Phaser Accumulators: a New Reduction Construct for Dynamic Parallelism

J. Shirako D. M. Peixotto V. Sarkar W. N. Scherer III
Department of Computer Science, Rice University
{shirako,dmp,vsarkar,scherer}@rice.edu

Abstract

A reduction is a computation in which a common operation, such as a sum, is to be performed across multiple pieces of data, each supplied by a separate task. We introduce phaser accumulators*, a new reduction construct that meshes seamlessly with phasers to support dynamic parallelism in a phased (iterative) setting. By separating reduction computations into the parts of sending data, performing the computation itself, and retrieving the result, we enable overlap of communication and computation in a manner analogous to that of split-phase barriers. Additionally, this separation enables exploration of implementation strategies that differ as to when the reduction itself is performed: eagerly when the data is supplied, or lazily when a synchronization point is reached.*

We implement accumulators as extensions to phasers in the Habanero dialect of the X10 programming language. Performance evaluations of the EPCC Syncbench, Spectral-norm, and CG benchmarks on AMD Opteron, Intel Xeon, and Sun UltraSPARC T2 multicore SMPs show superior performance and scalability over OpenMP reductions (on two platforms) and X10 code (on three platforms) written with atomic blocks, with improvements of up to 2.5× on the Opteron and 14.9× on the UltraSPARC T2 relative to OpenMP and 16.5× on the Opteron, 26.3× on the Xeon and 94.8× on the UltraSPARC T2 relative to X10 atomic blocks. To the best of our knowledge, no prior reduction construct supports the dynamic parallelism and asynchronous capabilities of phaser accumulators.

1 Introduction

With multicore processors, the computer industry is entering a new era of mainstream parallel processing in which the need for improved productivity and performance in parallel programming has taken on a new urgency. Many experts (*e.g.,* [6]) believe that software must be redesigned for multicore parallelism as was done in past decades for vector and cluster parallelism, and that *dynamic parallelism* is key to the redesign so as to overcome the limitations of past Bulk Synchronous Parallel (BSP) and Single Program Multiple Data (SPMD) paradigms. We recently introduced *phasers* [15], a new coordination construct that unifies collective and point-to-point synchronization in the presence of dynamic parallelism. In this paper we present *phaser accumulators*, a new construct for dynamic parallel reductions and describe its efficient implementation.

A parallel reduction is a well known operation that computes a single value from a collection of values in parallel. Typically, the programmer specifies an operator that is commutative and associative, along with the values to be reduced using the operator. Parallel reductions exist in a variety of parallel programming paradigms including OpenMP [11], MPI [16], and Microsoft's PLINQ [12]. The accumulators introduced in this paper are unique because they support reductions for *dynamic parallelism*, where the number of activities participating in the reduction can grow and shrink dynamically. We believe that the support for dynamic parallelism eases the burden on programmers because it helps address future multicore software requirements for self-adaptation and dynamic data driven execution.

We present experimental results for an implementation of phaser accumulators as in the Habanero [8] dialect of the X10 programming language. Performance evaluations of the EPCC Syncbench, Spectral-norm, and CG benchmarks on AMD Opteron, Intel Xeon, and Sun UltraSPARC T2 multicore SMPs show superior performance and scalability over OpenMP reductions (on two platforms) and X10 code (on three platforms) written with atomic blocks, with improvements of up to 2.5× on the Opteron and 14.9× on the UltraSPARC T2 relative to OpenMP and 16.5× on the Opteron, 26.3× on the Xeon and 94.8× on the UltraSPARC T2 relative to X10 atomic blocks. The performance of phaser accumulators was comparable to that of OpenMP reductions on the Intel Xeon platform.

The rest of the paper is organized as follows. Phasers are summarized in Section 2. We introduce our program-

ming model for accumulators and describe its API in section 3. Efficient implementation is a primary goal, so we study a range of implementation techniques for accumulators in Section 4. Finally, we present some empirical results for the efficiency and scalability of an X10-based implementation of accumulators on three parallel multi-core machines. Our results in Section 5 show that accumulators achieve superior performance and scalability over OpenMP reductions on two of the three platforms and over X10 code written with atomic blocks on all three platforms. We conclude that phaser accumulators are a valuable construct for supporting reductions with dynamic parallelism on current and future multi-core hardware.

2 Background

2.1 Phasers

In this section, we summarize the *phaser* construct introduced in [15]. Phasers integrate collective and point-to-point synchronization by giving each activity (task)[1] the option of registering with a phaser in *signal-only* or *wait-only* mode for producer-consumer synchronization or *signal-wait* mode for barrier synchronization. In addition, a *next* statement for phasers can optionally include a *single* statement which is guaranteed to be executed exactly once during a phase transition [18]. These properties, along with the generality of *dynamic parallelism* and the *phase-ordering* and *deadlock-freedom* safety properties, distinguish phasers from synchronization constructs in past work including barriers [7, 11], counting semaphores [13], and X10's clocks [4].

Before describing phasers, we briefly recapitulate the *async* and *finish* constructs for activity creation and termination that were introduced in v0.41 of the X10 programming language [4]. Though phasers as described in this paper may seem X10-specific, they are a general unification of point-to-point and collective synchronizations that can be added to other programming models such as OpenMP, Intel's Thread Building Blocks, Microsoft's Task Parallel Library, and Java Concurrency Utilities.

Async is the X10 construct for creating a new asynchronous activity. The statement, *async* $\langle stmt \rangle$, causes the parent activity to create a new child activity to execute $\langle stmt \rangle$. Execution of the async statement returns immediately i.e., the parent activity can proceed immediately to its next statement. The X10 statement, *finish* $\langle stmt \rangle$, causes the parent activity to execute $\langle stmt \rangle$ and then wait till all sub-activities created within $\langle stmt \rangle$ have terminated (including transitively spawned activities). Each dynamic instance of a *finish* statement can be viewed as being bracketed by matching instances of *start-finish* and *end-finish* instructions. Operationally, each instruction executed in an X10 activity has a unique *Immediately Enclosing Finish* (IEF) dynamic statement instance. In the X10 computation DAG introduced in [1], a *dependence edge* is introduced from the last instruction of an activity to the *end-finish* node corresponding to the activity's IEF.

[1]The terms "activity" and "task" are used interchangeably in this paper.

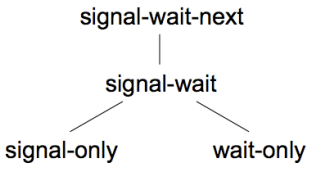

Figure 1. Capability lattice for phasers

A *phaser* is a synchronization object that supports the four operations listed below. At any point in time, an activity can be registered in one of four modes with respect to a phaser: *signal-wait-next*, *signal-wait*, *signal-only*, or *wait-only*. The mode defines the capabilities of the activity with respect to the phaser. There is a natural lattice ordering of the capabilities as shown in Figure 1. The phaser operations that can be performed by an activity, A_i, are defined as follows:

1. **new:** When A_i performs a `new phaser(MODE)` operation, it results in the creation of a new phaser, ph, such that A_i is registered with ph according to `MODE`. If `MODE` is omitted, the default mode assumed is *signal-wait*. Phaser creation also initializes the phaser to its first phase (phase 0). At this point, A_i is the only activity registered on ph to start with.

2. **phased async:**
   `async phased (` $ph_1\langle mode_1\rangle, \ldots$ `)` A_j
   When activity A_i creates an async child activity A_j, it has the option of registering A_j with any subset of phaser capabilities possessed by A_i. This subset is enumerated in the list contained in the `phased` clause. We also permit the "`async phased` A_j" syntax to indicate that A_i is transmitting all its capabilities on all phasers that it is registered with to A_j.

3. **drop:** A_i drops its registration on all phasers when it terminates. In addition, when A_i executes an end-finish instruction for finish statement F, it completely de-registers from each phaser ph for which F is the IEF for ph's creation.

4. **next:** The `next` operation has the effect of advancing each phaser on which A_i is registered to its next phase, thereby synchronizing all activities registered on the same phaser. As described in [15], the semantics of

```
finish {
  delta.f = epsilon+1; iters.i = 0;
  phaser ph = new phaser();
  for ( jj = 1 ; jj <= n ; jj++ ) {
    final int j = jj;
    async phased { // will be registered with ph
      while ( delta.f > epsilon ) {
        newA[j] = (oldA[j-1]+oldA[j+1])/2.0f ;
        diff[j] = Math.abs(newA[j]-oldA[j]);
        next; // barrier
        // Compute reductions in activity jj=1
        if (j == 1)
          { delta.f = diff.sum(); iters.i++; }
        next; // barrier
        temp = newA; newA = oldA; oldA = temp;
      } // while
    } // async
  } // for
} // finish
System.out.println("Iterations: " + iters.i);
```

Figure 2. One-Dimensional Iterative Averaging using Phasers

`next` depends on the registration mode that A_i has with a specific phaser, ph.

Figure 2 shows a version of the pedagogical One-Dimensional Iterative Averaging program from [5] rewritten to use phasers. Each iteration of the `jj` loop creates a new async registered on phaser `ph`. The *dynamic parallelism* supported by phasers is reflected in the fact that the set of activities registered on phaser `ph` can vary dynamically during program execution. Initially, the parent activity is the only one registered on `ph`. Each successive iteration of the `for-`jj loop creates a new `async` activity that is registered on `ph`. Though not shown in this example, it is also possible for some activities to drop their registration on `ph` while other activities continue to work with `ph` *e.g.,* if the termination condition for the `while` loop is changed to a local condition (such as "`delta.f > epsilon[j]`") which may have different values for different iterations of the `for-`jj loop.

Note the use of two `next` statements as barriers to bracket the reduction computations for `diff.sum()` and `iters.i++` performed by activity `jj=1`. As discussed in [15], this idiom can be replaced by the *single* statement available with phasers to reduce the barrier overhead from two barriers to one; however, the reduction computations would still be performed by a single activity.

2.2 Reductions

In the previous example in Figure 2, the `diff.sum()` computation was performed by a single activity, thereby making it a sequential bottleneck for the program. It is well known from past work that reductions and scans on commutative and associative operations can be performed in parallel [3], and that programming models such as MPI and OpenMP have introduced special primitives for such cases. However, prior parallel programming models have restricted reductions to a fixed set of threads or data-parallel operands, and have not supported reductions in conjunction with dynamic parallelism. For example, it is unclear from past work how to extend the dynamic parallelism in Figure 2 with a parallel reduction computation instead of a sequential computation in the `jj=1` activity. This provides the motivation for our work on phaser accumulators.

3 Accumulators

A *phaser accumulator* is a new construct that integrates with phasers to support reductions for dynamic parallelism in a phased (iterative) setting. By separating reduction operations into the parts of sending data, performing the computation itself, retrieving the result, and synchronizing among activities, we enable asynchronous overlap of communication, computation and synchronization in a manner analogous to that of fuzzy [7] or split-phase [9] barriers. This separation provides benefits to the programmer and the implementer. The programmer can use a value-oriented interface for reductions in which they need not be concerned with race conditions on shared locations involved in reductions. The implementer has the freedom to explore different implementation strategies for different hardware targets, such as the *eager* and *lazy* strategies discussed in Section 4.

3.1 Accumulator Operations

Accumulators support two logical operations with respect to a phaser — `accum.send(value)` to send a value for accumulation in the current phase, and `accum.result()` to receive the accumulated value from the previous phase. Thus, the barrier synchronization provided by phasers provides an ideal hook for reductions. If an activity performs multiple `send()` operations on the same accumulator in the same phase, they are treated as separate contributions to the reduction. In contrast to traditional reduction implementations, the `send()` and `result()` operations are separate, so a task not directly involved in — or no longer involved in — performing the reduction calculation is free to perform other work while awaiting the result of this computation. Further, since the `result()` operations returns the accumulated value from the previous phase, it does not encounter any race conditions with `send()` operations in the current phase.

The `send()` Accumulator operation presents the Accumulator with a task's datum. For an eager-compute Accumulator, the implementation is then responsible for remitting tasks' data to the center of reduction. For a lazy-

```
// Construct a new Accumulator
Accumulator(Phaser ph, Accumulator.Operation op,
            Class dataType);

// Remit a datum for a reduction operation
void Accumulator.send(Number datum);

// Retrieve the result of the reduction
Number Accumulator.result();
```

Figure 3. Accumulator API

compute Accumulator, the implementation needs merely to publish each datum to a known location so that it can be accessed later as needed for the reduction computation.

Figure 3 contains the programming interface for accumulators. In the constructor:

- `ph` is the host phaser with which the accumulator will be associated.
- `op` is the reduction operation that the Accumulator will perform. Currently supported operators include sum, product, min, max, bitwise-or, bitwise-and, and bitwise-exor. Support for logical-or, logical-and and user-defined reductions will be added in the future.
- `dataType` is the numerical type of the data upon which the Accumulator operates. The choices are currently limited to int.class, float.class, and double.class. The runtime system throws an exception if a `send()` operation is attempted with a type that does not match the type of the accumulator.

In our current implementation of accumulators, only tasks registered to their respective phasers in `signal-wait` or `signal-wait-next` modes are supported with accumulators. This requirement allows us to avoid buffering of sends for future phases and of results for previous phases. For the remainder of this paper, we will restrict our attention to only tasks with `signal-wait` registrations. In particular, support for tasks registered as `signal-only` or `wait-only` is a topic for future research. The restriction to `signal-wait` or `signal-wait-next` modes makes the accumulator semantics directly applicable to X10 clocks [4] as well.

3.2 Example

Figure 4 is the example from Section 2, reworked to use accumulators. Replacing the explicit summation code with a reduction call and using the next-with-single phaser variant eliminates an entire barrier from the code and gives the implementation flexibility on how best to support the `send` and `result` operations. Note that `ac.result()` is only computed once per loop iteration since it is included in a

```
finish {
  delta.f = epsilon+1; iters.i = 0;
  phaser ph = new phaser();
  accumulator ac = new accumulator(ph, ph.SUM,
                                   int.class);
  for ( jj = 1 ; jj <= n ; jj++ ) {
    final int j = jj;
    async phased { // will be registered with ph
      while ( delta.f > epsilon ) {
        newA[j] = (oldA[j-1]+oldA[j+1])/2.0f;
        diff[j] = Math.abs(newA[j]-oldA[j]);
        ac.send(diff[j]);
        // Local work can be overlapped with
        // accumulator operations
        next {
          // barrier w/ single statement
          delta.f = ac.result(); iters.i++;
        }
        temp = newA; newA = oldA; oldA = temp;
      } // while
    } // async
  } // for
} // finish
System.out.println("Iterations: " + iters.i);
```

Figure 4. One-Dimensional Iterative Averaging using Accumulators

single statement [15]. Also, if available, local work can be overlapped with the `ac.send()` accumulator operations.

4 Implementation Approaches

Herein we discuss the implementation of accumulators. As the requirements of Section 3 specifies only semantics, we have considerable latitude to explore implementations.

We explore three implementations of accumulators in this section: *eager*, *dynamic-lazy*, and *fixed-lazy*. The primary difference among the implementations is how and when the actual accumulation is performed. The eager method uses an atomic update operation to update the accumulated value as soon as an activity sends a value to the accumulator. The dynamic-lazy and fixed-lazy methods wait until all activities have sent their values to the accumulator before performing the reduction. The actual reduction operation is performed when all activities have reached the barrier. Figure 5 shows the major data structure layout used in each of the three implementations.

4.1 Eager Implementation

The eager implementation uses atomic operations to incrementally perform the reduction as soon as a value is sent by an activity. The accumulator contains an atomic variable of the type used for the reduction. This variable is updated at each `send` operation. No data structures are added to

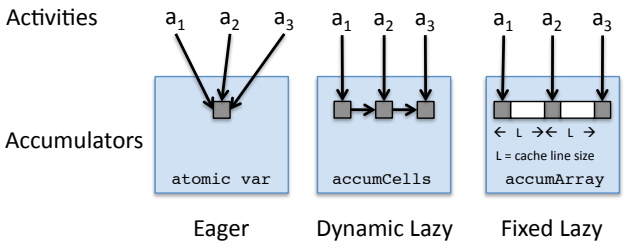

Figure 5. Storage data structures used by the three implementations

```
public void sendEager(final int val) {
  if (ope == Operator.SUM) {
    int n = 1;
    while (true) {
      final int cur = atomI.get();
      final int neo = cur + val;
      if (atomI.compareAndSet(cur, neo)) break;
      else delay(n++); //optional delay
    }
  }
  else . . .
}
```

Figure 6. Eager implementation of the `send` method

the activities for implementing the eager case. The phaser is extended with a list of accumulators associated with the phaser.

The accumulation depends on two parts: an accumulator `send` operation and a phaser `next` operation. When an activity calls `send` on an accumulator, the accumulator uses an atomic update to update the variable containing the reduction value. When a barrier point is reached by the `next` operation, the phaser iterates through each accumulator associated with the phaser. The value of the accumulation is read from the atomic variable and stored in the result field of the accumulator. The atomic variable is reset to the identity element for the next round of accumulation. No actual reduction is performed at the `next` operation. The final result is stored separately so that accesses to it may be non-atomic.

Figure 6 shows the implementation of the `send` method in the eager case for a sum reduction. The variable `atomI` is an atomic integer that holds the accumulated value for the accumulator. The method reads the current value of the accumulation and continually attempts to atomically update the value with the new sum until the update is successful.

The eager implementation is very aggressive in attempting to compute the accumulated value. Because it uses few data structures, it is readily added to an existing runtime. We would expect it to perform well in cases where there are non-overlapping calls to `send` on the accumulator. However, because all activities are writing to the same memory location (hot spot), this approach has an intrinsic problem with scaling to a large number of activities.

In an attempt to improve the scalability of the eager method, we explored the idea of adding a delay to the inner loop if the atomic update in the `sendEager` method fails [2]. By adding a delay, we reduce the contention and bus traffic for the update on the atomic variable. The delay function is a tunable parameter in the implementation, and is a function of n, the number of iterations executed in the while loop in Figure 6. We experimented with a variety of delay functions including: random, linear in n, exponential in n, and constant.

4.2 Dynamic-Lazy Implementation

The dynamic-lazy implementation delays the accumulation of values until the barrier point at the phaser's `next` operation. Each participant in the reduction gets a separate accumulation cell that holds its contribution to the accumulation. At the `next` operation, the values in all of the accumulation cells are accessed and reduced. This implementation is dynamic because it supports adding activities to the accumulation after the accumulator has been allocated. In contrast, the fixed-lazy implementation described in Section 4.3 allows only a predetermined number of participants.

The data structures for the dynamic-lazy implementation are more involved than for the eager implementation. The core data structure used in the implementation is an `accumCell` that holds the accumulation values for each activity participating in the accumulation (Figure 5). These cells are allocated in the accumulator

when an activity registers with the phaser associated with the accumulator. The cells are connected in two ways. First, they are chained together as a linked list so that the accumulator can walk the list to perform the reduction operation. Second, each activity maintains a link to its `accumCell` for a specific accumulator (Figure 5). As with the eager implementation, the phaser maintains a list of all accumulators associated with that phaser.

When an activity calls `send` on an accumulator, it finds the `accumCell` associated with that accumulator and stores the value in the cell. There is no synchronization needed in this operation since each activity has a unique cell per accumulator. Figure 7 shows the pseudocode for this operation.

The actual accumulation operation is performed at the synchronization point that occurs at the phaser's `next` operation. Multiple send operations by the same activity in the same phase would already have been accumulated in the activity's local `accumCell`. When the `next` is reached, the

```
public void sendDynamicLazy(final int val) {
  Activity activity = Runtime.getCurrentActivity();
  //get cell for this accumulator
  AccumCell cell = activity.cellMap[this];
  if (SUM == ope)
    cell.intValue += val;
}
```

Figure 7. Dynamic-Lazy implementation of the `send` method

master activity cycles through each accumulator associated with the phaser. For each accumulator, the `accumCell` list is traversed to perform the accumulation operation. This result is stored in the accumulator so that it can be accessed by activities after the `next` completes.

The dynamic-lazy implementation delays the accumulation operation until all of the values are ready. It does not require any separate synchronization beyond that required for the phaser's `next` operation. Though the dynamic-lazy implementation does not have scalability issues with respect to atomic operations, the accumulation operation is performed sequentially which could become a major scalability bottleneck as the number of activities participating in the accumulation increases. In future work, we plan to use a "tournament barrier" approach (as in [10]) to further improve the scalability of the dynamic-lazy implementation.

4.3 Fixed-Lazy Implementation

The fixed-lazy implementation is similar to the dynamic-lazy except that we accumulate results in a fixed size array instead of a linked list. The array is padded so that each activity accesses an element on a different cache line to avoid false sharing effects. This implementation sacrifices some flexibility by fixing the number of activities participating in the accumulation at the time the accumulator is created. If a new activity later joins the accumulator, the fixed-lazy implementation will need to be transformed to a dynamic-lazy implementation. The decrease in flexibility is made in an attempt to improve the performance of the fixed-lazy implementation. Using an array decreases the number of heap allocations since we can allocate a fixed size array rather than individual list cells. Also, when multiple accumulators are associated with a phaser, a potential future improvement is to store multiple accumulator elements for the same activity in consecutive positions in the array for improved locality.

Table 1 summarizes the actions taken in the implementation of activities, phasers, and accumulators for the three implementations described in this section. For each implementation variant we describe what happens during allocation of an accumulator, registration with a phaser (item #2 in Section 2.1), send of a value to an accumulator, and at the `next` operation of a phaser (item #4 in Section 2.1).

5 Experimental Results

In this section we present experimental results for our phaser accumulators developed in the Habanero multicore software research project at Rice University [8] . We present results on 8-core, 16-core, and 64-thread machines for the three benchmarks shown in Table 2 with the following implementation variants (the same source code was used for variants 3, 4, 5, and 6):

1. **Sequential X10** is the baseline for all speedup results except Syncbench (for which results are reported in microseconds rather than as relative speedups). For the NAS parallel benchmark CG, we use the "serial" option; for Spectral-norm, we ported the original Java benchmark to X10, avoiding parallel constructs.

2. **X10 with atomic** is parallel X10, but without using phaser accumulators. We effect reduction via phaser barriers and the X10 *atomic* construct. As in [14] we use a "lightweight" X10 version with regular Java arrays to avoid the large overheads incurred on X10 arrays in the current X10 implementation. However, all the other characteristics of X10 (*e.g.,* non-null used as the default type declaration and forbidden use of non-final static fields) are preserved faithfully in all the X10 versions. For the Spectral-norm benchmark, we parallelized the three inner reduction loops.

3. **X10 with phaser accumulators (eager)** is the delay-free eager implementation of the phaser accumulator described in Section 4.1.

4. **X10 with phaser accumulators (eager-with-delay)** adds delay between failed atomic updates to reduce memory interconnect bandwidth overhead. The delay is implemented as a busy-wait loop as follows:

   ```
   delay = BASE + COEF*random;
   for (i = 0; i < delay; i++) ;
   ```

   We use a random delay function whose cost is proportional to $\texttt{BASE} + \texttt{COEF} \times \texttt{random}$ ($0 \leq \texttt{random} < 1$). We used values of $\texttt{BASE} = 50$ and $\texttt{COEF} = 100$ for the performance experiments.

5. **X10 with phaser accumulators (dynamic-lazy)** is the dynamic-lazy implementation of the phaser accumulator described in Section 4.2.

6. **X10 with phaser accumulators (fixed-lazy)** implements the fixed-lazy phaser accumulator with a static cap on the number of activities as described in Section 4.3. This trades reduced accumulation overhead against limited dynamic parallelism.

Eager			
	activity	phaser	accumulator
allocation (accumulator)	–	add accum to accum list	allocate atomic shared variable
registration (phaser)	–	–	–
send (accumulator)	–	–	atomic update
next (phaser)	–	store result for each accum	store result
Dynamic-Lazy			
	activity	phaser	accumulator
allocation (accumulator)	–	add accum to accum list	–
registration (phaser)	store pointer to `accumCell`	assign `accumCell`s for each accum	allocate new `accumCell`
send (accumulator)	save result in `accumCell`	–	–
next (phaser)	–	reduction for each accum	reduction over `accumCell` list
Fixed-Lazy			
	activity	phaser	accumulator
allocation (accumulator)	–	add accum to accum list	allocate array
registration (phaser)	store index to accum array	assign indexes for each accum	–
send (accumulator)	store result in array	–	–
next (phaser)	–	reduction for each accum	reduction over array

Table 1. Actions taken in activities, phasers, and accumulators during reductions

For all runs, the main program was extended with a three-iteration loop within the same Java process, and the best of the three times was reported in each case. This configuration was deliberately chosen to reduce/eliminate the impact of JIT compilation time in the performance comparisons.

In an effort to reduce the number of variables that differ between the two platforms, we used the same X10 version in both cases (version 1.5 [17]) modified to use phasers instead of clocks. All X10 runs were performed with the following common X10 options:

```
-BAD_PLACE_RUNTIME_CHECK=false
-NUMBER_OF_LOCAL_PLACES=1
-PRELOAD_CLASSES=true -BIND_THREADS=true
```

In addition, `-INIT_THREADS_PER_PLACE` was always set to the number of CPUs for which the measurement was being performed.

All results in this paper were obtained on three SMP platforms. The first is an 8-way (2 Quad-Core) AMD Opteron Processor 8347 1.9 GHz SMP server with 4GB main memory running Fedora Linux release 8. For X10 test runs, we use the IcedTea Runtime Environment (build 1.7.0-b21) with the IcedTea 64-Bit Server VM (build 1.7.0-b21, mixed mode) and the "-Xms1000M -Xmx1000M" options. In addition, we used the Intel version 10.1 C/C++ compiler to measure the performance of the OpenMP version of Syncbench.

The second platform is a 16-way (4 Quad-Core) Intel Xeon E7330 2.4 GHz SMP with 32 GB main memory running Fedora Linux release 8. For X10 test runs, we use the IcedTea Runtime Environment (build 1.7.0-b21) with the IcedTea 64-Bit Server VM (build 1.7.0-b21, mixed mode) and the "-Xms1000M -Xmx1000M" options. We used the Intel version 11.0 C/C++ compiler to measure the performance of the OpenMP version of Syncbench.

The third platform is a 64-way (8 cores $\times$ 8 threads/core) 1.2 GHz UltraSPARC T2 (Niagara 2) with 32 GB main memory running Solaris 10. We conduct all X10 tests in the Java 2 Runtime Environment (build 1.5.0_12-b04) with Java HotSpot Server VM (build 1.5.0_12-b04, mixed mode) and the "-Xms1000M -Xmx1000M" options. Sun's C compiler v5.9 was used for the OpenMP version of Syncbench and Spectral-norm.

5.1 Speedup for EPCC Syncbench microbenchmark

This section presents the results for the EPCC Syncbench microbenchmark. The results for X10 atomic are not shown in the figures because they are significantly worse than the OpenMP and phaser accumulator results; including them in the graph obscures the remaining results and does not provide any new insights. The default X10 atomic implementation uses coarse grained locking that is not efficient. In contrast, the phaser accumulators eager implementation takes advantage of efficient updates to atomic variables using the Java Concurrency Utils library and the fixed-lazy and dynamic-lazy do not require updates to a shared variable (and thus do not require any locking). The comparative results for X10 atomic are included in the discussion below for completeness.

Figure 8 shows the accumulation overhead of OpenMP reduction and phaser accumulators with different imple-

Benchmark	Data Size	Description
Syncbench		EPCC OpenMP microbenchmark measuring time/reduction (Data size is same as number of threads)
CG	S, W, A	From NAS parallel benchmarks (S is the smallest, A is in between sizes S,W,A,B,C)
Spectral-norm	N=6000	From the Computer Language Benchmarks Game (6000×6000 matrix eigenvalue)

Table 2. List of benchmarks

mentations using the EPCC Syncbench microbenchmark on an 8-core Opteron SMP. The left chart of Figure 8 is the case of a balanced workload, in which all activities have the same amount of local work and initiate send operations at about the same time. The right chart represents the accumulation overhead when activities have unbalanced workloads. The accumulation overhead of phaser accumulators with fixed-lazy implementation is lower than that of OpenMP (2.54× with 8 cores) and X10's atomic (16.5×) in the balanced workload case, and than OpenMP (2.97×) and X10's atomic (7.64×) in the unbalanced load case. For different accumulator implementations, fixed-lazy has lower overhead than eager (1.34× with 8 cores), eager-with-delay (1.64×) and dynamic-lazy (1.08×) when the workload for each activity is balanced. On the 8-CPU Opteron SMP, adding delay to `sendEager` method didn't show any performance improvement; we attribute this to the rarity of atomic access conflicts making bus traffic not be a performance bottleneck. In the unbalanced case shown in the right chart, the eager implementation gives almost the same performance as dynamic-lazy.

Figure 9 shows the accumulation overhead on a 16-core Xeon SMP. As with the Opteron results, the fixed-lazy has the best results among the various accumulator implementations. On this platform, however, the OpenMP implementation outperforms the best accumulator implementation by 2.2× in the unbalanced case and 2.61× in the unbalanced case when using all 16 cores. We discuss the OpenMP results in more detail below. Once again, adding a delay to the eager implementation degrades performance because the delay times outweigh the overhead incurred by aggressively performing the atomic update until it succeeds.

The results for Syncbench on the 64-way UltraSPARC T2 SMP are shown in Figure 10. The results show that the overhead of phaser accumulators with the fixed-lazy implementation is significantly lower than that of OpenMP (14.9× with 64 threads), and X10's atomic (94.8×) in the balanced case; and lower overhead than OpenMP (8.48×), and X10 atomic (30.3×) in the unbalanced case. Across implementations, fixed-lazy has the lowest overhead at 1.94× below eager, 1.23× below eager-with-delay, and 1.26× below dynamic-lazy with balanced workloads. We note that the delay approach described in Section 4 improved eager performance by 1.57× with 64 threads. In the unbalanced load case shown in the right chart, Eager's overhead is lower than that of eager-with-delay (1.03 × with 64 threads) and dynamic-lazy (1.18×), and almost equal to fixed-lazy's overhead. The major computation of eager is from atomic updates in `sendEager`, which can overlap with other computation. Phaser accumulators with the fixed-lazy implementation show the lowest overhead in most cases on all of our test machines, however, this comes at the cost of dynamic parallelism.

The variety in the performance results between OpenMP and our accumulators underscores the importance of experimenting with different platforms and runtimes when benchmarking a new parallel programming construct. Unfortunately, it is difficult to provide a detailed analysis of the performance benefits of phaser accumulators compared to OpenMP because we used the vendor compiler's implementation of OpenMP (Intel compiler on the Opteron and Xeon and Sun's compiler on the UltraSPARC T2), and we do not have access to the implementation details of the OpenMP reductions implemented by those compilers. We suspect that the techniques presented in our paper would be applicable to an OpenMP implementation and would like to know why OpenMP is better on the Xeon, but can not attribute performance benefits to a specific technique without a direct comparison of the implementations.

5.2 Speedup at full capacity: NPB CG and Spectral-norm

The left side of Figure 11 compares performance using 8 cores on the Opteron SMP for NPB CG with sizes S, W and A, in addition to Spectral-norm benchmark. On average, the speedup obtained by X10 atomic is 2.32×, the speedup by phaser accumulator with eager implementation is 4.23×, eager-with-delay is 4.27×, dynamic-lazy is 4.80× and fixed-lazy implementation is 4.93×.

The left side of Figure 12 shows the performance comparison using 16 threads on the Intel Xeon. The speedup

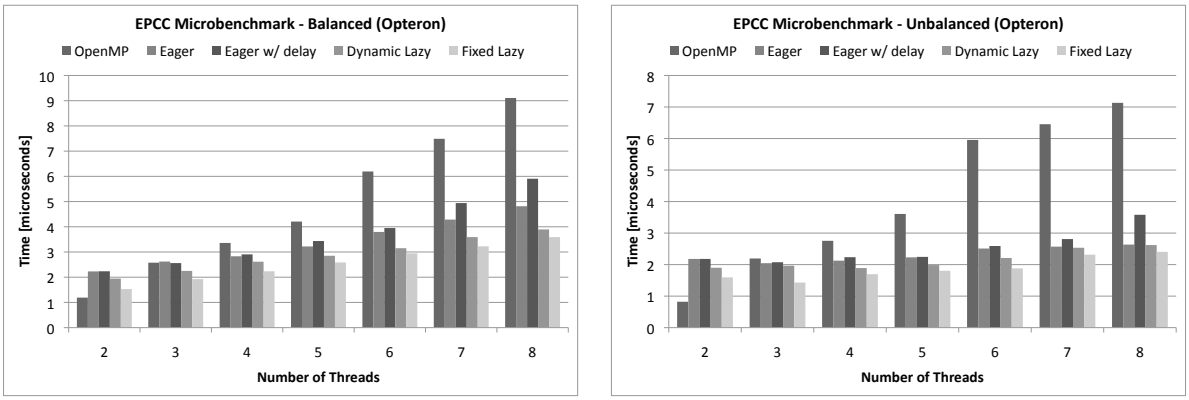

Figure 8. Syncbench microbenchmark results on 8-way AMD Opteron 8347 SMP

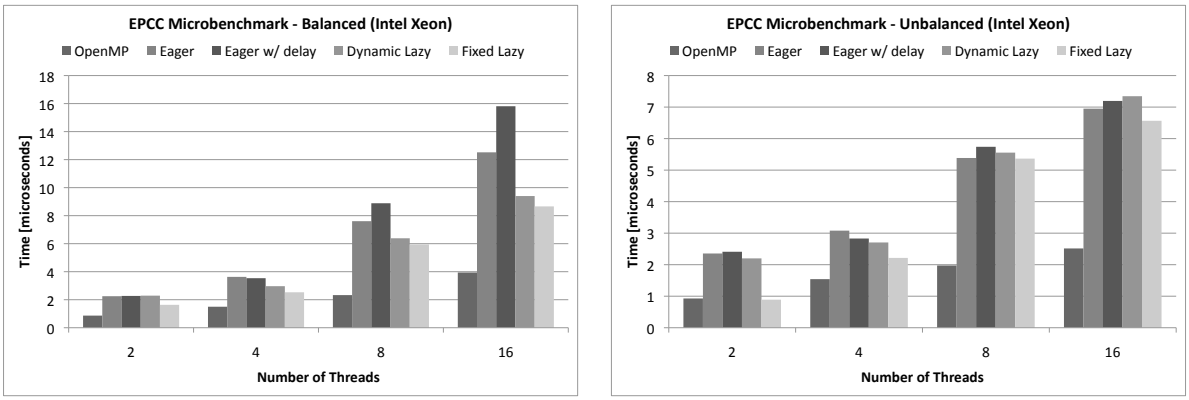

Figure 9. Syncbench microbenchmark results on 16-way Intel Xeon

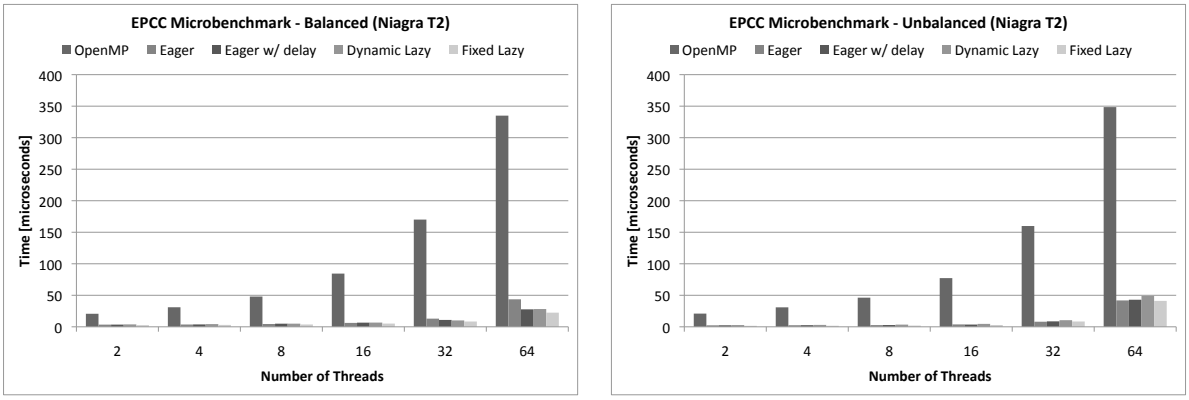

Figure 10. Syncbench microbenchmark results on 64-way Sun UltraSPARC T2

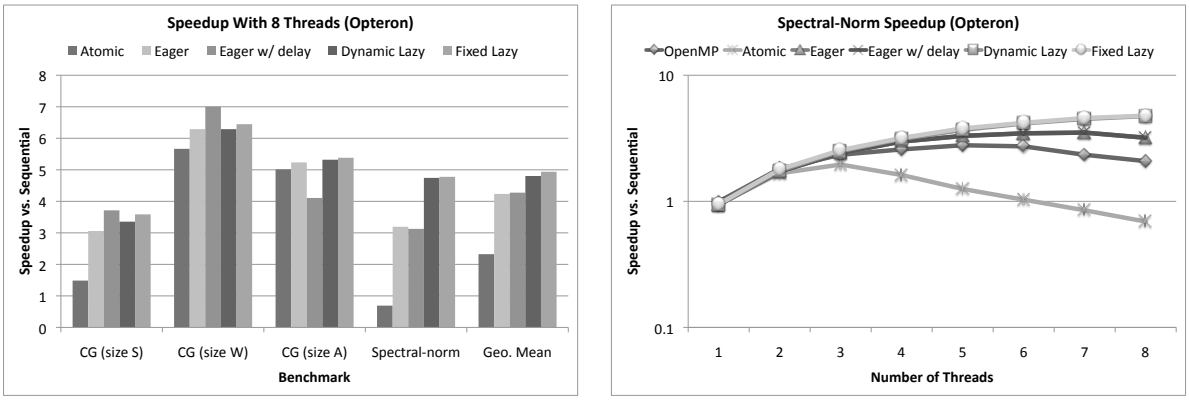

Figure 11. Speedup on 8-way AMD Opteron relative to serial X10 version with max threads across all benchmarks (Left) and detailed speedup curve for spectral-norm (Right).

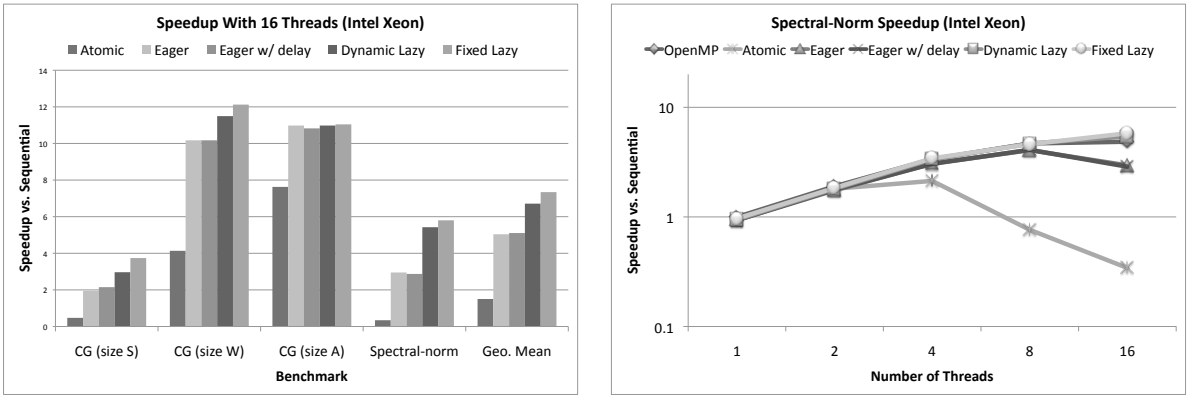

Figure 12. Speedup on 16-way Intel Xeon relative to serial X10 version with max threads across all benchmarks (Left) and detailed speedup curve for spectral-norm (Right).

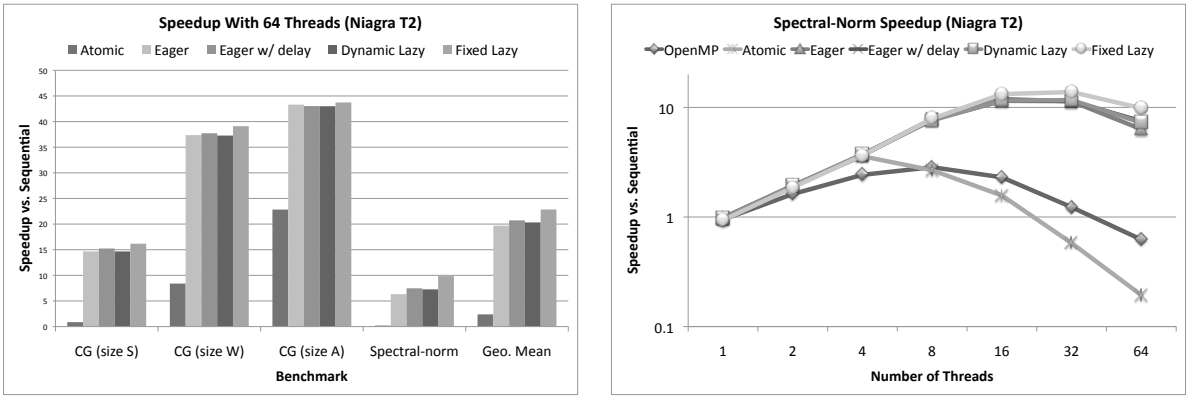

Figure 13. Speedup on 64-way UltraSPARC T2 relative to serial X10 version with max threads across all benchmarks (Left) and detailed speedup curve for spectral-norm (Right).

results show similar trends to the Opteron. The lazy implementations average a speedup of around 7×, the eager implementations average a speedup of about 5×, and X10 atomic averages 1.50× speedup.

The left side of Figure 13 shows the performance comparison using 64 threads on the UltraSPARC T2 SMP. The speedup obtained by X10 atomic averages improvement by a factor of 2.4; eager phaser accumulator by 19.7×; eager-with-delay by 20.7×; dynamic-lazy by 20.3×; and fixed-lazy implementation by 22.9× – all relative to sequential X10. The next section describes the detailed speedup results for the Spectral-norm benchmark.

5.3 Speedup for the Spectral-norm microbenchmark

The right side of Figure 11 shows the speedup for the Spectral-norm benchmark on the Opteron SMP. We measured speedup relative to the serial X10 version, without any parallel constructs. The OpenMP reduction peaks at five cores and the X10 atomic implementation at only three. By comparison, phaser accumulators scale to seven or eight cores. Relative to the serial X10 version, the speedup with all eight cores for the OpenMP reduction is a factor of 2.08; X10 atomic is 0.69×; the eager phaser accumulator is 3.19×; eager-with-delay is 3.19×; fixed-lazy is 4.74×; and dynamic-lazy is 4.77×.

The right side of Figure 12 shows the Spectral-norm performance comparison on the Intel Xeon. The phaser accumulator and OpenMP versions scale almost linearly up to four cores. The eager accumulator implementations drop from a speedup of just over 4× at 8 cores to just under 3× at 16 cores, indicating a bottleneck caused by shared variable atomic updates. The fixed-lazy accumulators and OpenMP implementations continue to get speedups up to 16 cores (4.85× for OpenMP and 5.80× for accumulators), though they both hit a pronounced speedup knee in the curve at four threads.

The right side of Figure 13 shows the performance comparison for Spectral-norm on the UltraSPARC T2 SMP. We again see that phaser accumulators have greater scalability than OpenMP reduction and X10 atomic. Where the 64-thread speedup ratio of OpenMP is 0.62× and X10 atomic is 0.19×, phaser accumulators show 6.33× speedup for eager, 7.47× for eager-with-delay, 7.26× for dynamic-lazy, and 9.85× for fixed-lazy.

6 Related Work

It is well known from past work that reductions and scans on commutative and associative operations can be performed in parallel [3], and programming models such as OpenMP [11] and MPI [16] have introduced special primitives for such cases.

In OpenMP, a parallel construct can optionally include a clause which specifies a reduction operator and a list of scalar shared locations. For each shared location, a private copy array is allocated sized to the number of implicit/explicit tasks created in the parallel construct, initialized to the identity element for the operator. On exit from the reduction, these arrays are populated with the values of the private copies in accordance with the specified operator. Supported reduction operators include sum, product, min, max, logical-or, logical-and, bitwise-or, bitwise-and, and bitwise-xor. It is the user's responsibility to prevent race conditions on the shared location parallel to the reduction. In contrast, there are no race conditions on a phaser accumulator since it is managed by the phaser runtime; further, a reduction on a phaser accumulator can be performed on any user-selected dynamic set of activities by allowing the user to control activity-phaser registrations as well as which activities perform send operations.

In MPI, reductions are embodied in the following collective routines: `MPI_Reduce`, `MPI_AllReduce`, `MPI_Reduce_scatter` and `MPI_Scan`. The set of possible reduction operators includes those supported by OpenMP and also `MINLOC` and `MAXLOC`. Each process participating in a reduction provides a pointer to a source location for the operand that it sends as input, and a pointer to a destination location to receive the result. It is the user's responsibility to ensure that the pointers are valid and that all processes in the same communicator invoke matching collective routines at the same time. By default, the matching calls are synchronized by a barrier operation. In contrast, a phaser accumulator uses a value-based interface thereby avoiding pointer-related errors; further, send operations can be performed asynchronously by different activities, as evidenced by the eager implementation, and a sound no-op semantics is provided for activities that do not perform a send operation on a specific accumulator in a specific phase.

Space limitations prevent us from providing a more detailed comparison with related work. To the best of our knowledge, no prior reduction construct supports the dynamic parallelism and asynchrony outlined above for phaser accumulators.

7 Conclusions and Future Work

In this paper, we introduced the *phaser accumulator*, a new reduction construct for dynamic parallelism. We presented three different implementations to support the phaser accumulator model: eager, dynamic-lazy, and fixed-lazy. Performance results obtained from a portable implementation of phaser accumulators on three different SMP platforms demonstrate that they can deliver superior perfor-

mance to existing reduction constructs like X10 atomic and OpenMP reductions. Also, our experiment showed that inserting a delay in atomic update operations can reduce congestion and improve the scalability of the eager implementation.

Opportunities for future research related to phasers include implementations on future many-core processors (such as Cyclops and Larrabee) and on distributed-memory clusters, and new compiler analyses and optimizations for accumulator operations. In addition, we plan to explore scan operations, user-defined reductions (including minloc, maxloc) on user-defined types, a tournament barrier implementation of phasers for reducing the serial overhead in the dynamic-lazy implementation, and stream parallelism using phaser accumulators.

Acknowledgments

We are grateful to all X10 team members for their contributions to the X10 software used in this paper. We would like to especially acknowledge Vijay Saraswat's work on the design and implementation of clocks in the current X10 SMP implementation. We gratefully acknowledge support from an IBM Open Collaborative Faculty Award, and from Microsoft fund R62710-792. Finally, we would like to thank AMD for their donation of the 8-CPU Opteron SMP system, Keith Cooper for access to the Intel Xeon system, and Doug Lea for access to the UltraSPARC T2 SMP system.

References

[1] Shivali Agarwal, Rajkishore Barik, Dan Bonachea, Vivek Sarkar, Rudrapatna K. Shyamasundar, and Katherine Yelick. Deadlock-free scheduling of x10 computations with bounded resources. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on parallelism algorithms and architectures*, pages 229–240, New York, NY, USA, 2007. ACM.

[2] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):6–16, 1990.

[3] G. E. Blelloch. *Vector Models for Data Parallel Computing*. MIT Press, Cambridge, MA, 1990.

[4] 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.

[5] Steve Deitz. Parallel programming in Chapel. http://www.cct.lsu.edu/ estrabd/LACSI2006/Programming2006.

[6] Jack Dongarra. Scheduling for numerical linear algebra library at scale. Keynote presentation, UW MSR Summer Institute 2008, http://www.cs.washington.edu/mssi/2008/talks/dongarra_msr_uw_0808.pdf, 2006.

[7] Rajiv Gupta. The fuzzy barrier: a mechanism for high speed synchronization of processors. In *ASPLOS-III: Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 54–63, New York, NY, USA, 1989. ACM.

[8] Habanero multicore software research project web page. http://habanero.rice.edu, 2008.

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

[10] John Mellor-Crummey and Michael Scott. Algorithms for Scalable Synchronization on Shared Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[11] OpenMP Application Program Interface, version 3.0, May 2008. http://www.openmp.org/mp-documents/spec30.pdf.

[12] Parallel programming with .net. http://blogs.msdn.com/pfxteam/, 2008.

[13] Vivek Sarkar. Synchronization Using Counting Semaphores. *Proceedings of the ACM 1988 International Conference on Supercomputing*, pages 627–637, July 1988.

[14] Jun Shirako, Hironori Kasahara, and Vivek Sarkar. Language extensions in support of compiler parallelization. In *The 20th International Workshop on Languages and Compilers for Parallel Computing (LCPC'07)*, 2007.

[15] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 277–288, New York, NY, USA, 2008. ACM.

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

[17] Release 1.5 of X10 system dated 2007-06-29. http://sourceforge.net/project/showfiles.php?group_id=181722 &package_id=210532&release_id=519811, 2007.

[18] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael Welcome, and Tong Wen. Productivity and performance using partitioned global address space languages. In *PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 24–32, New York, NY, USA, 2007. ACM.