Hierarchical Phasers for Scalable Synchronization and Reductions in Dynamic Parallelism

Jun Shirako Vivek Sarkar
Department of Computer Science
Rice University
Houston, TX, USA
{*shirako, vsarkar*}*@rice.edu*

***Abstract*—The *phaser* construct is a unification of collective and point-to-point synchronization with dynamic parallelism. This construct gives each task the option of synchronizing on a phaser in *signal-only/wait-only* mode for producer/consumer synchronization or *signal-wait* mode for barrier synchronization. A *phaser accumulator* is a reduction construct that works with phasers in a phased setting. Phasers and accumulators support *dynamic parallelism i.e.,* they allow dynamic addition and removal of tasks from the synchronizations and reductions that they support.**

Past implementations of phasers and phaser accumulators have used a *single master* task to advance a phaser to the next phase and to perform computations for lazy reductions, while also supporting dynamic parallelism. Though the single master approach provides an effective solution for modest levels of parallelism, it quickly becomes a scalability bottleneck as the number of threads increases. To address this limitation, we propose an approach based on *hierarchical phasers* for scalable synchronization and *hierarchical accumulators* for scalable reduction. Our approach also includes tunable initialization parameters that specify the *degree* and *number of tiers* for the phaser hierarchy, thereby allowing different values to be chosen for different platforms. Our performance results show significant scalability benefits from our approach. To the best of our knowledge, this is the first approach to support hierarchical synchronization and reductions in the presence of dynamic parallelism.

***Keywords*-Phasers; barrier synchronization; point-to-point synchronization; reductions; dynamic parallelism.**

I. Introduction

The computer industry is at a major crossroads. Instead of using processors with faster clock speeds, all computers— embedded, mainstream, and high-end — are being built using chips with an increasing number of processor cores, with little or no increase in clock speed per core. This trend poses a tremendous challenge for software enablement on future systems as the number of cores per socket continues to grow, and the cores become more heterogeneous. To address this trend, multiple programming models are emerging to address an increased need for *dynamic task parallelism* in multicore shared-memory multiprocessors. Examples include Chapel [1], Cilk [2], Fortress [3], Intel Threading Building Blocks [4], Java Concurrency Utilities [5], Microsoft Task Parallel Library [6], OpenMP 3.0 [7], and X10 [8]. All these models identified dynamic lightweight task parallelism as one of the prerequisites for success. In dynamic task parallelism, computations are dynamically created as *tasks* and the runtime scheduler is responsible for scheduling and synchronizing the tasks across the cores.

The Habanero Java (HJ) language under development at Rice University [9] proposes an execution model for multicore processors that builds on four orthogonal constructs:

1) Lightweight *dynamic task creation and termination* using *async* and *finish constructs* [10].
2) *Locality control* with task and data distributions using the *place* construct [11]. Places enable colocation of async tasks and data objects.
3) Mutual exclusion and isolation among tasks using the *isolated* construct [12].
4) Collective and point-to-point synchronization using the *phasers* construct [13] along with their accompanying *accumulators* [14].

The HJ language derives from initial versions of the X10 language (up to v1.5) that used Java as the underlying sequential language [8]. In contrast, more recent versions of X10 starting with v1.7 have moved to a Scala-like syntax [15]. Since HJ is based on Java, the use of certain primitives from the Java Concurrency Utilities [16] is also permitted in HJ programs.

This paper focuses on enhancements and extensions to the phaser construct, which is an extension of the clock construct from X10 [8]. There is a serious proposal in progress to add a subset of the phaser capability to Java 7 libraries [17, 18], influenced by our past work on phasers [13, 14]. The "tiering" functionality in the proposed java.util.concurrent Phaser class was also influenced in part by the work reported in this paper [19]. Our hope is that this paper will

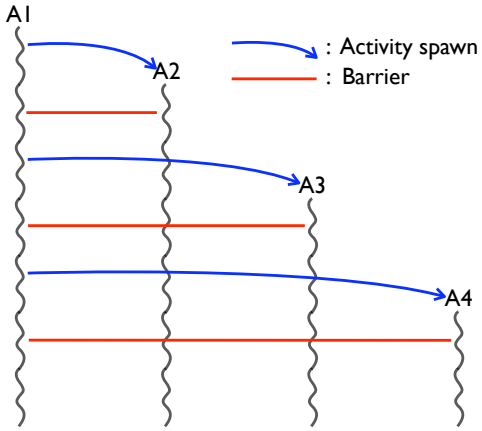


Figure 1: Barrier synchronization with dynamic parallelism

influence Java and other multicore software platforms to provide scalable support for synchronization and reduction operations with dynamic parallelism using phasers in their full generality.

Past implementations of phasers and phaser accumulators have used a *single master* task to advance a phaser to the next phase and to perform computations for "lazy" reductions. While the single master approach provides an effective solution for modest levels of parallelism, it quickly becomes a scalability bottleneck as the number of threads increases. To address this limitation, we propose an approach based on *hierarchical phasers* for scalable synchronization and *hierarchical accumulators* for scalable reduction. For simplicity, the programming constructs used are unchanged between flat phasers and hierarchical phasers, except for tunable initialization parameters that specify the degree and number of tiers for the phaser hierarchy. As discussed later in the paper, these parameters can have a significant impact on performance. To the best of our knowledge, this is the first approach to support hierarchical synchronization and reductions in the presence of dynamic parallelism.

Figure 1 uses a simple barrier example to illustrate some of the challenges involved in supporting hierarchical synchronization in the presence of dynamic parallelism. In this figure, activity $A1$ creates activity $A2$, synchronizes with $A2$ on a barrier, then creates activity $A3$, synchronizes with both $A2$ and $A3$ on a barrier, and so on. Though not shown in the figure, our model permits activities to leave a barrier (phaser) while others continue to synchronize on it, and also permits some activities to be registered in "signal-only" or "wait-only" modes. There are many reasons why this form of dynamic parallelism can be important for multicore processors. For example, adaptive algorithms may choose to create tasks dynamically based on input data; further, this functionality can also be useful in supporting speculative parallelism. Classical hierarchical approaches to barrier synchronization, such as *tournament barriers* [20] follow a static synchronization structure, and are unable to handle the dynamic parallelism addressed by our approach.

The rest of the paper is organized as follows. Section II provides background on phasers, accumulators, and the single master approach to their implementation. Section III discusses possible approaches for the programmer to build hierarchical phasers and accumulators explicitly by hand, analogous to hand-coded tournament barriers, and the limitations of the explicit approach. Sections IV and V describe our approach to automatic implementation of hierarchical phasers and hierarchical accumulators respectively. Section VI presents our experimental results, and Section VIII contains our conclusions.

II. Background

A. Phasers

In this section, we summarize the *phaser* construct introduced in [13]. 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 [21]. 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 [22, 23], counting semaphores [24], and X10's clocks [25, 26].

Before describing phasers, we briefly recapitulate the *async* and *finish* constructs for activity creation and termination. Though phasers as described in this paper may appear specific to async-finish task parallelism in X10 and HJ, 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.

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

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

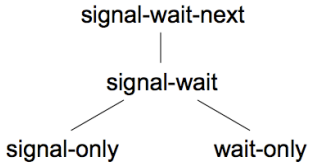


Figure 2: Capability lattice for phasers

next statement. The statement *foreach (point p : R)* ⟨*stmt*⟩, which is equivalent to *for (point p : R) async* ⟨*stmt*⟩, creates new child activities as iterations of a parallel loop. As discussed in [27], these iterations can often be chunked together for efficiency, even in cases when they are all synchronized on the same phaser.

The 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). 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 dynamic instruction has a unique *Immediately Enclosing Finish* (IEF) dynamic statement instance. In the async-finish computation DAG introduced in [28], a *dependence edge* is introduced from the last instruction of an activity to the *end-finish* node corresponding to the activity's IEF.

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 2. 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*. At this point, A_i is the only activity registered on ph.
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 support the "`async phased` A_j" syntax to indicate by default that A_i is transmitting all its capabilities on all phasers that it is registered with to A_j.
3) **drop:** 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. In addition, A_i drops its registration on all phasers when it terminates.
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 [13], the semantics of `next` depends on the registration mode that A_i has with each phaser that it is registered on.

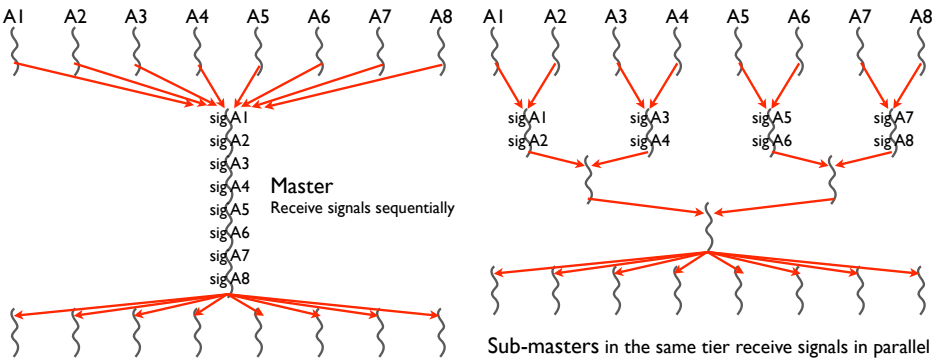
B. Accumulators

A *phaser accumulator* is a 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 that extends the overlap in fuzzy [22] or split-phase [29] barriers.

1) **new:** When A_i performs a `new accumulator(ph, op, dataType)` operation, it results in the creation of a new accumulator, a. `ph` is the host phaser with which the accumulator will be associated, `op` is the reduction operation that the accumulator will perform, and `dataType` is the numerical type of the data upon which the accumulator operates.
2) **send:** An `a.send()` operation performed by A_i sends a value for accumulation in accumulator `a` in the current phase. If an activity performs multiple `send()` operations on the same accumulator in the same phase, they are treated as separate contributions to the reduction.
3) **result:** The `a.result()` operation performed by A_i receives the accumulated value in accumulator `a` from the *previous* phase. Thus, the barrier synchronization provided by phasers provides an ideal foundation for reductions and there is no data race between `send()` and `result()` operations.

C. Scalability Bottleneck in Single-Level Phasers and Accumulators

As shown in Figure 3a, a single-level phaser barrier synchronization is divided into two operations, gather and broadcast. In the gather operation, a master activity waits for all signals from worker activities sequentially, and the waiting time can be proportional to the



(a) Single-level phaser (b) Hierarchical phaser

Figure 3: Single-level phaser with single master vs. Hierarchical phaser with sub-masters

number of workers. On the other hand, the broadcast operation is more scalable because each worker just waits for a broadcast signal from the master. Thus, gather operations are the major scalability bottleneck in phaser and accumulator operations.

III. Explicit Phaser Trees and their Limitations

One approach to addressing the scalability bottleneck in single-level phasers is to build hierarchical phasers and accumulators explicitly by hand, analogous to hand-coded tournament barriers. Figure 3b shows a tree-based hierarchical barrier synchronization that employs multiple sub-masters in the gather operation. Tree-based barriers have the advantage that gather and reduction operations in the same level (tier) can be executed in parallel by sub-masters. Also, in cases when the hierarchy of sub-masters follows the natural hierarchy in the hardware, each sub-master will naturally leverage data locality among workers in its sub-group.

However, explicit implementation of hierarchical phasers and accumulators comes with a high level of programming complexity. As an example, Figures 4 illustrates the difference between the single-level and hierarchical versions of the JGF BarrierBench microbenchmarks [30]. We have also constructed single-level and hierarchical versions of the JGF ReduceBench benchmark. Another issue with the hand-coded approach is the tree topology is hardwired in the code, making it hard to adapt to different architectures and to dynamic parallelism. Thus, our goal is to retain the simplicity of the single-level codes for phasers and accumulators, while delivering the scalable performance of the hierarchical barrier synchronizations. As discussed in Section VI, our automatic approach in fact achieves superior performance to the hand-coded approach.

```
Single-level phaser example:
----------------------------
finish {
  phaser ph = new phaser(SIG_WAIT);
  foreach (point [thd] : [0:nthreads-1])
   phased (ph<SIG_WAIT>) {
    for (int s = 0; s < sz; s++) {
      delay(delaylength);
      next;
    }
  }
}

Equivalent explicit multi-level phaser tree:
--------------------------------------------
finish {
  phaser ph = new phaser(SIG_WAIT);
  foreach (point [p] : [0:nSubPh-1])
   phased (ph<SIG_WAIT>) {
    phaser iph = new phaser(SIG_WAIT);
    foreach (point [q] : [1:nLeave-1])
     phased(ph<WAIT>, iph<SIG>) {
      for (int s = 0; s < sz; s++) {
        delay(delaylength);
        iph.signal();  ph.doWait();
    } }
    for (int s = 0; s < sz; s++) {
      delay(delaylength);
      iph.signal();  iph.doWait();
      ph.signal();  ph.doWait();
} } }
```

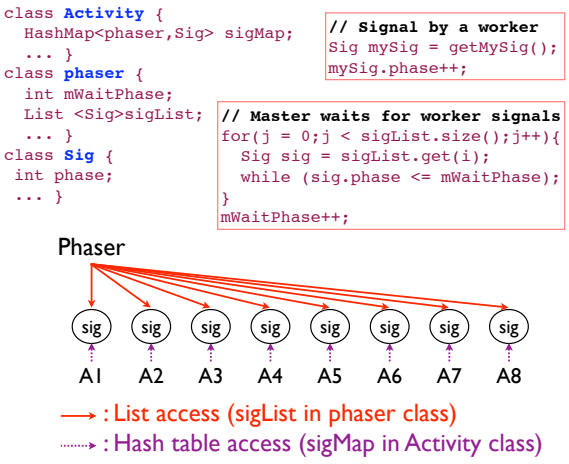
Figure 4: Single-level phaser and equivalent explicit multi-level phaser tree for JGFBarrierBench

IV. Hierarchical Phasers

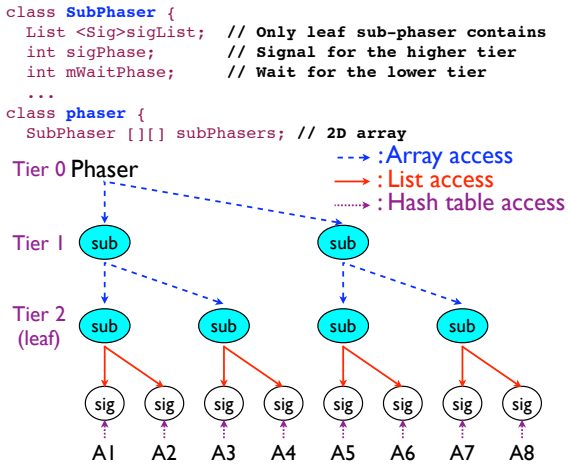
This section introduces the programming interface and implementation for hierarchical phasers. As with flat phasers, hierarchical phasers support dynamic parallelism so as to allow the set of activities synchronized on a phaser to vary dynamically.

```
finish {
  phaser ph = new phaser(SIG_WAIT,
                         numTiers, numDegree);
  foreach (point [thd] : [0:nthreads-1])
    phased (ph<SIG_WAIT>) {
      for (int s = 0; s < sz; s++) {
          delay(delaylength);
          next;
} } }
```

Figure 5: Multi-level phaser tree with hierarchical phaser extension



(a) Single-level phaser



(b) Tree-based phaser

Figure 6: Data structure for phaser synchronization

A. Programming Interface

In our approach, the programming constructs are unchanged between flat phasers and hierarchical phasers, except for tunable initialization parameters that specify the degree and number of tiers for the phaser hierarchy. Specifically, two additional parameters, numTiers and numDegree, can now be specified in the phaser constructor as shown in Figure 5.

The `numTiers` parameter ($= T$) specifies the number of tiers to be used by the runtime system's tree-based sub-phaser structure, and `numDegree` ($= D$) is the maximum number of child sub-phasers that can be created on a parent sub-phaser. A hierarchical phaser with `numTiers=1` is equivalent to a single-level phaser. The leaf of a sub-phaser tree has no child sub-phasers, and deals with the activities that are assigned to the leaf. Although there is no limit on the number of activities registered on a phaser, the runtime may run into some scalability bottlenecks if the number exceeds T^D, since that implies that the synchronizations and reductions will need to be serialized within "sub-masters" at the leaves of the phaser tree.

B. Runtime Implementation

Figure 6a shows pseudo codes and data structures for the gather operation of the single-level barrier. Each activity registered on a phaser has a `Sig` object corresponding to the phaser. `Sig` objects are contained in `HashMap sigMap` of `Activity` class so that an activity can be registered and synchronized on multiple phasers. These `Sig` objects are also included in `List sigList` of `phaser` class so that a master activity, which is dynamically selected from activities on a phaser and advances the phaser, can access them. All activities registered on the phaser send a signal to the master activity by incrementing its `Sig.phase` counter, and the master activity waits for all `Sig.phase` counters to be incremented by busy-wait loop. A registered activity can also spawn another activity, or child activity, and register the child on the phaser. a new `Sig` object corresponding to the child activity is added to `sigList`, and the child activity attends to synchronizations on the phaser. Busy-wait loops in phaser runtime have timeout periods that can be specified as a runtime parameter. When a busy-wait loop times out, the activity sleeps and the hardware thread switches to another activity with Java runtime support (`java.lang.Object.wait/notify`).

Figure 6b shows data structures for the gather operation of the tree-based barrier. `SubPhaser` class contains `List sigList`, `sigPhase` and `mWaitPhase` counters. The `sigList` of a leaf sub-phaser includes `Sig` objects for activities that are assigned to the leaf sub-phaser. `phaser` class has a two dimensional `SubPhaser` array and all activities can access the hierarchical sub-phasers so that any eligible activity can be a master activity to advance the sub-phaser. In the gather operation, all sub-masters on leaf sub-phasers check their `sigList` and wait

```
finish {
  phaser ph = new phaser(SINGLE,
                         numTiers, numDegree);
  accumulator a = new accumulator(SUM,
                                  int.class, ph);
  foreach (point [thd] : [0:nthreads-1])
    phased (ph<SINGLE>) {
      for (int s = 0; s < sz; s++) {
          delay(delaylength);
          a.send(1);
          next single {
              // Barrier w/ Single stmt
              // Master gets reduction result
              sum = a.result();
} } } }
```

Figure 7: Multi-level phaser tree with hierarchical phaser accumulator extension

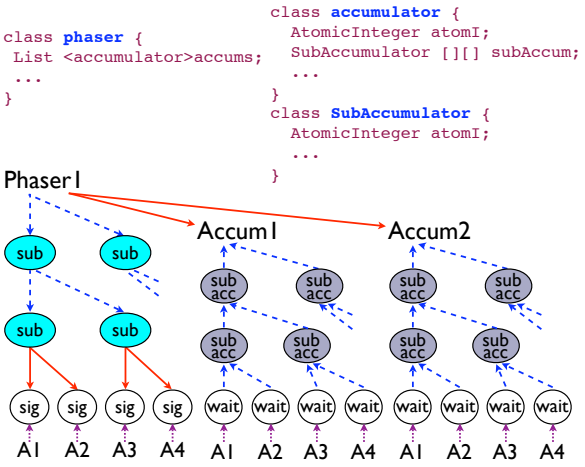


Figure 8: Data structure for tree-based phaser accumulator

for the signals from other activities in parallel, and increment their `sigPhase` counters after waiting the signals. A sub-master on non-leaf sub-phaser waits for the `sigPhase` increments of its child sub-phasers and also increments its `sigPhase`. Finally, the global master receives the signal from the top level sub-phasers and finishes the hierarchical gather operation.

When an activity spawns a child activity, the child is registered on the same leaf sub-phaser as its parent activity until the number of activities on the leaf reaches `numDegree`. If the leaf is full, the child activity is registered on another leaf sub-phaser. This process continues so long as the total number of levels does not exceed `numTiers`. Since this process needs additional atomic accesses, the initialization (registration) overhead of hierarchical phasers is generally larger than flat phasers.

V. Hierarchical Accumulators

A. Programming Interface

Accumulator objects are associated with a phaser object when they are allocated. If the phaser has a hierarchical structure, the accumulators inherit the same tree structure with the same `numTiers` and `numDegree` parameters. Therefore, there is no change in the accumulator interface as shown in Figure 7.

B. Runtime Implementation

Figure 8 shows data structures for the tree-based gather operation with accumulator extensions. A phaser object can support multiple accumulators (as shown in the `List accums` field); the figure shows two accumulators (`Accum1` and `Accum2`) associated with a single phaser (`Phaser1`). The accumulator implementation in phasers employs java.util.concurrent atomic objects like AtomicInteger — both the `accumulator` and `SubAccumulator` classes include an atomic variable. Sub-accumulators and sub-phasers use isomorphic tree structures. Each activity sends a value to the leaf sub-accumulator, and local accumulations on the leaves are performed individually. Also, a sub-master sends the local accumulation result to its parent sub-accumulator. Finally, all values are integrated into the global atomic variable on the accumulator object.

VI. Experimental Results

In this section, we present experimental results for the hierarchical phaser and accumulator implementations developed at Rice University for the Habanero Java language [9].

A. Experimental Setup

We obtained results for the EPCC Syncbench microbenchmark measuring barrier and reduction overheads with the following implementation variants.

1) **OpenMP** is the OpenMP implementation for barrier synchronization and reduction. Figures 9a, 9b, and 9c show code fragments for the OpenMP SyncBench microbenchmarks developed at EPCC [31] for barrier, for, and reduce computations respectively. For the OpenMP implementation studied in this paper, the barrier and for variants in Figures 9a and 9b showed comparable performance so we only report results for Figures 9a and 9c in this section. We also created equivalent HJ versions of Figures 9a and 9c for the phaser measurements outlined below. We set `innerreps=10,000` for both the OpenMP and HJ versions so as to focus on the barrier overhead, and minimize the impact of

```
a) OpenMP barrier directive
-------------------------------
#pragma omp parallel private(j)
{
    for (j=0; j<innerreps; j++){
        delay(delaylength);
        #pragma omp barrier
} }

b) OpenMP for directive
-------------------------------
#pragma omp parallel private(j)
{
    for (j=0; j<innerreps; j++){
        #pragma omp for
        for (i=0; i<nthreads; i++)
            delay(delaylength);
} }

c) OpenMP for reduction directive
-------------------------------
#pragma omp parallel private(j)
{
    for (j=0; j<innerreps; j++){
        #pragma omp for reduction(+:a)
        for (i=0; i<nthreads; i++){
            delay(delaylength);
            a += 1;
} } }
```

Figure 9: EPCC Syncbench codes for the experiments

# threads	(numTiers, numDegree)
2	(1, 1)
4	(2, 2)
8	(2, 4)
16	(2, 8)
32, 64, 128	(2, 16)

Table I: `numTiers` and `numDegree` for experiments

task creation overhead on the results reported in Sections VI-B – VI-D.

2) **Phaser normal** is the single-level phaser and accumulator implementation.
3) **Phaser tree hand** is the explicit tree-based phaser and accumulator version implemented by hand.
4) **Phaser tree auto** is the hierarchical phaser and accumulator version with the Habanero runtime support introduced in this paper.

For Sections VI-B, VI-C and VI-E, we used the tree parameters shown in Table I. Parameter (`numTiers = 1`, `numDegree = 1`) is equivalent to flat-level phaser, although it has additional overhead due to tree-based implementations. Section VI-D studies the impact of selecting alternate values for these parameters.

For all runs, the main program was extended with a 30-iteration loop within the same Java process, and the best of the 30 times was reported in each case. This configuration was deliberately chosen to reduce/eliminate the impact of JIT compilation time in the performance comparisons [32].

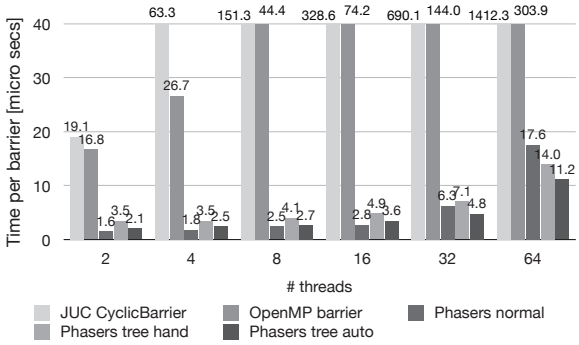


Figure 10: Barrier performance with Syncbench (64-thread Niagara 2)

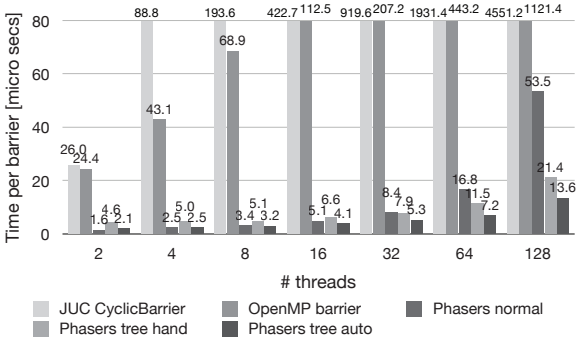


Figure 11: Barrier performance with Syncbench (128-thread Niagara 2)

We use the Habanero-Java (HJ) compiler and runtime [9] which are based on IBM X10 version 1.5 [33] for the performance experiments. All HJ runs were performed with the following options: `-NUMBER_OF_LOCAL_PLACES=1 -PRELOAD_CLASSES=true -BIND_THREADS=true -INIT_THREADS_PER_PLACE=$nthreads`. Here, `nthreads` is the number of threads for which the measurement was being performed.

All results in this paper were obtained on two platforms. The first is a 64-thread (8 cores $\times$ 8 threads/core) 1.2 GHz UltraSPARC T2 (Niagara 2) with 32 GB main memory running Solaris 10. We conducted all HJ 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). Sun's C compiler v5.9 with the compile options "-fast -xopenmp -lm" was used for the OpenMP evaluations[2]. The second is a 128-thread (16 cores $\times$ 8 threads/core) dual-chip UltraSPARC T2 SMP with 32 GB main memory

[2] `-fast` represents the highest optimization level option for the compiler.

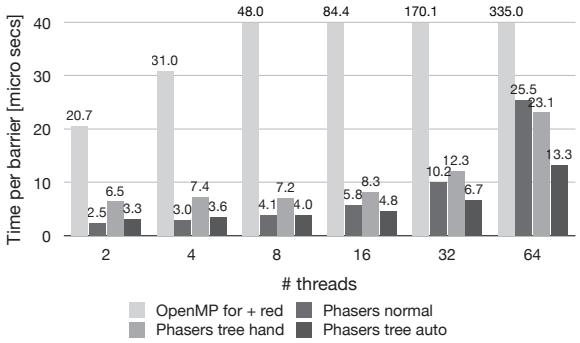


Figure 12: Barrier and reduction performance with Syncbench (64-thread Niagara 2)

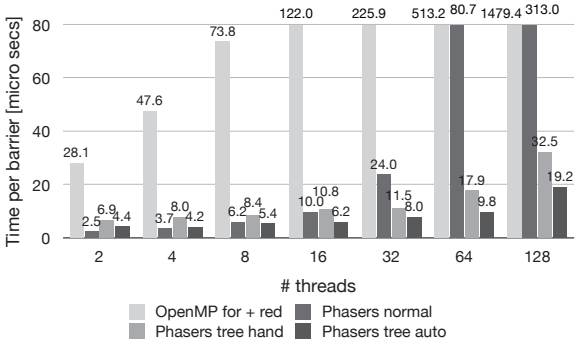


Figure 13: Barrier and reduction performance with Syncbench (128-thread Niagara 2)

running Solaris 10 and the same environments as the 64-thread Niagara 2.

B. Barrier Microbenchmark

This section presents barrier synchronization performance using the EPCC Syncbench barrier microbenchmark outlined in Figure 9a.

Figure 10 compares the barrier performance of the three variants of phasers summarized in Section VI-A with Java's CyclicBarrier and OpenMP's barrier operation (Figure 9a) on 64-thread UltraSPARC T2 SMP. Although the barrier overhead for OpenMP `barrier` and CyclicBarrier is quite large on this platform, all phaser constructs show much better performance with any number of threads. While single-level phaser implementation shows the best barrier performance up to 16 threads, the tree-based phaser is faster when 32 and 64 threads are used. Hierarchical phasers supported by the HJ runtime always perform better than explicit phaser trees. This is partly because the hierarchical sub-phaser structures the HJ runtime are implemented using dynamically sized 2D arrays, while hand-coded phaser trees need to use more complicated list structures due to API constraints. The 64-thread barrier overhead for hierarchical phasers with runtime support is 126.6× smaller than CyclicBarrier, 27.2× smaller than OpenMP `barrier`, 1.58× smaller than single-level phasers and 1.25× smaller than hand-coded phaser trees.

Figure 11 shows the barrier overhead on 128-thread UltraSPARC T2 SMP. Hierarchical phaser's barrier overhead using all 128 threads is 335.1× smaller than CyclicBarrier, 89.2× smaller than OpenMP `barrier`, 3.94× smaller than single-level and 1.57× smaller than hand-coded phaser trees.

C. Reduction Microbenchmark

This section presents barrier synchronization and reduction performance using the EPCC Syncbench reduction microbenchmark outlined in Figure 9c.

Figure 12 shows the barrier and reduction overhead on a 64-thread UltraSPARC T2 for an OpenMP `reduction`, and for a phaser with an accumulator as described in Section V. The single-level phaser is the fastest implementation when the number of threads is two and four. However, the hierarchical phaser with runtime support shows better performance than the single-level phaser with more than four threads. As before, the hierarchical phaser with runtime support always performed better than explicit phaser trees. The barrier and reduction overhead of a hierarchical phaser with runtime support is 25.2× smaller than an OpenMP `reduction`, 1.92× smaller than a single-level phaser and 1.74× smaller than a hand-coded phaser tree when all 64 threads are used.

Figure 13 shows the barrier and reduction overhead on a 128-thread UltraSPARC T2. The single-level phaser with 64 and 128 threads exhibits significant overhead due to contention on a single atomic variable. On the other hand, the overhead of a tree-based phaser is comparable to the barrier overhead shown in Figure 11 because the hierarchical accumulations on sub-accumulators distribute the contention and reduce hot spots. The 128-thread barrier and reduction overhead for a hierarchical phaser with runtime support is 77.2× smaller than OpenMP, 16.3× smaller than single-level phaser and 1.69× smaller than hand-coded phaser trees. Thus, the benefit of using hierarchical phasers increases as we move from 64 to 128 threads.

D. Impact of Number of Tiers and Degree

This section studies the performance impact of changing the `numTiers` and `numDegree` parameters in-

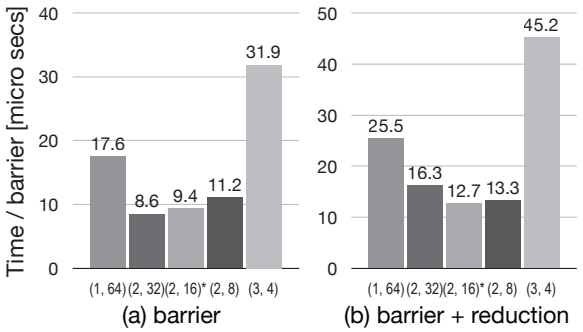


Figure 14: Performance with different # degree and tiers on 64-thread Niagara T2

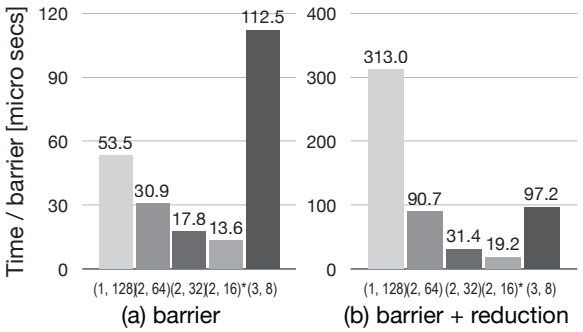


Figure 15: Performance with different # degree and tiers on 128-thread Niagara T2

troduced in Section IV. Figures 14 and 15 show the barrier and barrier+reduction overhead of hierarchical phasers and accumulators with runtime support on a 64-thread and 128-thread UltraSPARC T2 SMPs.

Figure 14a shows that the combination of (`numTiers = 2, numDegree = 32`) gives the best barrier performance on 64-thread UltraSPARC T2, while (`numTiers = 2, numDegree = 16`) is close behind. However, (`numTiers = 2, numDegree = 16`) gives the best performance in Figures 14b (64 threads), 15a (128 threads) and 15b (128 threads). This was the reason why (`numTiers = 2, numDegree = 16`) was selected as the default value for the results reported in Sections VI-B, VI-C and VI-E. The results for `numTiers = 3` suggest that it is too large a value to use for platforms with 64 or 128 threads. Finally, we observe that it may have been natural to expect `numDegree = 8` to yield better performance than `numDegree = 16` since the UltraSPARC T2 processor contains 8 cores with 8-way multithreading per core. The fact that `numDegree = 16` always delivered better performance reinforces the importance of empirically tuning these parameters.

```
a) OpenMP barrier increasing # threads
-------------------------------
  for (nth = 2; nth <= 64; nth++) {
#pragma omp parallel for num_threads (nth)
      for (i=0; i<nth; i++){
          delay(delaylength);
      }
  }

b) Phaser barrier increasing # threads
-------------------------------
  finish {
      final phaser ph = new phaser(SIG_WAIT);
      for (int nth = 2; nth <= 64; nth++) {
          async phased (ph<SIG_WAIT>) {
              for (int n = nth; n <= 64; n++) {
                  next;
                  delay(delaylength);
              }
          }

          next;
          delay(delaylength);
      }
  }
```

Figure 16: EPCC Syncbench extended for barrier with dynamic parallelism

E. Barrier with Dynamic Parallelism

This section presents barrier synchronization performance in the presence of dynamic parallelism. To focus on the barrier and dynamic task creation overhead, we extend the EPCC Syncbench microbenchmark to support dynamic parallelism as shown in Figure 16. These code examples mirror the dynamic parallelism structure shown earlier in Figure 1 (Section I.). As shown in Figure 16a, the number of threads cannot be changed within a OpenMP parallel region. The outermost loop index `nth` represents the number of parallel threads, and `parallel for` with `num_threads` clause forks threads, executes tasks in parallel, processes a barrier synchronization and join the threads at each iteration. Figure 16b shows the Habanero-Java version of the extended Syncbench code. The outermost loop also iterates from 2 to 64, and the parent activity that performs the `new phaser(SIG_WAIT)` operation executes the outermost loop. Since Habanero-Java supports barrier operations with dynamic parallelism, the parent activity spawns only one child activity and then synchronizes with its children on a barrier. The number of barriers performed by each child activity varies from 63 to 1, and supports the same shape of parallelism as OpenMP but with lower thread fork/join overhead.

Table II summarizes the results obtained for the OpenMP and HJ versions of this microbenchmark for dynamic parallelism on the 64-thread Niagara-2 system. A key point to note in this test case is that there is no `innerreps` parameter that amortizes

Benchmark version	Time / Barrier
OpenMP version (Figure 16a)	220.2 μs
HJ version, Phaser tree auto (Figure 16b)	23.8 μs

Table II: Barrier performance for Dynamic parallelism (64-thread Niagara-2)

the overhead of task creation over multiple barrier operations. As a result, task creation overhead will be a non-trivial contributor to the average time for a barrier operation measured for this test case. However, this will usually be the case when dynamic parallelism is used in conjunction with barriers/phasers. The results obtained in Table II show that the hierarchical implementation of phasers introduced in this paper again performs significantly better than the OpenMP version.

F. Application Benchmarks

In this section, we present performance results for three Java Grande Forum Benchmarks (LUFact, SOR and MolDyn) [30], and two NAS Parallel Benchmarks (CG and MG) [34]. Figure 17 shows speedup for CyclicBarrier, single-level phasers and hierarchical phasers with the largest data size on the 128-thread Niagara-2. The impact of hierarchical phasers will of course depend on the extent to which synchronization poses a significant overhead for this combination of benchmark and hardware. The improvement due to the hierarchical implementation is 1.13× for LUFact, 1.16× for SOR and 1.05× for CG. MolDyn and MG have enough large granularity of parallelism for synchronization overhead to not be a significant contributor to performance. On average (geometric mean), the hierarchical phaser is 4.0× faster than the CyclicBarrier and 1.06× faster than a single-level phaser.

Table III shows the geometric mean speedup (relative to serial execution) for all five benchmarks on the 64 and 128 thread Niagara-2 systems, for the three data sizes listed in Table IV. On the 64-thread Niagara-2, the benefit of hierarchical phasers relative to flat phasers is 1.08× for small data size, 1.02× for middle data size and 1.01× for large data size. Also, the benefit on 128-thread Niagara-2 is 1.23× for small, 1.08× for middle and 1.06× for large data size. As can be expected, the relative improvement for hierarchical phasers is more for smaller data sizes, where the relative impact of synchronization overhead is larger. We can expect larger improvements in the future as the number of threads in an SMP node increases from hundreds to thousands. Both

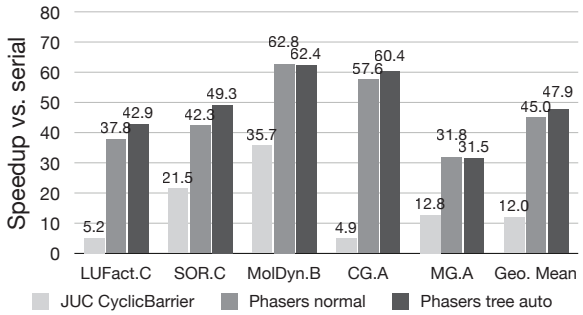


Figure 17: Application Benchmark Performance (128-thread Niagara 2)

HW threads	Data Size	Geometric mean of speedups		
		cyclicBarr	phaser normal	phaser tree
64	small	1.6×	18.7×	20.1×
64	middle	5.8×	27.6×	28.3×
64	large	19.1×	34.0×	34.4×
128	small	0.7×	15.6×	19.2×
128	middle	3.0×	33.7×	36.5×
128	large	12.0×	45.0×	47.9×

Table III: Average Speedup for Benchmark Applications on 64-threads/128-threads Niagara-2 system

flat and hierarchical phasers show dramatically larger speedups than CyclicBarrier.

VII. Related Work

There is an extensive literature on barrier synchronization and reduction. In this section, we focus on a few past contributions that are most closely related to this paper.

The tournament barrier [20] is a binary-tree style hierarchical barrier, which is based on the butterfly barrier [35]. The barrier requirement can be viewed as a tournament. Only one process from each two-process game will continue to the next round, and the overall winner announces the end of the contest to all other processes. Unlike tree-based phasers, the winners and total number of processes are fixed, and cannot be changed dynamically.

Gupta *et al.* introduced an adaptive combining tree approach to reduce the latency in recognition of tree-based barrier synchronization [36]. The processes that arrive early at the barrier adapt the combining tree so that it has a structure appropriate for reducing the latency for the processes that arrive later. Also, Scott *et al.* extended this work to improve memory and interconnect contention issues [37].

Collective operations are critical for high performance computing, and much of research on collective

Data Size	JGF LUFact	JGF SOR	JGF MolDyn	NPB MG	NPB CG
small	size-A	size-A	size-A	class-S	class-S
middle	size-B	size-B	size-B	class-W	class-W
large	size-C	size-C	size-B	class-A	class-A

Table IV: Data size for each benchmark

and reducing operations has been guided by MPI and MPI-like libraries [38].

For all busy-wait barrier implementations, thread switching overhead to handle more software threads than hardware threads is a common problem. Parallel loop chunking [27] is a useful compiler optimization to limit the impact of this thread switching overhead.

VIII. Conclusions and Future Work

Our performance results show significant benefits from the hierarchical approach compared with flat phasers and even with hand-coded trees of phasers. For a barrier microbenchmark, hierarchical phasers performed better than flat phasers for $16+$ threads on a 64-thread Niagara-2 SMP with a best case improvement factor of $1.58\times$ for 64 threads; on a 128-thread Niagara-2 SMP, hierarchical phasers performed better than flat phasers for $4+$ threads with a best case improvement factor of $3.94\times$ for 128 threads. These benefits become even more significant for reductions. For a reduction microbenchmark, hierarchical accumulators performed better than flat accumulators for $4+$ threads on a 64-thread Niagara-2 SMP with a best case improvement factor of $1.92\times$ for 64 threads; on a 128-thread Niagara-2 SMP, hierarchical accumulators performed better than flat accumulators for $4+$ threads with a best case improvement factor of $16.3\times$ for 128 threads. Our results show that the choice of *(numTiers, numDegree)* parameters for a given architecture can also have a significant impact on performance, thereby offering opportunities for future work on auto-tuned selection of these parameters. Other opportunities for future research related to hierarchical phasers include extensions for a hybrid solution that adaptively selects between tree-based or single-level phasers depending on the machine and number of threads used, and dynamic re-balancing of the phaser tree when activities are dropped from the tree.

Acknowledgments

This work was supported in part by the National Science Foundation under the HECURA program, award number CCF-0833166. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation. We also gratefully acknowledge support from Microsoft fund R62710-792. We would like to thank all Habanero team members for their contributions to the HJ software that served as the foundational infrastructure for this research. Finally, we would like to thank Doug Lea for access to the UltraSPARC T2 SMP system used to obtain the experimental results reported in this paper, and for his feedback on an early draft of this paper.

References

[1] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, 2007.

[2] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. New York, NY, USA: ACM, 1998, pp. 212–223.

[3] E. Allan, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt, "The Fortress language specification version 1.0," Sun Microsystems, Tech. Rep., Apr. 2005.

[4] J. Reinders, *Intel threading building blocks*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007.

[5] T. Peierls, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes, *Java Concurrency in Practice*. Addison-Wesley Professional, 2006.

[6] J. Duffy, *Concurrent Programming on Windows*. Addison-Wesley, 2008.

[7] A. R. Board, *OpenMP Fortran Application Program Interface v 3.0*, 2008.

[8] 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 *Proceedings of OOPSLA '05*. New York, NY, USA: ACM Press, 2005, pp. 519–538.

[9] "The Habanero Java (HJ) Programming Language." [Online]. Available: http://habanero.rice.edu/hj

[10] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-First and Help-First Scheduling Policies for Async-Finish Task Parallelism," in *IPDPS '09: International Parallel and Distributed Processing Symposium*, 2009.

[11] S. Chandra, V. Saraswat, V. Sarkar, and R. Bodik, "Type inference for locality analysis of distributed data structures," in *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2008, pp. 11–22.

[12] R. Barik and V. Sarkar, "Interprocedural load elimination for dynamic optimization of parallel programs," in *The Eighteenth International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2009.

[13] J. Shirako *et al.*, "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*. New York, NY, USA: ACM, 2008, pp. 277–288.

[14] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer, "Phaser Accumulators: a New Reduction Construct for Dynamic Parallelism," in *23rd IEEE IPDPS*, 2009.

[15] "X10 v1.7 language specification," http://x10.sourceforge.net/docs/x10-170.pdf.

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

[17] D. Lea, "Proposed phaser class in java.util.concurrent library for java 7 release," http://g.oswego.edu/dl/concurrent/dist/docs/java/util/concurrent/Phaser.html.

[18] A. Miller, "Set your java 7 phasers to stun," http://tech.puredanger.com/2008/07/08/java7-phasers/, 2008.

[19] "Personal communication with Doug Lea regarding an earlier unpublished version of this paper."

[20] D. Hengsen, R. Finkel, and U. Manber, "Two algorithms for barrier synchronization," *International Journal of Parallel Programming*, vol. 17, no. 1, 1988.

[21] K. Yelick *et al.*, "Productivity and performance using partitioned global address space languages," in *Proceedings of the international workshop on Parallel symbolic computation*. New York, NY, USA: ACM, 2007, pp. 24–32.

[22] R. Gupta, "The fuzzy barrier: a mechanism for high speed synchronization of processors," in *Proceedings of the third international conference on Architectural support for programming languages and operating systems*. New York, USA: ACM, 1989, pp. 54–63.

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

[24] V. Sarkar, "Synchronization Using Counting Semaphores," in *Proceedings of the International Conference on Supercomputing*, July 1988, pp. 627–637.

[25] P. Charles *et al.*, "X10: an object-oriented approach to non-uniform cluster computing," in *Proceedings of the ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, New York, NY, USA, 2005, pp. 519–538.

[26] N. Vasudevan, O. Tardieu, J. Dolby, and S. A. Edwards, "Compile-time analysis and specialization of clocks in concurrent programs," in *Proceedings of the 2009 International Conference on Compiler Construction (CC 2009)*.

[27] J. Shirako *et al.*, "Chunking parallel loops in the presence of synchronization," in *ICS '09: Proceedings of the 23rd annual international conference on Supercomputing*. New York, NY, USA: ACM, 2009, pp. 181–192.

[28] 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 parallelism algorithms and architectures*. New York, NY, USA: ACM, 2007, pp. 229–240.

[29] A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick, "Parallel programming in Split-C," in *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, 1993, pp. 262 – 273.

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

[31] "EPCC OpenMP Microbenchmarks," http://www2.epcc.ed.ac.uk/computing/research_activities/openmpbench/openmp_index.html.

[32] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," in *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, 2007.

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

[34] "Nas parallel benchmarks," http://www.nas.nasa.gov/Resources/Software/npb.html.

[35] E. D. Brooks III, "The butterfly barrier," *International Journal of Parallel Programming*, vol. 15, no. 4, 1986.

[36] R. Gupta and C. R. Hill, "A scalable implementation of barrier synchronization using an adaptive combining tree," *International Journal of Parallel Programming*, vol. 18, no. 3, 1989.

[37] M. Scott and J. Mellor-Crummey, "Fast, Contention-Free Combining Tree Barriers for Shared-Memory Multiprocessors," *International Journal of Parallel Programming*, vol. 22, no. 4, pp. 449–481, 1994.

[38] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance analysis of mpi collective operations," *Cluster Computing*, vol. 10, no. 2, 2007.