded Java programs where all threads operate on a single shared heap. An important safety result in X10 is that any program written with async, finish, atomic can never deadlock. (This result also holds with the addition of foreach, ateach, and clock constructs.) We now briefly describe the three constructs below.

Async is the X10 construct for creating or forking a new asynchronous activity. The statement, *async* ⟨*stmt*⟩, causes the parent activity to create a new child activity to execute ⟨*stmt*⟩. Execution of the async statement returns immediately i.e., the parent activity can proceed immediately to its next statement.

Consider the following X10 code example of an async construct. The goal of this example is to use two activities to compute in parallel the sums of the odd and even numbers in the range $1 \ldots n$. This is accomplished by having the main program activity use the *async for (int i . . .) . . .* statement to create a child activity to execute the for-i loop and print oddSum, while the main program activity proceeds in parallel to execute the for-j loop and print evenSum.

Proceedings of the 2006 Workshop on Programming Models for Ubiquitous Parallelism, co-located with PACT 2006, September 2006, Seattle, Washington.

```
public static void main(String[] args) {
  final int n = 100;
```

```
  async { // Compute oddSum in child activity
    double oddSum = 0;
    for (int i = 1 ; i <= n ; i += 2 )
      oddSum += i;
    System.out.println("oddSum = " + oddSum);
  }
  // Compute evenSum in parent activity
  double evenSum = 0;
  for (int i = 2 ; i <= n ; i += 2 )
    evenSum += i;
  System.out.println("evenSum = " + evenSum);
} // main()
```

X10 permits the use of async to create multiple nested activities in-line in a single method, unlike Java threads where the body of the thread must be specified out-of-line in a separate Runnable class. Also, note that the child activity uses the value of local variable n from the parent activity, without the programmer having to pass it explicitly as a parameter. X10 provides this sharing of local variables for convenience, but requires that any local variables in the parent activity that are accessed by a child activity must be defined as final (constant) in the parent activity.

The X10 statement, *finish* ⟨*stmt*⟩, causes the parent activity to execute ⟨*stmt*⟩ and then wait till all sub-activities created within ⟨*stmt*⟩ have terminated globally. If async is viewed as a fork construct, then finish can be viewed as a join construct. However, as discussed below, the async-finish model is more general than the fork-join model. X10 distinguishes between *local termination* and *global termination* of a statement. The execution of a statement by an activity is said to terminate locally when the activity has completed all the computation related to that statement. For example, the creation of an asynchronous activity terminates locally when the activity has been created. A statement is said to terminate globally when it has terminated locally and all activities that it may have spawned (if any) have, recursively, terminated globally.

Consider a variant of the previous example in which the main program waits for its child activity to finish so that it can print the result obtained by adding oddSum and EvenSum:

```
public static void main(String[] args) {
  final int n = 100;
  final BoxedDouble oddSumResult = new BoxedDouble();
  double evenSum = 0;
  finish {
    async { // Compute oddSum in child activity
    for (int i = 1 ; i <= n ; i += 2 )
      oddSumResult.val += i;
    }
    // Compute evenSum in parent activity
    for (int i = 2 ; i <= n ; i += 2 )
      evenSum += i;
  } // finish
  System.out.println("Sum = " +
                     (oddSumResult.val + evenSum));
} // main()
```

The finish statement guarantees that the child activity terminates globally before the print statement is executed. Note that the result of the child activity is communicated to the parent in a shared object, oddSumResult, since X10 does not permit a child activity to update a local variable in its parent activity. In this case, the local variable oddSumResult contains a pointer to an object with a val field, thereby enabling oddSumResult.val to be updatable even though oddSumResult is a constant pointer. It is also worth noting that the X10 memory model is weak enough to allow oddSumResult.val to be allocated to a register during the execution of the entire for-i loop.

The atomic construct in X10 is used to coordinate accesses by multiple activities to shared data. The X10 statement, *atomic* ⟨*stmt*⟩, causes ⟨stmt⟩ to be executed atomically i.e., its execution occurs as if in a single step during which ⟨stmt⟩ executes and terminates locally while all other concurrent activities in the same place are suspended. Compared to user-managed locking, the X10 user only needs to specify that a collection of statements should execute atomically and leaves the responsibility of lock management and alternative mechanisms for enforcing atomicity to the language implementation. Commutative operations, such as updates to histogram tables and insertions in a shared data structure, are a natural fit for atomic blocks when performed by multiple activities. An atomic block may include method calls, conditionals, and other forms of sequential control flow. For scalability reasons, blocking operations like finish and force are not permitted in an atomic block. Also, as we will see, an X10 atomic block can only be used to enforce atomicity within a place.

X10 also permits the modifier atomic on method definitions as a shorthand for enclosing the body of the method in an atomic block.

Current programming models use two separate levels of abstraction for shared-memory thread-level parallelism (e.g., Java threads, OpenMP, pthreads) and distributed-memory communication (e.g., Java messaging, RMI, MPI, UPC) resulting in significant complexity when trying to combine the two. In this section, we show how the three core X10 constructs introduced earlier can be extended to multiple places. A place is a collection of resident (non-migrating) mutable data objects and the activities that operate on the data. Every X10 activity runs in a place; the activity may obtain a reference to this place by evaluating the constant *here*.

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

X10 supports a partitioned global address space (PGAS) that is partitioned across places. Each mutable location and each activity is associated with exactly one place, and places do not overlap. A scalar object in X10 is allocated completely at a single place. In contrast, the elements of an

array, may be distributed across multiple places.

The statement, *async* (⟨*place-expr*⟩) ⟨*stmt*⟩, causes the parent activity to create a new child activity to execute ⟨*stmt*⟩ at the place designated by ⟨*place-expr*⟩. The async is local if the destination place is same as the place where the parent is executing, and remote if the destination is different. Local async's are like lightweight threads, as discussed earlier in the single-place scenario. A remote async can be viewed as an active message, since it involves communication of input values as well as remote execution of the computation specified by ⟨*stmt*⟩. The semantics of the X10 finish operator is identical for local and remote async's viz., to ensure global termination of all asyncs created in the scope of the finish.

3. AN SMP IMPLEMENTATION FOR X10

We have constructed an optimized SMP implementation of X10 that extends the reference implementation described in [1]. There were two main reasons for basing this SMP implementation on the Java Concurrency Utilities [4]:

1. Performance: a more efficient implementation can be obtained by creating lightweight X10 activities that can be processed by a fixed number of Java threads in a `ThreadPoolExecutor`, compared to creating a separate Java thread for each X10 activity.

2. Simplicity: the X10 runtime can be simplified since it need not perform low-level synchronizations that are encapsulated in the JUC for implementing thread pools and other services needed by a parallel language runtime system.

The highlights of the implementation are as follows:

- An `X10ThreadPoolExecutor` class is introduced as an extension to the JUC `ThreadPoolExecutor` class. The main functionality in this extension is the ability for the thread pool to increase and decrease the number of threads in the pool as required by the X10 runtime system.

- If an X10 activity performs a blocking operation (*finish*, *force* or *next*), then the Java thread executing the X10 activity will also block and new threads will be added to the pool as needed. If the number of blocked activities (and threads) becomes a significant bottleneck, we will consider a more sophisticated implementation in which the continuation of a blocked activity is serialized thereby enabling its Java thread to do something else till the activity becomes unblocked.

- A `LinkedBlockingQueue` is used to hold X10 activities ready for execution by the `X10ThreadPoolExecutor`. A LIFO policy is used on the queue so as to avoid the large growth in queue size that can occur with a FIFO policy.

- A modified version of the JUC `CountDownLatch` class is used to support a count-down latch in which increments are permitted to the latch's counter. This is necessary to support X10's finish construct, since the number of descendant activities that a finish operation needs to wait on can increase after a finish operation is initiated.

- A modified version of the JUC `CyclicBarrier` class is used to implement X10 clocks. As with the modified `CountDownLatch`, the main modification is to allow the count to be increased dynamically before it trips (and to also change the reset value).

- As in the original X10 reference implementation, a single lock per place is used to support atomic blocks. Current JVMs are very efficient in dealing with uncontended locks. If a single lock per place becomes a source of contention, our recommendation is that the original X10 program be rewritten to use multiple places thereby reducing the amount of contention that occurs within a place.

X10 programs executing on this implementation can emulate multiple places on a single SMP system by using multiple

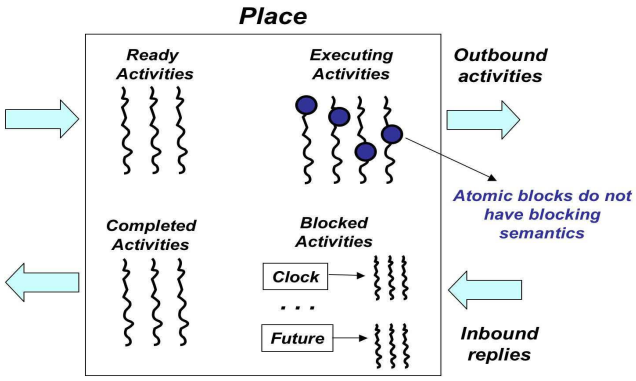

Figure 1: Logical structure of a single X10 place

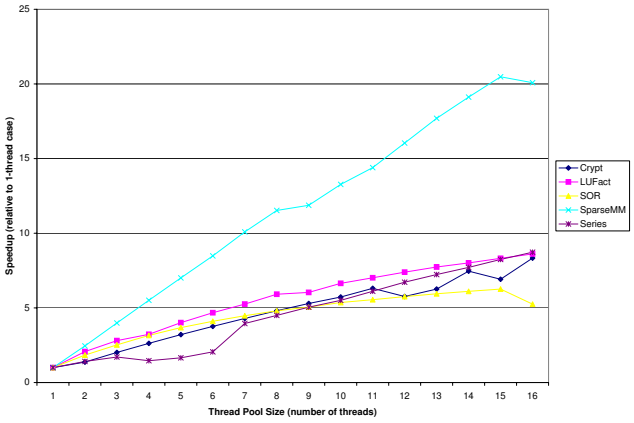

Figure 3: Performance measurements of five Java Grande Forum benchmarks rewritten in X10

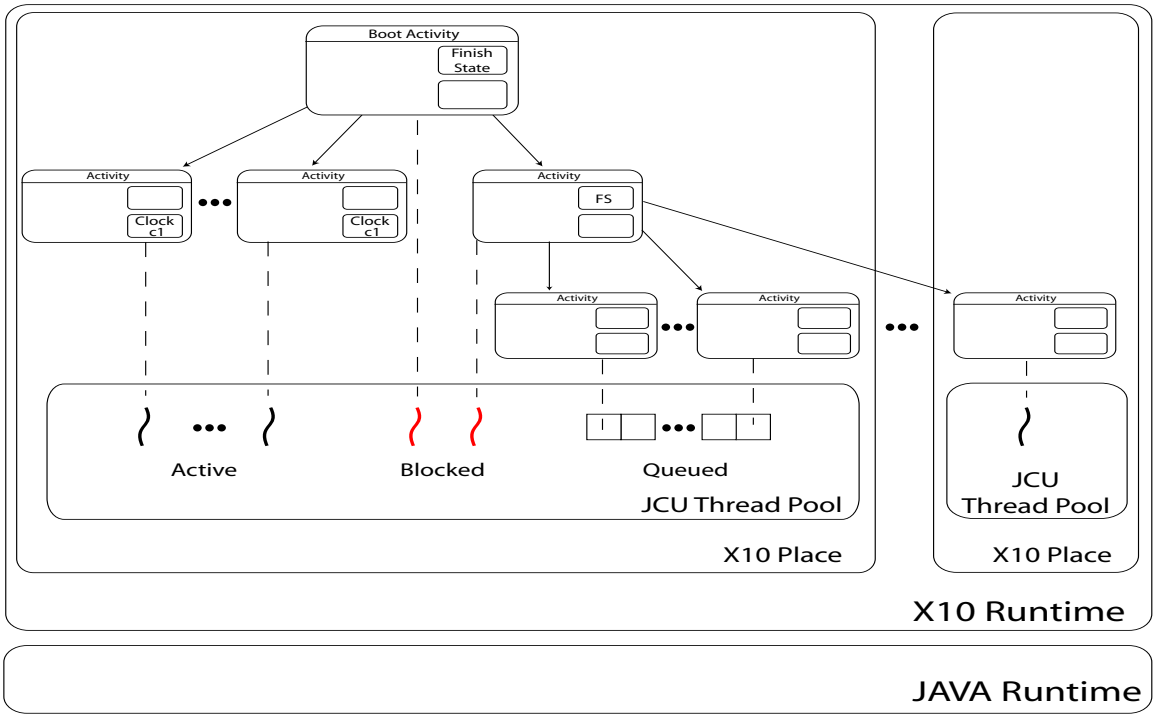

Figure 2: Structure of X10 Runtime System for SMP Implementation

thread pools. The X10 Run Time System (RTS) is written in the Java programming language and thus can take advantage of object oriented language features such as garbage collection, dynamic type checking and polymorphic method dispatch. The design of a place in the SMP implementation corresponds to the *executor* pattern [3]. A place acts as an executor that dispatches individual activities to a pool of JAVA threads, as outlined in Figures 1 and 2. Activities in X10 can be created by async statements, future expressions, foreach and ateach constructs, and array initializers. The thread pool model makes it possible to reuse a thread for multiple X10 activities. However, if an X10 activity blocks, *e.g.*, due to a force operation on a future, or a next operation on a clock, its thread will not be available until the activity resumes and completes execution. In that case, a new JAVA thread will need to be created to serve any outstanding activities. As a result, the total number of threads that can be created in a single place is bounded by the sum of the number of threads initially created in the pool and the maximum number of activities that can be simultaneously blocked.

Atomic blocks are currently implemented using a single mutual exclusion lock per place. While this implementation strategy limits parallelism, it is guaranteed to be deadlock-free since no X10 computation can require the implementation to acquire locks for more than one place. We call an atomic block *analyzable* if the locations and places of all data to be accessed in the atomic section can be computed on entry to the atomic section. Analyzability of atomic sections is not a language requirement, but serves as an important special case for which optimized implementations of atomic sections can be developed using multiple locks instead of a single lock [5].

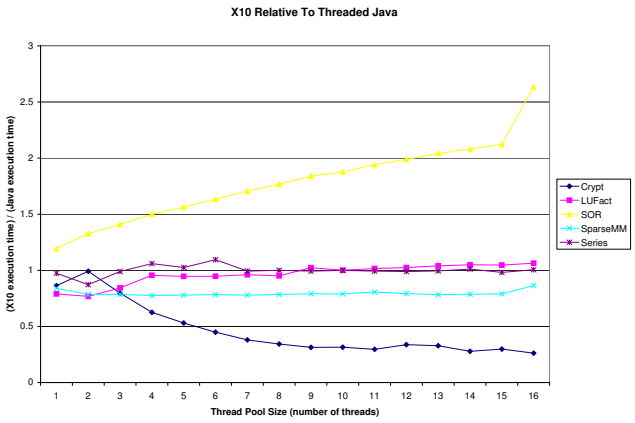

Figure 4: Performance of X10 version relative to original Java version

4. PRELIMINARY EXPERIMENTAL RESULTS

In this section, we present preliminary results comparing the performance of our SMP X10 implementation with a baseline multithreaded Java implementation on an SMP system. The results were obtained on a 16-core 1.9GHz Power5 SMP system with 63.5GB of main memory AIX 5.3 and an IBM J9 VM for Java 5. The Java Grande Forum benchmarks [2] contains five multithreaded benchmarks in Section

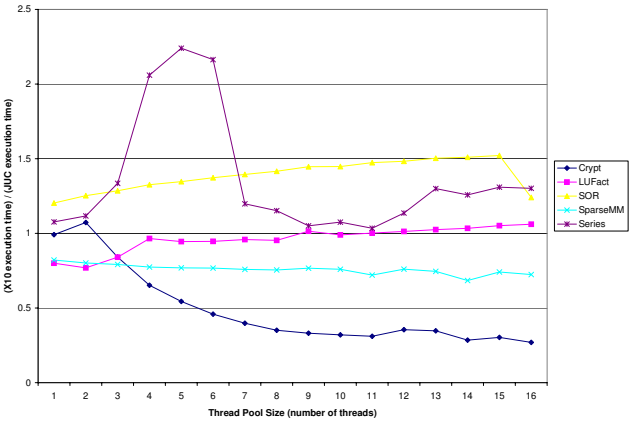

Figure 5: Performance of X10 version relative to Java version written using the JUC

2 — crypt, lufact, series, sor, sparsematmult. We report results for X10, Java, and JUC versions of all five benchmarks with the largest input size (size C). All results were obtained using the `-J-mx2000M -J-ms2000M` options to set the heap size to 2GB, and the `J-Xjit:count=0` option to ensure that each method was JIT-compiled on its first invocation. Since X10's class loading semantics is more relaxed than that of Java, an additional `-PRELOAD_CLASSES=true` option was used in the X10 case to cause all X10 classes referenced by the application to be forcibly loaded before the main program is executed. This allows for better code to be generated when each method is compiled on its first invocation.

Figure 3 summarizes the performance (relative to the 1-thread case) obtained as the number of threads is increased from 1 to 16. Each benchmark is self-validating, so these results represent executions that passed validation. The benchmark configurations used to obtain these results were such that dynamic compilation overhead was included in the measurements for sor and sparsematmult, but not for crypt, lufact, and series. For 16 threads, a speedup in the range of 8× to 9× was obtained for three of the benchmarks (crypt, lufact, series), a lower speedup of 5.2× for sor, and a super-linear speedup of 20.1× for sparsemm. We are studying these results so as to identify opportunities for further optimization in the X10 runtime system.

Figure 4 summarizes the performance of the X10 version relative to the original Java version of the Java Grande Forum benchmarks. For two of the benchmarks (sparsemm, crypt), the X10 version performed consistently better than the Java version. For another two benchmarks (lufact, series), both versions showed roughly comparable performance. Finally, for one benchmark (sor), the Java version outperformed the X10 version, with a performance gap that increased with the number of threads. The sor benchmark uses the clock construct in X10, which was implemented using the JUC CyclicBarrier to obtain these performance results. We are investigating ways in which to further improve the performance of the cyclic barrier.

Finally, Figure 5 summarizes the performance of the X10 version relative to hand-coded JUC versions. There are only two benchmarks (sor, series) for which the JUC version outperformed the X10 version. We are studying these results so as to determine why the X10 version is slower in these two cases.

5. CONCLUSIONS

In this paper, we described our experiences with an SMP implementation of the X10 language based on the Java Concurrency Utilities. The preliminary results are encouraging in showing how our approach offers users the best of both worlds — higher-level abstractions of concurrency, coupled with state-of-the-art performance obtained by leveraging optimized implementations of the JUC. We believe that further performance improvements should be achievable in the future, as we leverage the lessons learned from the performance results reported in this paper.

Acknowledgments

X10 is being developed in the context of the IBM PERCS (Productive Easy-to-use Reliable Computing Systems) project, which is supported in part by DARPA under contract No. NBCH30390004. We are grateful to the following people for their contributions to the X10 implementation described in this paper: Philippe Charles, Kemal Ebcioglu, Robert Fuhrer, Christian Grothoff, Christoph von Praun, and Vijay Saraswat.

6. REFERENCES

[1] Philippe Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Christoph von Praun, Vijay Saraswat, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA 2005 Onward! Track*, 2005.

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

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

[4] Doug Lea. The Concurreny Utilities, 2004. JSR 166, http://www.jcp.org/en/jsr/detail?id=166.

[5] 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.