Local Parallel Iteration in X10

Josh Milthorpe
IBM T.J. Watson Research Center, Yorktown Heights, NY, USA
jjmiltho@us.ibm.com

Abstract
X10 programs have achieved high efficiency on petascale clusters by making significant use of parallelism within places, however, there has been less focus on exploiting local parallelism within a place.

This paper introduces a standard mechanism - foreach - for efficient local parallel iteration in X10, including support for worker-local data. Library code transforms parallel iteration into an efficient pattern of activities for execution by X10’s work-stealing runtime. Parallel reductions and worker-local data help to avoid unnecessary synchronization between worker threads.

The foreach mechanism is compared with leading programming technologies for shared-memory parallelism using kernel codes from high performance scientific applications. Experiments on a typical Intel multicore architecture show that X10 with foreach achieves parallel speedup comparable with OpenMP and TBB for several important patterns of iteration. foreach is composable with X10’s asynchronous partitioned global address space model, and therefore represents a step towards a parallel programming model that can express the full range of parallelism in modern high performance computing systems.

Categories and Subject Descriptors D.1.3 [Concurrent Programming]: parallel programming

Keywords X10, parallel iteration, loop transformations, work stealing

1. Introduction

Data parallelism is the key to scalable parallel programs \[6\]. Although X10 programs have demonstrated high efficiency at petascale, these impressive results have made little use of parallelism within a place, focusing instead on parallelism between places \[8\]. As most scientific codes make heavy use of iteration using for loops, parallel iteration (sometimes called ‘parallel for loop’) is the most obvious approach for exploiting shared-memory parallelism.

The foreach statement was a feature of early versions of the X10 language. The statement was defined as follows:

    1. val complete = new Rail[Boolean](ITERS);
    2. foreach (i in 0..(ITERS-1)) {
      3.  when (complete(i+1)) {
      4.    compute();
      5.  atomic complete(i) = true;
      6. }

Figure 1: Loop with ordering dependence between iterations

Secondly, any intermediate data structures had to be duplicated for each iteration of the loop to avoid data races between iterations. For example, the loop in Figure 2 contains a data race due to sharing of the array temp between threads.

    1. val input:Rail[Double];
    2. val output:Rail[Double];
    3. val temp = new Rail[Double](N);
    4. foreach (i in 0..(ITERS-1)) {
      5.  for (j in 0..(N-1)) {
      6.    temp(j) = computeTemp(i, input(j));
      7.  }
      8.  output(i) = computeOutput(i, temp);

Figure 2: Loop with ordering dependence between iterations

For correctness, temp had to be made private to the body of the loop, requiring that the array be duplicated N times.

For these reasons, the time to compute a parallel loop using foreach was often orders of magnitude greater than an equivalent sequential loop. Furthermore, foreach (p in region) \[8\] was trivially expressible in terms of simpler X10 constructs as finish for (p in region) async \[S\]. The foreach construct was therefore removed in X10 version 2.1. Despite its removal, there remains a strong need for an efficient mechanism for parallel iteration in the X10 language.

The main contributions of this paper are:

- a standard mechanism for local parallel iteration in the X10 language;
• support for worker-local data in the X10 language;
• experimental evaluation of these mechanisms on a multicore architecture typical of HPC compute nodes;
• comparison with leading programming technologies for shared-memory parallelism, specifically, OpenMP and TBB.

2. Related Work

The standard for shared-memory parallelism is OpenMP [1]. All leading C/C++ and Fortran compilers implement the OpenMP API, which provides efficient implementations of common parallel patterns. OpenMP parallel for loops support different scheduling choices including static scheduling for regular workloads, and dynamic and guided scheduling for irregular workloads. In addition, OpenMP supports the creation of explicit tasks, which allow expression of a broader range of parallel patterns. However, the interaction between explicit tasks and the implicit tasks is not fully defined, which makes them difficult to compose [9].

Intel Threading Building Blocks is a C++ template library for task parallelism [2]. In addition to efficient concurrency primitives, memory allocators, and a work-stealing task scheduler, TBB provides implementations of a range of common parallel algorithms and data structures. Parallel iteration and reduction are implemented as parallel_for and parallel_reduce, respectively. The TBB scheduler inspects the dynamic behavior of tasks to perform optimizations for cache locality and task size [7]. Unlike X10, neither OpenMP nor TBB provide support for distributed-memory parallelism.

3. Parallel Iteration With foreach

A new construct for parallel iteration may be defined as follows:

\[
\text{foreach (Index in IterationSpace) Stmt}
\]

The body Stmt is executed for each value of Index, making use of available parallelism. The iteration must be both serializable and parallelizable, in other words, it is correct to execute Stmt for each index in sequence, and it is also correct to execute Stmt in parallel for any subset of indices. The compiler applies one of a set of transformations (see section 4) to generate a set of parallel activities that implements the foreach statement. The transformations available to the compiler may depend on the type of the index expression, and the choice of transformation may be controlled by annotations.

The index expression is evaluated on entry to the foreach statement to yield a set of indices, which must be invariant throughout the iteration. It is desirable that the index set should support recursive bisection. Dense, rectangular index sets (Range and DenseIterationSpace) are trivially bisectable; for other region types, we envisage the introduction of a new interface SplittableRegion defining a split operation, to allow bisection of other region types, similar to TBB’s splitting constructor.

The body of the foreach statement must be expressible as a closure. In addition to the usual restrictions on closures – for example, var variables may not be captured – there are further restrictions specific to foreach. A conditional atomic statement (when) may not be included as it could introduce ordering dependencies between iterations. Unconditional atomic may be included as it cannot create an ordering dependency. These restrictions may be checked dynamically in the same manner as the X10 runtime currently enforces restrictions on atomic statements. Apart from these restrictions, foreach is composable with other X10 constructs including finish, async and at.

Correct execution of foreach assumes no preemption of X10 activities; each activity created by foreach runs to completion on the worker thread which started it. There is an implied finish; all activities created by foreach must terminate before progressing to the next statement following the construct.=

3.1 Reduction Expression

Along with parallel iteration, reduction is a key parallel pattern and a feature of many scientific codes. The foreach statement may be enhanced to provide a parallel reduction expression as follows:

\[
\text{result :U = reduce[T,U] ( reducer :(a:T, b:U)=> U, identity :U) foreach (Index in IterationSpace) { Stmt offer Exp:T; }}
\]

An arbitrary reduction variable of type T is computed using the provided reducer function \( \text{reduce:(a:T, b:T)=> T} \) and an identity value \( \text{identity:T} \) such that \( \text{reduce(identity, x)} = x \). For example, the following code computes a vector dot product of arrays \( x \) and \( y \):

```java
1 val input : Rail [ Double ];
2 val output : Rail [ Double ];
3 foreach (i in 0..( ITERS -1) ) {
4    val temp = new Rail [ Double ](N);
5    for (j in 0..(N-1) ) {
6        temp (j) = computeTemp (i, input (j));
7    }
8    output (i) = computeOutput (i, temp);
9 }
```

However, for data structures of any significant size, repeated allocation is unlikely to be efficient due to increased load on the garbage collector.

An alternative option in Native X10 (using the C++ backend) is stack allocation, as follows:

```java
1 val input : Rail [ Double ];
2 val output : Rail [ Double ];
3 foreach (i in 0..( ITERS -1) ) {
4    @StackAllocate val temp = @StackAllocateUninitialized new Rail [ Double ](N);
5    for (j in 0..(N-1) ) {
6        temp (j) = computeTemp (i, input (j));
7    }
8    output (i) = computeOutput (i, temp);
9 }
```

The annotation @StackAllocate indicates that a variable should be allocated on the stack, rather than the heap. The second annotation, @StackAllocateUninitialized, indicates that the constructor
call should be elided, leaving the storage uninitialized. This avoids the cost of zeroing memory, but should be used with care to ensure values are not read before they are initialized. Stack allocation is a good choice for many applications, however it is limited to variables that will fit on the stack (no large arrays), and is not supported in Managed X10 (using the Java backend).

As an alternative to either duplication or stack allocation, we propose a new class, x10.compiler.WorkerLocal, which provides a lazy-initialized worker-local store. A worker-local store is created with an initializer function; the first time a given worker thread accesses the store, the initializer is called to create its local copy of the data. The definition of foreach can be extended to support worker-local data as follows:

```java
foreach (Index in IterationSpace)
local {
  val 11 = Initializer1;
  val 12 = Initializer2;
} Stmt
```

The value initializers in the local block may capture the environment of the parallel iteration, but may not reference any symbol defined inside the body. The body of the iteration may refer to any of the variables defined within the local block. Because the body may not include blocking statements each execution of the body must run to completion on the worker thread on which it began, therefore it has exclusive access to its worker local data for the entire duration.

The x10.compiler.WorkerLocal class is very similar in design to TBB’s enumerable_thread_specific type, which is also a lazy-initialized thread-local store.

4. Implementation

The foreach, reduce and local keywords can be supported in X10 by extending the language syntax, however, we have not actually implemented these changes in the compiler. Instead, we have created two new classes, x10.compiler.Foreach and x10.compiler.WorkerLocal, which are intended as targets for future versions of the language; in the interim, these classes can be used directly from user code.

Given the definition of the foreach statement in Section 3, a variety of code transformations are possible. The X10 compiler should provide an efficient default transformation (for example, recursive bisection), combined with annotations to allow the user to choose different transformations for particular applications.

To illustrate some possible transformations, we consider the following implementation of a simple “DAXPY” using a foreach statement over a LongRange as follows:

```java
1 foreach (i in lo..hi) {
2   x(i) = alpha * x(i) + y(i);
3 }
```

As a first step, the body of the foreach is extracted into a closure that executes sequentially over a range of indices as parameters:

```java
1 val body = (min_i :Long , max_i :Long) => {
2   for (i in min_i..max_i) {
3     x(i) = alpha * x(i) + y(i);
4   }
5 }
```

The body closure is then used to construct a parallel iteration using one of the code transformations in the following subsections.

4.1 Basic

The basic transformation can be applied to any iterable index set, to create a separate activity for each index:

```java
1 finish for (i in lo..hi) async body(i, i);
```

This is equivalent to the original definition of foreach.

4.2 Block Decomposition

A block decomposition can be applied to any countable index set, and divides the indices into contiguous blocks of approximately equal size. By default, Runtime.NTHREADS blocks are created, one for each worker thread. Each block is executed as a separate async, except for the first block which is executed synchronously by the worker thread that started the loop.

```java
1 val numElem = hi - lo + 1;
2 val blockSize = numElem / Runtime.NTHREADS;
3 val leftOver = numElem % Runtime.NTHREADS;
4 finish {
5   for (var t:Long=Runtime.NTHREADS-1; t>0; --t) {
6     val tLo = lo + t * leftOver + (t < leftOver) ? (blockSize+1) : blockSize + leftOver;
7     val tHi = tLo + ((t < leftOver) ? (blockSize+1) : blockSize);
8     async body(tLo..tHi);
9   }
10   body(0, blockSize + leftOver ? 1 : 0);
11 }
```

4.3 Recursive Bisection

A recursive bisection transformation can be applied to any splitable index set. In this approach, the index set is divided into two approximately equal pieces, with each piece constituting an activity. Bisection recurs until a certain minimum grain size is reached.

```java
1 static def doBisect1D(lo:Long , hi:Long ,
2   grainSize:Long ,
3   body:(min :Long , max : Long) => void) {
4   if ((hi -lo) > grainSize) {
5     async doBisect1D(lo, (lo+hi)/2L , hi ,
6       grainSize , body);
7     doBisect1D((lo+hi)/2L , grainSize ,
8       body);
9   } else {
10     body(lo, hi-1);
11   }
12 }
```

With the recursive bisection transformation, if a worker thread’s deque contains any activities, then the activity at the bottom of the deque will represent at least half of the index set held by that worker. Thus idle workers tend to steal large contiguous chunks of the index set, preserving locality.

5. Evaluation

We identified a number of application kernels representing common patterns in high-performance scientific applications. The use of kernels instead of full applications allows the effects of data-parallel transformations to be studied in a simplified context free from scheduling effects due to other parts of the applications. Using these kernels, we compared the different compiler transformations for foreach the different storage options for intermediate data structures. Finally, we compare the performance of the X10 versions of these kernels with versions written in C++ with OpenMP and/or TBB.
5.1 Application Kernels

5.1.1 DAXPY

The DAXPY kernel updates each element of a vector as $x_i = \alpha x_i + y_i$.

```java
1 foreach (i in 0..(N-1)) {
2   x(i) = alpha * x(i) + y(i);
3 }
```

5.1.2 Dense Matrix Multiply

The MatMul kernel is an inner-product formulation of dense matrix multiplication which updates each element $c(i+j*M) = \sum_{k=1}^{K} a(i+k*M) * b(k+j*K)$.

```java
4 foreach (i in 0..(N-1)) {
5   for (j in 0..(K-1) ) {
6     temp += a(i+k*M) * b(k+j*K);
7   }
8   c(i+j*M) = temp;
9 }
```

5.1.3 Sparse Matrix-Vector Multiply

The SpMV kernel is taken from the X10 Global Matrix library \(^3\), available for download at http://x10-lang.org. It performs sparse matrix-vector multiplication and forms the basis of many GML algorithms.

```java
6 foreach (j in 0..(M-1)) {
7   temp = Double 0.0;
8   for (k in 0..(K-1) ) {
9     temp += a(i+k*M) * b(k+j*K);
10    }
11   c(i+j*M) = temp;
12 }
```

5.1.4 Jacobi Iteration

The Jacobi kernel combines a stencil update of interior elements of a two-dimensional region with a reduction of an error residual. The Jacobi benchmark is available in the X10 applications repository \(^4\).

```java
1 error = reduce [Double](
2   (a:Double , b:Double )=>{ return a+b;} , 0.0)
3 foreach (i in 0..(N-1) ) {
4   x(i) = alpha * x(i) + y(i);
5 }
```

5.1.5 LULESH Hourglass Force

LULESH2.0 \(^5\) is a mini-app for hydrodynamics on an unstructured mesh. It models an expanding shock wave in a single material originating from a point blast. The simulation iterates over a series of time steps up to a chosen end time. At each time step, node-centered kinematic variables and element-centered thermodynamic variables are advanced to a new state. The new values for each node/element depend on the values for neighboring nodes and elements at the previous time step. A model implementation is provided using C++, OpenMP and MPI; we ported this implementation to X10.

The LULESH application contains a number of important computational kernels which update different node and element variables. The kernel which computes the Flanagan-Belytschko anti-hourglass force for a single grid element accounts for the largest portion – around 20% – of the application runtime. It requires a number of intermediate data structures which are all small 1D or 2D arrays. The LULESH Hourglass Force kernel is available in the X10 applications repository \(^4\).

```java
1 foreach (i in 0..(numElem-1))
2 local {
3   val hourgam = new Array_2[Double](
4     hourgamStore , 8, 4);
5   val xdl = new Rail[Double](8);
6   ... }
7 }
8 foreach (i in 0..3) {
9   val i3 = 8*i2;
10  val volinv = 1.0 / determ(i2);
11  for (i1 in 0..3) {
12    ... val setHourgam = (idx:Long) => {
13      hourgam(idx,i1) = gamma(i1,idx)
14      + volinv * ((dvdx(i1,idx) * hourmodx
15        + dvdv(i1, idx) * hourmody)
16        + dvdz(i1, idx) * hourmodz);
17    }; setHourgam(0);
18    setHourgam(1);
19    ... setHourgam(7);
20 }
21 calcElemFBHourglassForce(xd1, yd1, zd1,
22     hourgam,
23     coefficient, hgfx, hgyf, hgI2); ...
24 }
```

5.2 Experimental Setup

The kernels described in \(^5\) were executed on an Intel Xeon E5-4657L v2 @ 2.4 GHz. The machine has four sockets, each with 12 cores supporting 2-way SMT for a total of 96 logical cores. X10 version 2.5.2 was modified to implement the x10.compiler.foreach and x10.compiler.WorkerLocal classes as described in Section \(^4\). GCC version 4.8.2 was used for post-compilation of the Native X10 programs, as well for the C++ versions of the kernels. Intel TBB version 4.3 update 4 was used for the TBB versions of the kernels. Each kernel was run for a large number of iterations (100-5000), enough to generate a minimum total runtime of several seconds), recording the mean time over a total of 30 test runs.

5.3 Comparison of Compiler Transformations

We first compare the efficiency of parallel iteration using the compiler transformations described in Section \(^4\). Each kernel was compiled using the basic, block and recursive bisection transformations. Figure \(^3\) shows the scaling with number of threads for each kernel using the different transformations. Parallel speedup (single-core

---

1. https://sourceforge.net/projects/anuchem/
3. https://sourceforge.net/projects/anuchem/
```plaintext
The results for the basic transformation illustrate why the original definition of for each in X10 was infeasible: for all kernels, the basic transformation fails to achieve parallel speedup for any number of threads.

The block and bisect transformations are more promising: all codes show some speedup up to at least 32 threads. The results fail to completely separate the two transformations; for each kernel, each transformation exhibits a greater parallel speedup for some portion of the tested thread range (2–96). The block transformation achieves the greatest maximum speedup for DAXPY, Jacobi and LULESH, whereas the 1D and 2D bisection transformations achieve the greatest speedup for SpMV and MatMul respectively.

The fact that neither transformation is obviously superior indicates the importance of allowing the programmer to choose between them on a per-application or even per-loop basis.

We next compare the three different approaches to storage of local data that were discussed in Section 5.2. Figure 4 shows the scaling with number of threads for the LULESH hourglass force kernel using per-iteration heap allocation, stack allocation, and x10.compiler.WorkerLocal.

The greatest total speedup for LULESH (× 22) is achieved with 56 threads using stack allocation, however, there is not a significant performance difference between the three approaches over the entire range. The intermediate data structures are not large (no array is larger than 32 8-byte elements), so it may be that the cost of allocating multiple copies is insignificant compared to other factors. Other application examples are needed to more thoroughly evaluate approaches to storing local data.

5.4 Comparison of Programming Models

We implemented the kernels listing in §5.1 using OpenMP and TBB. OpenMP codes used schedule(block) for parallel for loops, and TBB codes used the default auto_partitioner.

Figure 5 shows the scaling with number of threads for each kernel using the different programming models. Parallel speedup is normalized to the best single-thread time for any of the three models. Each programming model achieves the greatest maximum speedup for one of the kernels. For the DAXPY kernel, OpenMP significantly outperforms both X10 and TBB. TBB was not tested for the Jacobi or LULESH kernels.

None of the kernel codes presented here achieve anything near perfect parallel speedup across the full range of threads tested. The maximum speedup achievable for a code depends on many factors in addition to the programming model, including: the level of parallelism available in the algorithm; the balance between floating point, memory and other operations; and cache locality. We hope
```

to further explore these issues with regard to particular kernels, to determine whether enhancements to the X10 scheduler – for example, support for affinity-based scheduling [7] – are necessary to achieve greater parallel performance.

6. Conclusion

This paper presented the foreach construct, a new standard mechanism for local parallel iteration in the X10 language. It was shown that this mechanism achieves parallel speedup comparable with OpenMP and TBB for a range of kernels typical of high performance scientific codes. None of the compiler transformations are novel, nor is the provision of a mechanism for worker-local data. However, the mechanisms presented in this paper are composable with the X10 APGAS model, which exposes data locality in the form of places and supports asynchronous remote activities. The mechanisms presented here therefore represent a further step towards a programming model that can express the full range of parallelism in modern high performance computing systems.

Acknowledgments

Thanks to Olivier Tardieu and David Grove for their advice on many important details of the X10 runtime and compiler. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under Award Number DE-SC0008923.

References