Phasers: a Unified Deadlock-Free Construct for Collective and Point-to-point Synchronization

J. Shirako D. M. Peixotto V. Sarkar W. N. Scherer III

Department of Computer Science, Rice University, 6100 Main Street, Houston TX 77025

{shirako, dmp, vsarkar, scherer}@rice.edu

ABSTRACT

Coordination and synchronization of parallel tasks is a major source of complexity in parallel programming. These constructs take many forms in practice including mutual exclusion in accesses to shared resources, termination detection of child tasks, collective barrier synchronization, and point-to-point synchronization. In this paper, we introduce *phasers*, a new coordination construct that unifies collective and point-to-point synchronizations. We establish two safety properties for phasers: *deadlock-freedom* and *phase-ordering*. Performance results obtained from a portable implementation of phasers on three different SMP platforms demonstrate that phasers can deliver superior performance to existing barrier implementations, in addition to the productivity benefits that result from their generality and safety properties.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms

Languages

1. INTRODUCTION

The computer industry is entering a new era of mainstream parallel processing in which the need for improved productivity in parallel programming has taken on a new urgency. One of the major obstacles to improved productivity is the complexity of coordination and synchronization of parallel tasks that is inherent in current parallel programming models. Coordination and synchronization constructs take many forms in practice such as mutual exclusion in accesses to shared resources using locks, termination detection of child threads using join operations, collective synchronization using barriers, and point-to-point synchronization using semaphores. Recent efforts on productivity improvements in parallel programming include transactional memory systems for mutual exclusion [7], work stealing schedulers to support dynamic nested parallel execution model with lightweight task creation and termination detection as in Cilk [11], and deadlock-free barrier synchronizations as in X10's clocks [2]. These approaches have motivated new research on delivering efficient parallel performance, while retaining safety guarantees that are important for productivity.

In this paper, we focus on reducing the complexity of collective and point-to-point synchronization, both of which are used extensively in parallel algorithms. We introduce *phasers*, a new coordination construct that unifies collective and point-to-point synchronization. We establish two safety properties for phasers: *deadlock-freedom* and *phase-ordering*. Performance results obtained from a portable implementation of phasers on three different SMP platforms demonstrate that they can deliver superior performance to existing barrier implementations, in addition to the productivity benefits that result from their generality and safety properties.

The rest of the paper is organized as follows. The phaser construct is introduced in Section 2, and its phase-ordering and deadlock-avoidance properties are established in Section 3. Section 4 presents experimental results to compare the performance of phasers with other synchronization constructs, using an SMP implementation of phasers developed in the Habanero multicore software research project at Rice University [6]. Section 5 discusses related work, and Section 6 contains our conclusions.

2. OVERVIEW OF PHASERS

In this section, we introduce *phasers*, a new coordination construct that unifies collective and point-to-point synchronizations and embodies the following contributions:

1. **Integration of producer-consumer with barrier synchronization.** An activity has the option of registering with a phaser in *signal-only* mode or *wait-only* mode for producer-consumer synchronization, in addition to *signal-wait* mode for barrier synchronization.
2. **Support for single statements.** A *next* statement for phasers can optionally include a *single* statement which is guaranteed to be executed exactly once during a phase transition [16].
3. **Scalable implementation on multicore SMPs.** By design, phasers are amenable to scalable implementation on multicore SMPs, as demonstrated in Section 4.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICS'08, June 7–12, 2008, Island of Kos, Aegean Sea, Greece.
Copyright 2008 ACM 978-1-60558-158-3/08/06 ...$5.00.

```
int iters = 0; delta = epsilon+1;
while ( delta > epsilon ) {
  finish {
    for ( jj = 1 ; jj <= n ; jj++ ) {
      final int j = jj;
      async {
        newA[j] = (oldA[j-1]+oldA[j+1])/2.0f ;
        diff[j] = Math.abs(newA[j]-oldA[j]);
      } // async
    } // for
  } // finish
  delta = diff.sum(); iters++;
  temp = newA; newA = oldA; oldA = temp;
}
System.out.println("Iterations: " + iters);
```

Figure 1: One-Dimensional Iterative Averaging in X10 using Async and Finish

These properties, along with the generality of *dynamic parallelism* and the *phaser-ordering* and *deadlock-freedom* safety properties, distinguish phasers from synchronization constructs in past work including barriers [5, 10], counting semaphores [12] and X10's clocks [2].

Section 2.1 briefly summarizes the async, finish and clock constructs from X10 [2], which provide the context for our work on collective and point-to-point synchronization. Section 2.2 introduces the phaser construct and the operations that can be performed on phasers. Though the description of phasers in this paper may appear to be specific to X10, they are a general unification of point-to-point and collective synchronizations that can be incorporated in any parallel programming model with a shared address space such as OpenMP, Intel's Thread Building Blocks, Microsoft's Task Parallel Library, Java Concurrency Utilities, Unified Parallel C, Co-Array Fortran and Titanium.

2.1 Async, Finish, Clocks

This section provides a brief summary of the async, finish, and clock constructs introduced in v0.41 of the X10 programming language [2].

2.1.1 *async* ⟨*stmt*⟩

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.

2.1.2 *finish* ⟨*stmt*⟩

The X10 statement, *finish* ⟨*stmt*⟩, causes the parent activity to execute ⟨*stmt*⟩ and then wait till all sub-activities created within ⟨*stmt*⟩ have terminated (including transitively spawned activities). There is an implicit *finish* statement surrounding the main program in an X10 application.

Each dynamic instance of a `finish` statement can be viewed as being bracketed between 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*

```
finish async { // Outer async for clock allocation
  delta.f = epsilon+1; iters.i = 0;
  final clock C = clock.factory.clock();
  for ( jj = 1 ; jj <= n ; jj++ ) {
    final int j = jj;
    async clocked(C){
      while ( delta.f > epsilon ) {
        newA[j] = (oldA[j-1]+oldA[j+1])/2.0f ;
        diff[j] = Math.abs(newA[j]-oldA[j]);
        next; // Barrier
        if (j == 1) {
          delta.f = diff.sum(); iters.i++;
        }
        next; // Barrier
        temp = newA; newA = oldA; oldA = temp;
      } // while
    } // async
  } // for
} // finish async
System.out.println("Iterations: " + iters.i);
```

Figure 2: One-Dimensional Iterative Averaging in X10 using Async, Finish and Clocks

is introduced from the last instruction of an activity to the *end-finish* node corresponding to the activity's IEF.

We use the pedagogical One-Dimensional Iterative Averaging program from [3] as a running example in this paper. The goal of this program is to perform iterative averaging on a one-dimensional array, A[0:n+1], initialized with A[0:n] = 0 and A[n+1] = n+1. A[0] = 0 and A[n+1] = n+1 are fixed boundary conditions, and a 2-point stencil is used to iteratively replace A[j] by the average of A[j-1] and A[j+1] for $1 \leq j \leq n$. The algorithm terminates when the sum of element changes from one iteration to the next is less than a given threshold. Figure 1 contains a simple X10 version of this example using `finish` and `async` constructs. Note that the async activities created in iterations of the j loop can all run in parallel, and that the finish statement ensures that all the async's have terminated before execution proceeds to the `diff.sum()` computation.

2.1.3 *Clocks*

While the code in Figure 1 is correct, the overhead of repeatedly spawning and terminating activities in each iteration of the `while` loop and the potential accompanying loss of locality can be a major performance bottleneck. Instead, it would be preferable to perform a ÒbarrierÓ-like coordination within each iteration of the `while` loop, without terminating the activities involved. This serves as one of the motivations for X10's *clocks*.

An X10 activity, A_i, can allocate a clock, C, and then create a child activity A_j registered with C by using the clocked async construct, "`async clocked(C) <stmt>`". A_j starts executing in the same clock phase as its parent, A_i. Note that, unlike barriers in standard SPMD models, X10's clock construct permits the set of activities registered with a clock to vary dynamically.

When multiple activities (such as A_i and A_j) registered with the same clock need to perform a collective barrier synchronization, they do so by executing a `next;` statement which forces the activity to suspend until all clocks with

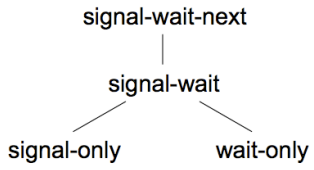


Figure 3: Capability lattice for phasers

which it is registered can advance. A clock can advance only when all activities that are registered with it execute a `next;` statement. X10 also permits an activity to *drop* a clock, which implicitly unregisters it with C. A terminating activity implicitly drops all clocks with which it is registered.

Figure 2 shows an X10 version of the Iterative Averaging example that uses clocks. A key difference from the previous version is that the while loop is now pushed inside the body of each async, so activities are not created and terminated in every iteration of the while loop.

2.2 Phasers

A *phaser* is a synchronization object that supports the operations described 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 3.

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`. Phaser creation also initializes the phaser to its first phase (phase 0).

2. **phased async:** `async phased (` $mode_1(ph_1), \ldots$ `)`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. The following constraints are imposed on the transmission of phasers:

   (a) **Capability rule:** A_i can only transmit a capability on phaser ph to A_j if A_i itself has that capability or higher (Figure 3). The capability rule is imposed to avoid race conditions on phaser operations.

   (b) **IEF Scope rule:** A_i can only transmit a capability on phaser ph to A_j if the creation (`new`) instruction for ph has the same Immediately Enclosing Finish (c.f. Section 2.1) as the async statement for A_j *i.e.,* both ph and A_j were created in the scope of the same dynamic finish statement. The IEF rule is imposed to avoid a potential deadlock between finish operations and wait/next operations on phasers.

   An attempt to transmit a phaser that does not obey the above two rules will result in a PhaserException being thrown at runtime. We allow the following default syntax to indicate that A_i is transmitting all its capabilities on all phasers that it is registered with to A_j, "A_i : `async phased` A_j".

3. **drop:** There are two ways in which A_i can drop a registration that it holds with a phaser:

   (a) **Activity termination.** When A_i terminates execution, it performs an implicit `next` operation, and then completely de-registers from each phaser that it was registered with.

   (b) **End-finish.** When A_i executes an end-finish instruction for finish statement F, it completely de-registers from each phaser ph created by A_i in the scope of F *i.e.,* for which F is the IEF for the creation statement of ph. Thus, the lifetimes of phasers naturally follow the lexical scoping of finish constructs.

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 indicated in Table 1, the semantics of `next` depends on the registration mode that A_i has with a specific phaser, ph.

5. **next with single statement:** The `next` ⟨`stmt`⟩ operation has the semantics of a `next` statement as defined above, with the extension of executing `stmt` as a single statement. This operation is only permitted if A_i is registered in *signal-wait-next* mode on the phaser (see Table 1). Further, we require all other activities registered with the phaser in *signal-wait-next* mode and executing a next with single statement must execute the same static `next` ⟨`stmt`⟩ statements. These constraints are imposed to ensure the integrity of the single statement [16].

6. **Phaser-specific signal:** We permit A_i to perform a phaser-specific signal operation, `ph.signal()`, for any phaser ph on which it is registered with a `signal` capability. The operation performed depends on A_i's registration mode for ph. If the registration mode is *signal-wait-next* or *signal-wait*, then `ph.signal()` effectively converts ph into a *fuzzy barrier* [5] for A_i by allowing local work to be performed between the `ph.signal()` and `next` operation.

7. **signal:** A `signal` operation performed by A_i is shorthand for a `ph.signal()` operation performed on each phaser ph with which A_i is registered with a `signal` capability.

Figure 4 shows the X10 iterative averaging example from Figure 2 rewritten to use phasers. There are two major differences between the phaser version and X10-clock version. First, the two `next` statements and their intervening conditional statement that computes `diff.sum()` in the X10 version have been combined into one next-with-single statement in the phaser version. Not only does this reduce overhead, but it also lets the runtime take the responsibility for determining which activity should execute the *single* statement *i.e.,* which activity should be the *master* (using the terminology from Section 4). The second major difference is that the programmer is not required to insert an additional `async` at the outer level, as in the X10 version. X10 prohibits a

Operation	Registration Mode	Action
`next`	*signal-wait-next* or *signal-wait*	`signal` + `wait` (or just `wait` if previous operation was `signal`)
	signal-only	`signal`
	wait-only	`wait`
`next` ⟨`stmt`⟩ (next w/ single stmt)	*signal-wait-next*	`signal` + `wait` + single execution of `stmt` (or just `wait` + single execution of `stmt` if previous op was `signal`)
	signal-wait	*error*
	signal-only	*error*
	wait-only	*error*
`ph.signal()`	*signal-wait-next* or *signal-wait*	`signal` on `ph` (or *error* if previous op by activity on `ph` was `signal`)
	signal-only	`signal` on `ph`
	wait-only	*no-op*
`signal`	*	Perform `ph.signal()` on each phaser that activity has a `signal` capability on

Table 1: Semantics of phaser operations as a function of registration modes

```
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 { // Single statement
          delta.f = diff.sum(); 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 Phasers for Collective Synchronization

clocked async from being created immediately in the scope of a finish to avoid a potential deadlock with the end-finish operation. Phasers instead avoid deadlock by enforcing the IEF scoping rule on phasers, which ensures that all activities have de-registered from phaser *ph* after the end-finish instruction.

Figure 5 shows an alternate version of the iterative averaging example as an illustration of the use of phasers for point-to-point synchronization instead of barrier synchronization. For simplicity, this version uses a `for` loop with a fixed number of averaging iterations in each async activity, so it will not produce the same output as the version in Figure 4 which used a `while` loop to control termination. In this example, each *async* is registered with three phasers — one in *signal-only* mode and two in *wait-only* mode. Note that phasers gracefully handle boundary conditions that often arise in point-to-point synchronization. For example, it is not necessary to provide a `signal` operation to match the `wait` on `ph[j-1]` in the *async* for the `j = 1` iteration or on `ph[j+1]` in the *async* for the `j = n` iteration.

```
finish {
  phaser[] ph= new phaser[n+2]; // array of phasers
  for ( jj = 1 ; jj <= n ; jj++ ) {
    final int j = jj;
    async phased(signalOnly(ph[j]),
            waitOnly(ph[j-1]), waitOnly(ph[j+1]) ) {
      for ( int iter=0 ; iter<NUM_ITERS ; iter++ ) {
        newA[j] = (oldA[j-1]+oldA[j+1])/2.0f ;
        signal; // Signals ph[j]
        // Activity can do local work here
        next; // Await signals from iters j-1 & j+1
        temp = newA; newA = oldA; oldA = temp;
      } // for
    } // async
  } // for
} // finish
```

Figure 5: Alternative version of One-Dimensional Iterative Averaging using Phasers for Point-to-Point Synchronization

3. PHASE-ORDERING AND DEADLOCK-AVOIDANCE IN PHASERS

In this section we precisely define the guarantees phasers provide for phase ordering and deadlock avoidance by building on the the notion of an *X10 computation DAG* introduced in [1] for *finish* and *async* constructs. Due to space limitations, these safety properties are established by informal proof sketches rather than formal proofs.

3.1 Phase ordering

We model execution constraints using a Dynamic Computation DAG (DCD). The DCD has a node for each instance of an instruction executed during a computation. The edges in the DCD encode constraints in the execution order. For the computation to be correct, an instruction may not execute until all of its predecessors have executed. In addition to the normal instruction nodes, two special *accumulator* nodes are added to model phasers. The `begin-accum` and `end-accum` nodes are used to enforce constraints between phases so that instructions following a wait will not execute until all signals for that phase have executed.

The different types of edges in the DCD are discussed below.

1. **continue:** A *continue* edge represents sequential control flow. There is an edge from node A to B in the DCD if the computation executes the instruction sequence $A; B$ in the same activity.

2. **async:** An *async* edge is added from node A to B if A is an `async` instruction and B is the first instruction in the new activity.

3. **finish:** A *finish* edge is added from node A to B if A is the last instruction of an activity and B is the `end-finish` instruction for the IEF of that activity.

4. **next:** A `next` instruction decomposes into a `signal` followed by a `wait` instruction, with edges added for both instructions as described below.

5. **signal:** A *signal* edge is added from node A to B if A is a `signal` instruction and B is the `begin-accum` node for the phase of the phaser to which the signal is sent.

6. **wait:** A *wait* edge is added from node A to B if A is the `end-accum` node for the phase of the phaser and B is a `wait` instruction performed on that phaser.

7. **next with single:** An edge is added from the `signal` node to the `begin-accum` node as normal. The nodes in the single section are sequenced with continue edges starting from the `begin-accum` node and ending at the `end-accum` node. Finally, edges are added from the `end-accum` node to all `wait` nodes in the same manner as for a standard wait edge.

We annotate each node in the DCD with the current signal and wait phase for each phaser the activity is registered with. For each phaser ph with which the activity is registered, define the functions

- $\mathcal{S}(ph, i) : P \times Insts \rightarrow \mathbf{N} \cup \{\infty\}$ is the number of signal operations that have been performed on phaser ph by the current activity (or ancestors) before the activity executes instruction i

- $\mathcal{W}(ph, i) : P \times Insts \rightarrow \mathbf{N}$ is the number of wait operations that have been performed on phaser ph by the current activity (or ancestors) before the activity executes instruction i

Here, $Insts$ is the set of all instruction instances, P is the set of all phasers, and $\mathbf{N}$ is the set of non-negative integers. The ancestors of an activity are defined recursively to be the parent of an activity (i.e. the activity that executed the `async` starting the current activity) and any ancestors of the parent. The main activity has no ancestors.

In addition to the $\mathcal{S}$ and $\mathcal{W}$ values associated with each node in the DCD, there are $\mathcal{S}$ and $\mathcal{W}$ values associated with each phaser that are independent of any activity. The values for the $\mathcal{S}$ and $\mathcal{W}$ functions are defined in Table 2.

Figure 6 shows an example of a DCD and the $\mathcal{S}$ and $\mathcal{W}$ values for two activities registered as *signal-wait-next* on the phaser ph. The example illustrates several features including a fuzzy barrier and next with single.

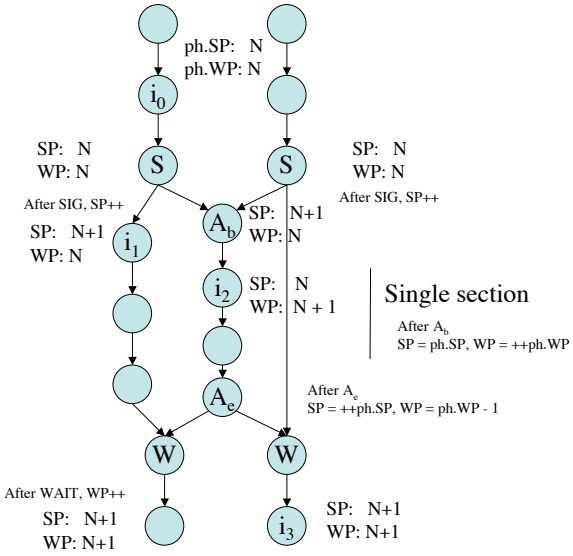


Figure 6: Example of a Dynamic Computation DAG and its associated $\mathcal{S}$ (SP) and $\mathcal{W}$ (WP) values for the phaser ph. The figure shows two `signal` nodes (S), two `wait` nodes (W), a `begin-accum` (A_b), and `end-accum` (A_e) node.

We can now precisely define the *phase-ordering property*, which states how the signal phase and wait phase relate to the execution order of instructions.

Phase-ordering property: Given two instruction instances i_1 and i_2 in the DCD, if $\exists$ a phaser $phs.t.\mathcal{S}(ph, i_1) < \mathcal{W}(ph, i_2)$ then $i_1 \prec i_2$ where $i_1 \prec i_2$ is shorthand for "i_1 precedes i_2".

We now establish that the definitions of $\mathcal{S}$ and $\mathcal{W}$ given in Table 2 satisfy the phase-ordering property. Consider four cases:

- The activity executing i_1 is registered on phaser ph as *wait-only*. Then $\mathcal{S}(ph, i_1) = \infty$, and the $\mathcal{S}(ph, i_1) < \mathcal{W}(ph, i_2)$ precondition will be false.

- The activity executing i_1 is registered on phaser ph as *signal-only*. If $\mathcal{S}(ph, i_1) < \mathcal{W}(ph, i_2)$ then i_2 is being executed by an activity that has waited on the phaser more times than the activity executing i_1 has signaled. It must be that i_1 is executed before i_2, because i_2 could not be executed without at least one more signal from the activity executing i_1.

- The activity executing i_1 is registered on phaser ph as *signal-wait*. If $\mathcal{S}(ph, i_1) < \mathcal{W}(ph, i_2)$ then the same argument follows as in the signal case above. If i_1 occurs in a split phase then we cannot establish any order on the activities executing in the next wait phase $\mathcal{W}(ph, i_1) + 1$ because we have signaled that they may continue on to phase $\mathcal{W}(ph, i_1) + 1$. The inequality will only hold for instructions two wait phases ahead, $\mathcal{W}(ph, i_1) + 2$.

- i_1 occurs in a next with single section. When i_1 occurs in the next with single section, it is executed with $\mathcal{S}$

Operation	Registration Mode		
	signal-wait-next or *signal-wait*	*signal-only*	*wait-only*
`create`	$\mathcal{S}$= 0 $\mathcal{W}$= 0	$\mathcal{S}$= 0 $\mathcal{W}$= 0	$\mathcal{S}$= ∞ $\mathcal{W}$= 0
`async`	$\mathcal{S}$= p.$\mathcal{S}$ $\mathcal{W}$= p.$\mathcal{W}$	$\mathcal{S}$= p.$\mathcal{S}$ $\mathcal{W}$= p.$\mathcal{W}$	$\mathcal{S}$= ∞ $\mathcal{W}$= p.$\mathcal{W}$
`signal`	$\mathcal{S}$= $\mathcal{S}$+1 $\mathcal{W}$= $\mathcal{W}$	$\mathcal{S}$= $\mathcal{S}$+1 $\mathcal{W}$= $\mathcal{W}$	ERROR ERROR
`wait`	$\mathcal{S}$= $\mathcal{S}$ $\mathcal{W}$= $\mathcal{W}$+1	ERROR ERROR	$\mathcal{S}$= ∞ $\mathcal{W}$= $\mathcal{W}$+1
`begin-accum`	$\mathcal{S}$= $ph.\mathcal{S}$ $\mathcal{W}$= $++ph.\mathcal{W}$	$\mathcal{S} = \mathcal{S}$ $\mathcal{W} = \mathcal{W}$	$\mathcal{S} = \mathcal{S}$ $\mathcal{W} = \mathcal{W}$
`end-accum`	$\mathcal{S}$= $++ph.\mathcal{S}$ $\mathcal{W}$= $ph.\mathcal{W} - 1$	$\mathcal{S} = \mathcal{S}$ $\mathcal{W} = \mathcal{W}$	$\mathcal{S} = \mathcal{S}$ $\mathcal{W} = \mathcal{W}$

Table 2: Definition of the functions $\mathcal{S}$ and $\mathcal{W}$. Here, p.$\mathcal{S}$ is the value of $\mathcal{S}$ for the parent activity that executed the `async` and $ph.\mathcal{S}$ is the private value of the signal phase for the phaser ph.

Benchmark	Data Size	Description
BarrierBench		From JGF thread v1.0 Section 1 (Data size is fixed in program)
LUFact (single)	C	From JGF thread v1.0 Section 2 (Size C is its largest size)
MolDyn (single)	B	From JGF thread v1.0 Section 3 (Size B is its largest size)
SOR (pt-to-pt)	C	From JGF thread v1.0 Section 2 (Size C is its largest size)
CG (single)	A	From NAS Parallel Benchmarks (A is in between sizes S,W,A,B,C)
MG (single)	A	From NAS Parallel Benchmarks (A is in between sizes S,W,A,B,C)
BarrierJacobi (single)		From JCIP code examples (2048×2048 matrix for 512 steps)

Table 3: Benchmark programs and their data sizes. SOR used point-to-point synchronization in the Java and phaser versions. All benchmarks labeled with (single) used a single statement.

and $\mathcal{W}$ values taken from the phaser itself. The activity executing in the single block is executing in the next wait phase, allowing it to bound the execution of instructions occurring before the single block. It is also executing in the current signal phase, allowing instructions occurring in the next wait phase to bound its execution. Finally, its wait phase will be the same as the signal phase of an instruction executed in the fuzzy phase of the barrier. Thus there is no order enforced between instructions in the single section and those executed in a fuzzy phase.

3.2 Deadlock avoidance

We use the same DCD to provide guarantees about the deadlock-avoidance property of phasers. An instruction instance in the DCD will not execute until all its predecessors have executed. Logical deadlock can only occur if there is a cycle in the DCD. This result does not address deadlock due to limited physical resources [1].

Deadlock results when the instruction i_1 has a predecessor in the DCD, i_2, such that there is a path from i_1 to i_2. In other words, if there is a cycle in the DCD then there is a deadlock situation because the instruction i_1 cannot execute until i_2 executes, but i_2 cannot execute until i_1 executes.

Deadlock-avoidance property: Phasers introduce no cycles into the DCD.

The semantics of phaser operations ensure that no cycle will be created in the DCD. There are two cases to consider where phasers might possibly introduce a cycle, but we will show that the semantics of phasers prevent any such cycles from being created.

First, a cycle could be created if an activity A_1 is waiting on a phaser that another activity A_2 is signaling on, and A_2 is waiting on a phaser for a signal from A_1. For this to happen, it must be the case that A_1 is waiting on a phaser but has not yet signaled on all the phasers it is registered with, preventing A_2 from proceeding. We can see from Table 1 that there is no way for an activity to wait on a phaser without first signaling on all of its phasers. There is no explicit `wait` in the programming model. The only way for an activity to perform a `wait` operation is by using `next` which explicitly signals all of an activity's phasers before waiting on them. Thus all signals are performed before all waits and no cycle will be introduced into the DCD.

The second case to consider is the interaction between phasers and the finish construct. A cycle could be created if an activity A_1 is waiting at a finish node for a spawned activity A_2 to complete, and the activity A_2 is waiting for a signal from A_1 on a shared phaser. The IEF Scope rule described in Section 2 prevents this situation from occurring. An activity can only transmit a phaser to another activity with the same IEF, and upon reaching a finish node the activity automatically drops all phasers created in the scope of that finish. The above two rules ensure that no cycle is created in the DCD between finish nodes and phaser synchronization nodes.

A combination of static and dynamic properties is sufficient to establish the deadlock-freedom and phase-ordering safety properties. An example of a static property is that an activity is prohibited from performing a wait operation on a specific phaser. Examples of dynamic properties include the Capability and IEF Scope rules described in Section 2, and ensuring that an activity does not perform two consecutive signals on a phaser on which it is registered in *signal-wait* mode. The advantage of runtime checks is that an exception is thrown at the point at which an error occurs, which helps in diagnosing the source of the error (analogous to runtime checks for null pointers and index-out-of-bounds).

4. EXPERIMENTAL RESULTS

In this section, we present experimental results using an SMP implementation of phasers developed in the Habanero multicore software research project at Rice University [6]. Sections 4.1, 4.2 and 4.3 contain the results obtained on an 8-CPU AMD Opteron 8347 SMP with 2 quad-core processor chips, a 64-CPU IBM Power5+ SMP with 32 dual-core processor chips, and a 64-way Sun UltraSPARC T2 system with 8 eight-core chips and 8 threads per core.

The results were obtained for the following versions of the seven benchmarks shown in Table 3:

1. **Sequential Java.** This version was used as the baseline for all speedup results except BarrierBench. For the Java Grande Forum benchmarks, this version was taken from version v2.0 of the JGF benchmark release [9] with one exception — the sequential version of SOR was obtained by serializing the parallel version in threadv1.0 because the v2.0 version uses a different algorithm from the threadv1.0 version. For NAS parallel benchmarks, this version was obtained by using the "serial" option, and for BarrierJacobi, this version was obtained by removing all parallel constructs from the JUC version [4].

2. **Threaded Java.** For the Java Grande benchmarks, this version was taken from version threadv1.0 of the JGF benchmark release [9]; for NAS parallel benchmarks, this version was obtained by using the parallel option, and for BarrierJacobi; this version was obtained from the JUC version [4].

3. **X10 with clocks.** This version represents the performance that is obtained from an X10 version implemented with standard clocks. As in [13] 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.

4. **X10 with tournament barriers.** This version replaces clock operations by tournament barriers [9] everywhere in the X10 program. Tournament barriers are not a standard construct in Java, but a hand implementation of a static synchronization pattern available with the JGF benchmarks – they do not provide any of the safety and dynamic features of phasers.

5. **X10 with phasers (unfixed master)** This version measures the performance of the SMP phaser implementation with the *unfixed master option* which enables the runtime to change the "master" activity for each phaser during any phase transition, thereby providing an opportunity for adapting to load imbalances.

6. **X10 with phasers (fixed master)** This version measures the performance of the SMP phaser implementation with the *fixed master option* which keeps the "master" activity fixed for each phaser *i.e.,* the master can only be changed when the current master activity drops its registration with ph_j.

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 [15]) modified to optionally use *tournament barriers* or *phasers* in lieu 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.

4.1 8-way Opteron SMP

All results in this subsection were obtained on a Quad-Core AMD Opteron Processor 8347 1.9 GHz SMP server with 4GB main memory running Fedora Linux release 8. The execution environment used for all Java runs is the IcedTea Runtime Environment (build 1.7.0-b21) with IcedTea 64-Bit Server VM (build 1.7.0-b21, mixed mode) and the "-Xms1000M -Xmx1000M" options. In addition, Intel's icc OpenMP compiler was used to measure the performance of the OpenMP version of BarrierBench.

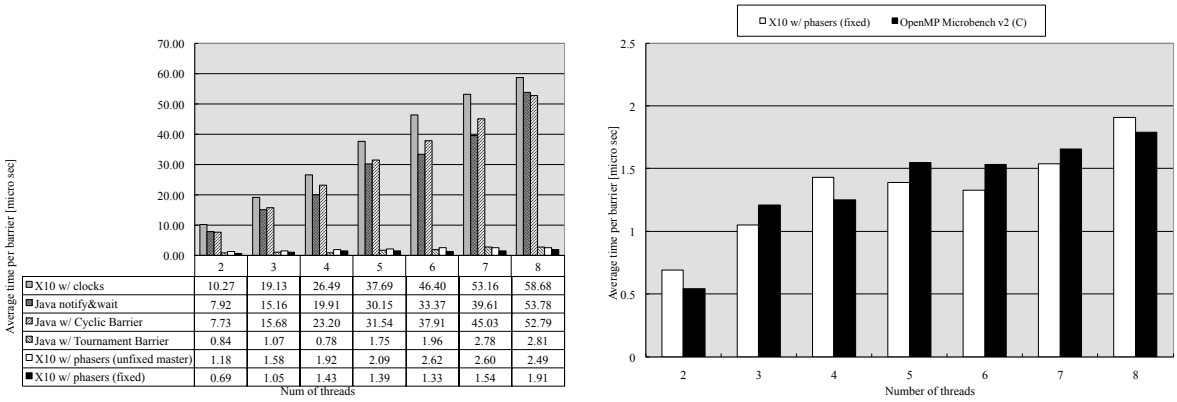
Figure 7 contains two charts that use the BarrierBench microbenchmark introduced in [14] to calibrate the performance of the different versions of barriers and phasers listed at the start of this section. No results are provided for the serial or 1-CPU case because the only "useful" work done by this microbenchmark is to coordinate synchronization among multiple threads. As shown in the top chart, the barrier overhead associated with phasers is significantly lower than that of X10's clocks (up to 30.7×), Java's notify-wait as implemented in the SimpleBarrier class in [9] (up to 28.2×), and JUC's cyclic barrier (up to 27.7×). In addition, the "fixed master" mode is a better choice than "unfixed master", since there is little load imbalance among the threads in these benchmarks. Finally, even though the gap with the TournamentBarrier version (as implemented in [9]) is small, phasers with "fixed master" was still 1.4× faster than TournamentBarrier for 8 threads.

The second chart in Figure 7 compares the performance of phasers with "fixed master" with that of OpenMP barriers using the Intel icc implementation, and shows that OpenMP was faster with 2, 4, and 8 threads and phasers were faster in the remaining configurations. The difference in performance is fairly small and it is nice to see that phasers can be competitive with the OpenMP native code implementation. Our initial experiments used gcc's OpenMP implementation and found that phasers were up to 30.4× faster. The large difference in performance underscores the importance of using a proprietary vendor implementation of OpenMP as the baseline for experimental results.

Figure 8 shows the performance comparison for the six other benchmarks listed in Table 3 in the form of speedup relative to the serial Java version. We see that the phaser runtime (with or without a fixed master) delivers the best average speedup, and is also remarkable in its consistency (unlike the TournamentBarrier which gave good speedups in many cases, but led to significant degradation for MG). It is worth noting that we obtained larger speedups (closer to 8 for 8 CPUs) when running these benchmarks with smaller data sizes than those used for Figure 8.

4.2 64-way Power5+ SMP

The results in this subsection were obtained on a p595+ 64-way Power5+ 2.3GHz SMP server with 512GB main memory running AIX5.3 TL5. All runs were performed with SMT turned off, and with a large page size of 256GB. The execution environment used for all Java runs was IBM's J9



	2	3	4	5	6	7	8
X10 w/ clocks	10.27	19.13	26.49	37.69	46.40	53.16	58.68
Java notify&wait	7.92	15.16	19.91	30.15	33.37	39.61	53.78
Java w/ Cyclic Barrier	7.73	15.68	23.20	31.54	37.91	45.03	52.79
Java w/ Tournament Barrier	0.84	1.07	0.78	1.75	1.96	2.78	2.81
X10 w/ phasers (unfixed master)	1.18	1.58	1.92	2.09	2.62	2.60	2.49
X10 w/ phasers (fixed)	0.69	1.05	1.43	1.39	1.33	1.54	1.91

Figure 7: BarrierBench microbenchmark results on 8-CPU AMD Opteron 8347 SMP

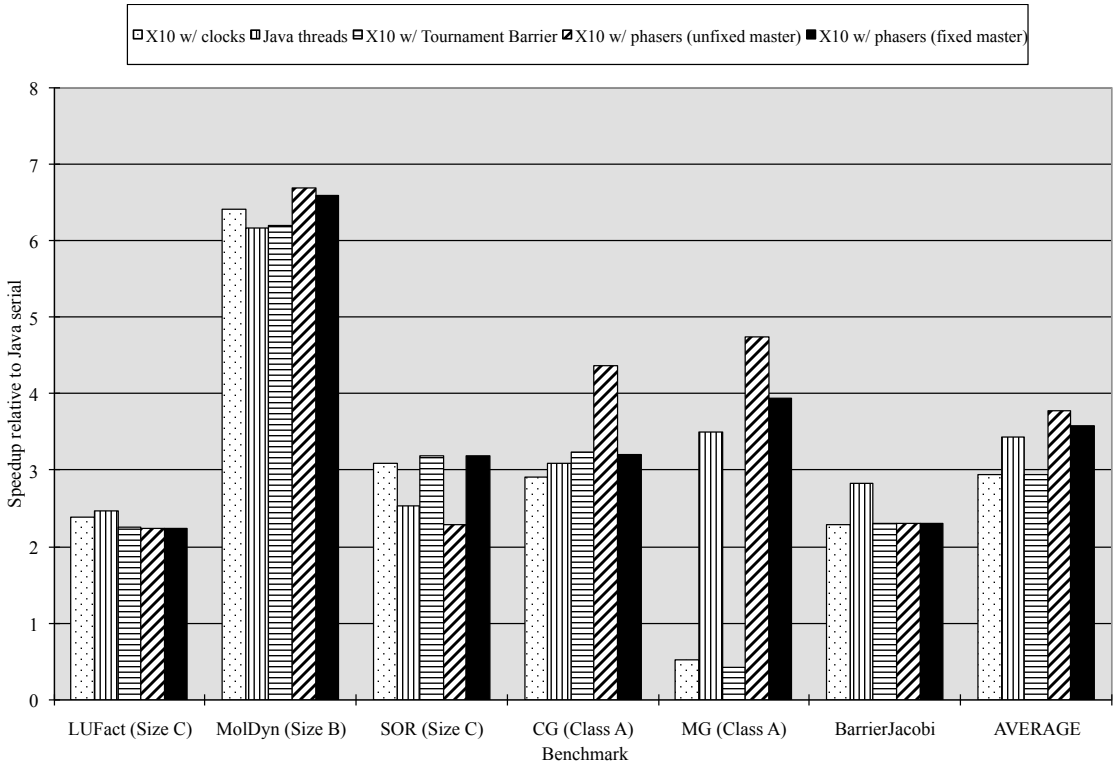
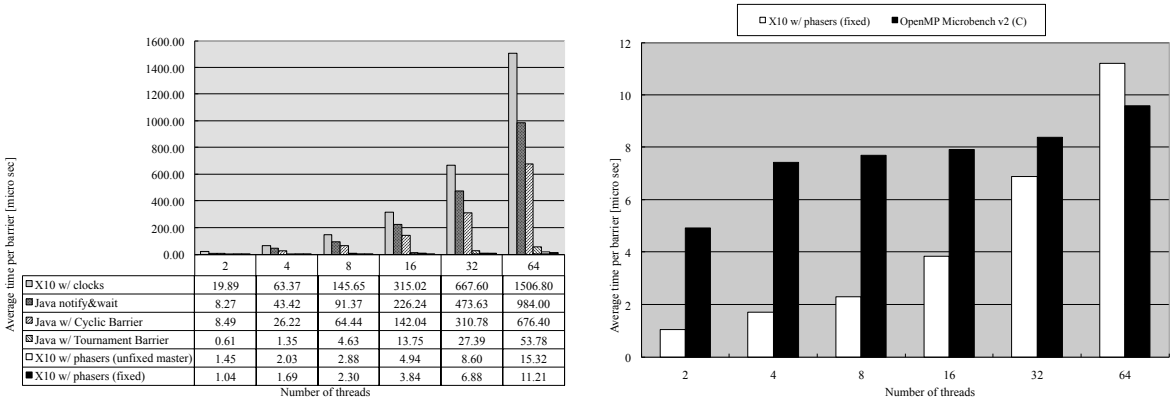


Figure 8: Speedup relative to serial Java version on 8-CPU AMD Opteron 8347 SMP



	2	4	8	16	32	64
X10 w/ clocks	19.89	63.37	145.65	315.02	667.60	1506.80
Java notify&wait	8.27	43.42	91.37	226.24	473.63	984.00
Java w/ Cyclic Barrier	8.49	26.22	64.44	142.04	310.78	676.40
Java w/ Tournament Barrier	0.61	1.35	4.63	13.75	27.39	53.78
X10 w/ phasers (unfixed master)	1.45	2.03	2.88	4.94	8.60	15.32
X10 w/ phasers (fixed)	1.04	1.69	2.30	3.84	6.88	11.21

Figure 9: BarrierBench microbenchmark results on 64-CPU Power5+ SMP

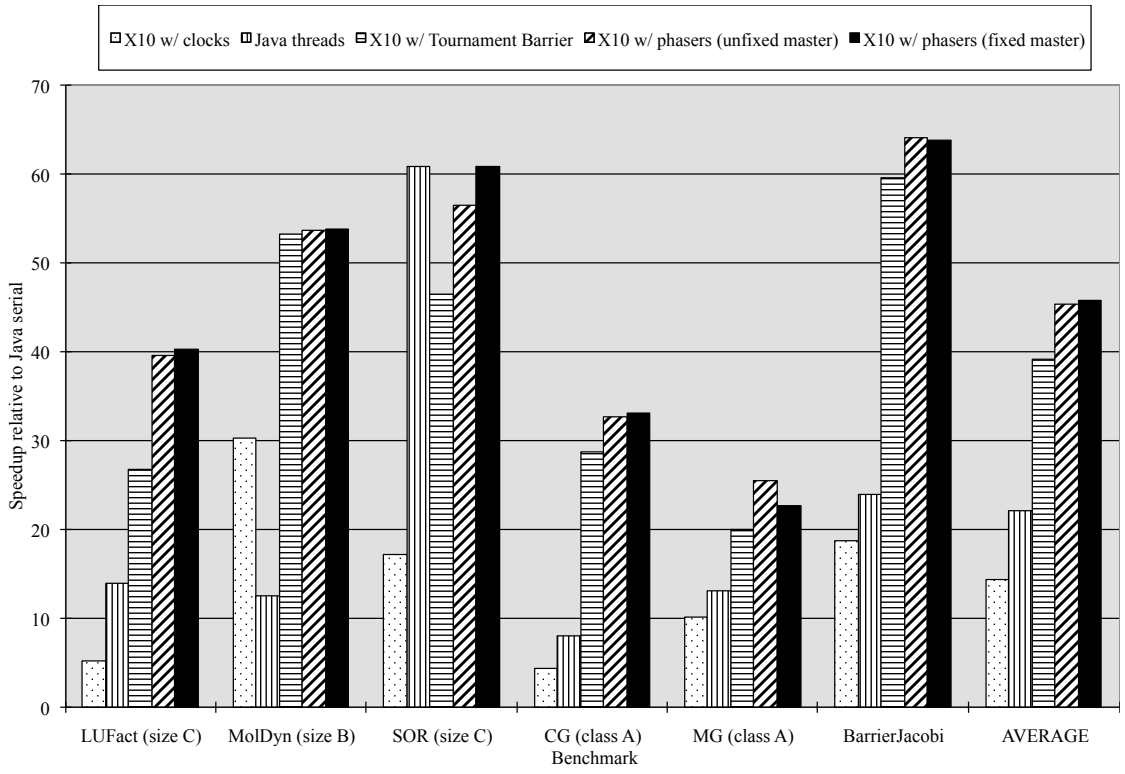


Figure 10: Speedup relative to serial Java version on 64-CPU Power5+ SMP

VM (build 2.4, J2RE 1.6.0) with the following options, "-Xjit:count=0,optLevel=veryHot, ignoreIEEE -Xms1000M -Xmx1000M". IBM's xlc OpenMP compiler (XL C/C++ Enterprise Edition V8.0 for AIX) was used to measure the performance of the OpenMP version of BarrierBench.

As shown in the top chart of Figure 9, the barrier overhead associated with phasers is significantly lower than that of X10's clocks (up to 134.5×), Java's notify-wait (up to 87.8×), and JUC's cyclic barrier (up to 60.4×). In addition, the gap with the TournamentBarrier version (as implemented in [9]) was 2× with 8 threads and 4.8× with 64 threads.

The second chart in Figure 9 compares the performance of phasers with "fixed master" with that of OpenMP barriers using the xlc implementation. It shows that the phaser implementation is still faster than OpenMP for 2, 4, 8, 16 and 32 threads, while a cross-over was observed at 64 threads.

Moving to Figure 10, we see that the phaser runtime (with or without a fixed master) obtains significantly better average speedup (45.7×) compared to parallel Java (22.0×) or current X10 (14.3×). The TournamentBarrier version yielded a speedup of 39.1×, which is closer to but still less than the speedup obtained by phasers. However, it's possible that some of the lessons learned from the TournamentBarrier implementation will become relevant to implementation of phasers on larger-scale SMPs in the future.

4.3 64-way UltraSPARC T2 SMP

All results in this subsection were obtained on a 64-way (8-core × 8 threads/core) 1.2 GHz UltraSPARC T2 (Niagara 2) with 32 GB main memory running Solaris 10. The execution environment used for all Java runs is the Java(TM) 2 Runtime Environment (build 1.5.0_12-b04) with Java HotSpot(TM) Server VM (build 1.5.0_12-b04, mixed mode) and the "-Xms1000M -Xmx1000M" options. Sun's C compiler 5.9 was used for the OpenMP version of BarrierBench.

The results for the barrier micro-benchmark are shown in Figure 11. The results for the standard benchmark suite are shown in Figure 12. Figure 11 shows the barrier overhead of phasers is significantly lower than X10's clocks (up to 85.36×), Java's notify-wait (up to 81.89×) and cyclic barrier (up to 75.76×). Another observable point is that TournamentBarrier shows better performance than phasers on the T2 processor, suggesting that it may have an advantage on processors with simultaneous multithreading.

Figure 12 also shows that the performance of phasers is better than others and almost equal to that of TournamentBarrier. Though the overhead of TournamentBarrier is smaller than that of phasers, the single statements in phasers reduce the number of barriers for LUFact, MolDyn, CG, MG and BarrierJacobi. Also, phasers provide efficient support for point-to-point synchronization in SOR.

5. RELATED WORK

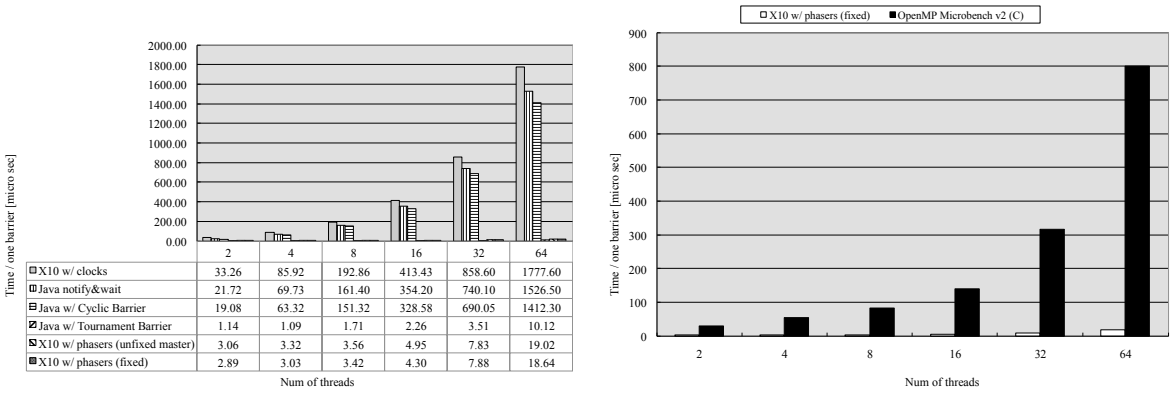
There is an extensive literature on barrier and one-way (point-to-point) synchronization. In this section, we focus on key comparable work. Like barriers, condition variables, and semaphores, phasers are used for coordinating the flow and execution of threads and processes; this is in contrast to mutual exclusion (locks) and transactional memories, which are more typically oriented towards preserving the consistency of data in (potentially) concurrent environments.

The JUC `CyclicBarrier` class [4] supports periodic barrier synchronization among a set of threads. Unlike Phasers, however, `CyclicBarrier`s do not support the dynamic addition or removal of threads; nor do they support one-way synchronization or split-phase operation.

OpenMP provides directives for barrier synchronization and single-thread execution regions [10]. The single and master directives are used to mark regions that should only be executed by a single thread. The main difference between the two is that the single region may be executed by any one thread in a thread group, but the master region is always executed by the master thread. The single construct presented in this paper serves a purpose similar to the OpenMP single directive, but we require that the single section be executed only after all threads have reached the single statement. The master directive in OpenMP is used to force only the master thread to execute a block of code in a parallel region, and there are no barriers on entry to or exit from the master region. We do not differentiate between single and master regions. Our implementation of phasers allows the master thread (i.e. the thread executing the single statement) to either be fixed for the life of the phaser or to change at each encounter of a single section. Also, in OpenMP, the master and single directives cannot be directly nested under work-sharing constructs (e.g. parallel for), but no such restriction exists for phasers. Perhaps the most important difference between OpenMP single directive and the phaser single statements is that threads can be dynamically added to the group of threads participating in a phaser with a single statement, but the number of threads participating in an OpenMP single section is fixed on entry to a parallel region.

Titanium is a dialect of Java for SPMD parallelism [8]. The language has a notion of single values which are the values used to ensure coherence of synchronization points. The designers want to statically ensure that the sequence of global synchronizations is identical across all processors so they make a conservative check to ensure that the statements with global effects is coherent across all processors. They define specific rules for constructing expressions that may have global effects to ensure that the coherence condition can be checked statically. Our phasers do not require all activities to reach the same synchronization point except in the case of next-with-single barriers. The runtime performs a dynamic check that all activities execute the same next-with-single, rather than the conservative static check used in Titanium.

Gupta's work on fuzzy barriers [5] introduces the concept of overlapping synchronization with "real" work in a model that is now widely known as a split-phase barrier. In this paradigm, a region of instructions is targeted for execution between the point when a processor enters the barrier and when is waits for all other processors to also enter. When processors enter the barrier synchronization region, they signal to the other processors that they are ready to synchronize. They may continue to execute any instructions in the synchronization region; but before a processor leaves the synchronization region, it must receive a notification from all processors indicating that they have entered the synchronization region. Unlike Gupta's work, in which compiler analysis uses low-level code motion to select a region of code for execution after signaling the barrier, the `signal`



	2	4	8	16	32	64
X10 w/ clocks	33.26	85.92	192.86	413.43	858.60	1777.60
Java notify&wait	21.72	69.73	161.40	354.20	740.10	1526.50
Java w/ Cyclic Barrier	19.08	63.32	151.32	328.58	690.05	1412.30
Java w/ Tournament Barrier	1.14	1.09	1.71	2.26	3.51	10.12
X10 w/ phasers (unfixed master)	3.06	3.32	3.56	4.95	7.83	19.02
X10 w/ phasers (fixed)	2.89	3.03	3.42	4.30	7.88	18.64

Figure 11: BarrierBench microbenchmark results on 64-way Sun UltraSPARC T2

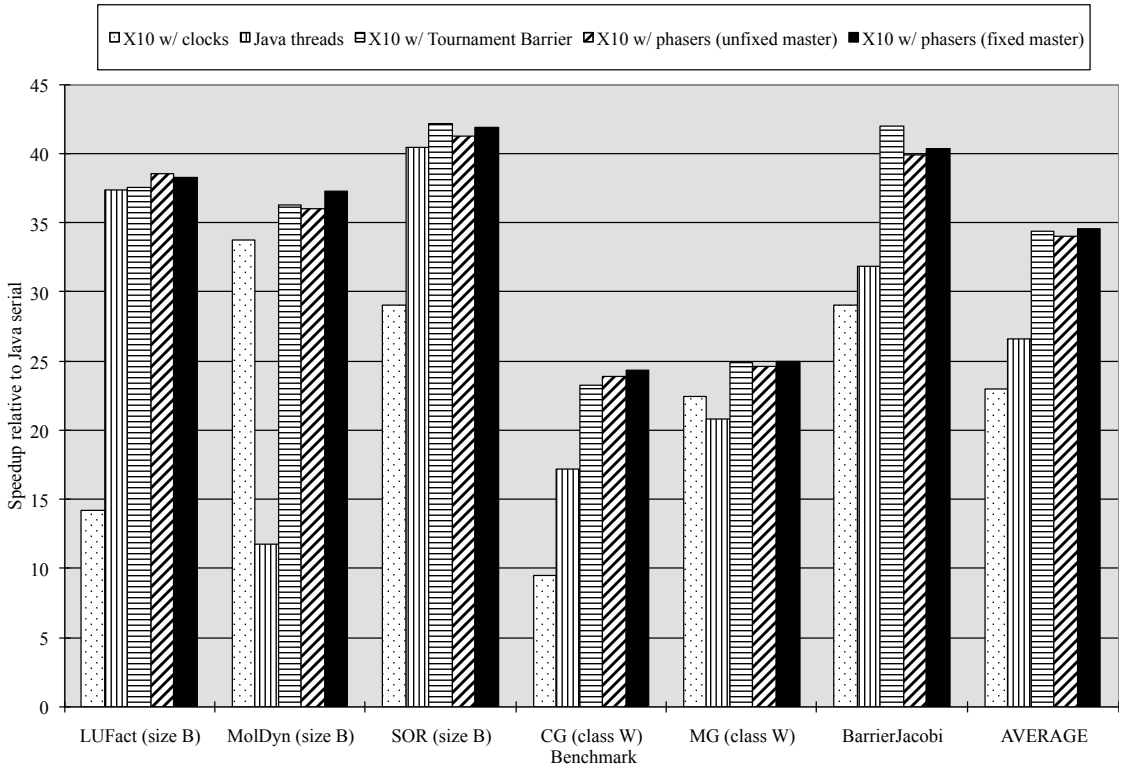


Figure 12: Speedup relative to serial Java version on 64-way Sun UltraSPARC T2

and `next` operations in Phasers allow explicit programmer control over when code is executed.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced *phasers*, a new coordination construct that unifies collective and point-to-point synchronizations. We established two safety properties for phasers: *deadlock-freedom* and *phase-ordering*. Performance results obtained from a portable implementation of phasers on two different SMP platforms demonstrate that they can deliver superior performance to existing barrier implementations, in addition to the productivity benefits that result from their generality and safety properties. Opportunities for future research related to phasers include extensions for reduction and collective operations, implementations on heterogeneous multicore processors (such as the Cell) and on distributed-memory clusters, and new compiler analyses and optimizations for phaser operations. We also believe that the ability for a thread to mark completion of multiple phases via a single `signal` statement could further reduce runtime overheads for certain workload scenarios.

Acknowledgments

We would like to thank Nathan Tallent for his feedback on this paper, and Rajkishore Barik for his contributions to early discussions related to phasers. 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, the IBM Poughkeepsie Benchmark Center for access to the 64-CPU Power5+ SMP system, and Doug Lea for access to the UltaSPARC T2 SMP system.

7. REFERENCES

[1] S. Agarwal, R. Barik, D. Bonachea, V. Sarkar, R. K. Shyamasundar, and K. Yelick. Deadlock-free scheduling of x10 computations with bounded resources. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 229–240, New York, NY, USA, 2007. ACM.

[2] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA 2005 Onward! Track*, 2005.

[3] S. Deitz. Parallel programming in chapel. http://www.cct.lsu.edu/ estrabd/LACSI2006/Programming2006.

[4] B. Goetz. *Java Concurrency In Practice*. Addison-Wesley, 2007.

[5] R. 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.

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

[7] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM Press.

[8] P. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, and K. Yelick. Titanium Language Reference Manual. Technical Report CSD-01-1163, University of California at Berkeley, Berkeley, Ca, USA, 2001.

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

[10] OpenMP specifications. http://www.openmp.org/blog/specifications/.

[11] C. F. J. C. K. R. D. Blumofe, C. E. Leiserson, K. H. Randall, and Y. Zhou. CILK: An efficient multithreaded runtime system. *Proceedings of Symposium on Principles and Practice of Parallel Programming (PPoPP'95)*, pages 207–216, July 1995.

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

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

[14] L. A. Smith and J. M. Bull. A multithreaded java grande benchmark suite. In *Proceedings of the Third Workshop on Java for High Performance Computing*, June 2001.

[15] 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.

[16] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. Welcome, and T. 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.