Insertion Tree Phasers: Efficient and Scalable Barrier Synchronization for Fine-grained Parallelism

Stefan Marr[1], Stijn Verhaegen, Bruno De Fraine, Theo D'Hondt, and Wolfgang De Meuter
Software Languages Lab, Vrije Universiteit Brussel
Brussels, Belgium
Email: {stefan.marr, stverhae, bdefrain, tjdhondt, wdmeuter}@vub.ac.be

***Abstract*—This paper presents an algorithm and a data structure for scalable dynamic synchronization in fine-grained parallelism. The algorithm supports the full generality of phasers with dynamic, two-phase, and point-to-point synchronization. It retains the scalability of classical tree barriers, but provides unbounded dynamicity by employing a tailor-made insertion tree data structure.**

It is the first completely documented implementation strategy for a scalable phaser synchronization construct. Our evaluation shows that it can be used as a drop-in replacement for classic barriers without harming performance, despite its additional complexity and potential for performance optimizations. Furthermore, our approach overcomes performance and scalability limitations which have been present in other phaser proposals.

***Keywords*-Fine-grained parallelism; barriers; phasers; synchronization; many-core; data structures; trees; algorithms**

I. Introduction

With the rise of many-core architectures and the prospect of exa-scale computing, fine-grained parallelism becomes relevant for a wide range of problems and will be necessary to fully utilize the available computational capacities [1]. In conjunction with this hardware development, new constructs have been proposed to enable the synchronization of fine-grained parallelism. Most notable are X10's clocks [2] and Habanero's phasers [3].

Clocks, as well as phasers, are constructs for barrier synchronization, i. e., to synchronize activities. In addition to the notion of a classic barrier, clocks support adding and dropping participants dynamically. Phasers extend clocks through a mechanism that enables point-to-point synchronization. This allows for the expression of dependencies between activities on a more fine-grained level in order to reduce the typical over-synchronization of barriers. From our perspective, phasers are currently the most sophisticated barrier synchronization construct proposed in the literature.

The literature regarding implementation strategies for barrier synchronization discusses different implementation aspects and optimizations [4]–[7]. Notably absent is the discussion of dynamic barriers, i. e., barriers with support for changing groups of participants. This changed only with the introduction of clocks and phasers.

With the proposal of hierarchical phasers, a scalable implementation of barrier synchronization with the full generality of phasers was introduced [8]. However, the proposed approach has limitations.

In this paper, we propose an implementation strategy that is derived from traditional tournament barrier implementations to overcome those limitations. This paper's contributions are as follows:

- an implementation strategy which combines the scalability of traditional tree barriers with phaser semantics
- the insertion tree structure with the necessary stability for synchronization of dynamic groups of participants
- the first complete documentation of an implementation strategy for scalable phasers[2]
- an evaluation of the scalability and its use as a drop-in replacement for traditional barriers

II. Barrier Synchronization Constructs

Barriers are constructs used to synchronize concurrently executing activities. A barrier is a synchronization point which all activities have to reach before any of them can progress further. Depending on the language or library used, a barrier can be restricted to a subset of the activities in the systems. We refer to activities that participate in a barrier as *participants*.

Barriers are often used in the field of scientific computing and allow for the construction of algorithms that operate on shared data in a step-wise manner. This approach ensures data-integrity and avoids data-races by construction.

However, barriers usually imply over-synchronization, which restricts the realized parallelism more than is necessary. To overcome this over-synchronization, barriers have been extended by two main concepts. The first is the notion of *fuzzy* or *two-phase* barriers [4], [5] and the second is point-to-point synchronization.

The goal of fuzzy/two-phase barriers is to reduce the unnecessary stalling of activities. To this end, barrier synchronization is split into two steps. The first step is completed with a *checkpoint*. It is reached when all data-dependencies

[1]Supported by a doctoral scholarship of IWT-Vlaanderen, Belgium.

[2]All source code is available at http://barriers.googlecode.com/

for the adjacent phase have been satisfied. Thus, the data which is required by other activities has been generated. At the checkpoint, the participant announces synchronization, but remains in the same phase and continues with other useful computations without waiting. After all work has been done for that phase, the second step is completed and the participant reaches the *decision point* to await the synchronization of the barrier. If all participants have already reached the checkpoint, a participant does not need to block at a decision point. This allows the parts of adjacent phases that do not have direct data dependencies to overlap. Furthermore, as soon as all participants have reached the checkpoint, the synchronization is propagated, hiding the overhead of the barrier operation and making this information immediately available at the decision point.

The *point-to-point* synchronization allows for the expression of producer-consumer relationships between participants of a barrier. This concept was first proposed by Shirako et al. with phasers [3]. Phasers circumvent the problem of over-synchronization by making data dependencies explicit. For this purpose, activities can indicate a mode with which they want to be registered on the phaser. To participate in a phaser as a producer, an activity is registered in *signal-only* mode. Thus, it will only announce synchronization for a phase, but will never await global synchronization. Consumers, on the other hand, can be registered in *wait-only* mode. This implies that they will never actively announce synchronization, but will await global synchronization, usually to ensure that their data dependencies are satisfied.

In addition, phasers introduce single statements. They are used to specify code which is executed exactly once during a phase transition. This is similar to OpenMP's single-thread execution regions and typical uses of the return value of the POSIX Threads barrier. However, the single statement allows an additional barrier synchronization to be omitted, since it executes the code explicitly during a phase transition and thus performs considerably better.

X10's clocks [2] were designed as a barrier construct to facilitate fine-grained fork/join parallelism. They support fuzzy/two-phase barrier semantics, but more importantly enable participants to join and leave a barrier dynamically. Phasers also provide this functionality. Both constructs are the first to enable barrier synchronization in the presence of fully dynamic fork/join parallelism.

III. Barrier Implementation Strategies

This section will discuss implementation strategies for barriers. We will give a brief overview of traditional barriers, clocks, phasers, and the limitations of these strategies.

A. Traditional Barriers

Until recently, the literature on barrier implementation strategies has disregarded dynamic groups of participants. Thus, classic implementation strategies do not support adding or dropping participants from a barrier.

The simplest barrier is a central barrier. It is implemented using an atomically updated counter [7]. It can be adapted to support dynamic participants, but unfortunately does not scale well. The dissemination barrier scales better, but requires barrier participants to be known upfront [9], which prevents its use as a dynamic barrier.

Tree barriers like the tournament barrier are not prepared to allow the adding and dropping of participants either. However, the general concept behind the tournament barrier is adaptable for our purpose, thus we will briefly describe the algorithm of Hensgen et al. [9].

The idea is to view the process of determining global synchronization as a tournament with $log_2(\#participants)$ rounds. Hensgen used an algorithm that initialized the barrier data structures with predefined winners and losers for each round. When a participant reaches the barrier, it starts in the first round. If it is a loser in that round, it sets a flag and then busy-waits on a global flag which indicates barrier synchronization. The winner in that round busy-waits on the flag set by the loser before it goes on to the next round. The overall winner sets the global flag on which all other participants busy-wait.

B. Clocks and Phasers

The implementation of clocks is based on a central counter. Once all participants have reached the checkpoint, the central phase counter is incremented. At the decision point, the participant awaits the correct phase counter value.

Phasers are more complex. Their implementation is based on a master activity which performs the synchronization. The master can be either fixed or chosen at runtime. When the master arrives at the barrier to achieve global synchronization, it waits for all other participants to announce that they have reached the next phase. A phaser maintains a list of synchronization objects for the registered activities. Each activity has a corresponding object which encapsulates, among other information, a signal count. The master activity iterates over a list with these signal counters and busy-waits on each until the participant announces synchronization by increasing the counter. Afterwards, the master can execute a single statement and then announce global synchronization by incrementing a global phase counter. The reason for using a master activity seems to be the additional complexity which is introduced by signal-only participants. Since they can advance their own phase count independently, additional precautions are necessary to achieve correct global synchronization.

Hierarchical phasers have been proposed to provide better scalability than the original phasers [8]. The basic idea is to rely on a tree structure to achieve scalability. Every tree node is a phaser. The structure of the tree is predefined by the parameter *tiers*, which defines the maximal

number of levels of the tree, and the parameter *degree*, which represents the arity of the tree. Problematic with the proposal of Shirako and Sarkar is that they leave out details necessary to replicate their implementation. From our perspective, the exact strategy to build the tree and add new participants is crucial for the performance properties of their implementation. Depending on the chosen approach, there are different possibilities for interacting with the tree and modifying it, which have an impact on the scalability and runtime overhead. The authors themselves acknowledge that the need for atomic operations increases the registration overhead, but do not detail their strategy. Sec. VI will discuss this in more detail together with our evaluation results.

IV. Implementing a Dynamic Barrier with an Insertion Tree

We will briefly describe the basic principles used for the synchronization tree, including the overall invariants which have to be satisfied to allow consistent synchronization, the data structures, and the used algorithm.

A. Basic Principles of the Synchronization Tree

The barrier participants are the leaf nodes of the tree, and all non-leaf nodes (helper nodes) are used to allow synchronization in $log_2(\#participants)$ steps.

We are using shedding card games as a metaphor. In games like Rummy, cards must be discarded as fast as possible and the game goes on until only one player is left, which is the overall loser. Thus, for our synchronization tree, a winning activity reaches a helper node first. Afterwards, it can go on to do useful computations. The other activity for that helper node loses and needs to crawl up the tree and challenge another *loser*. The last activity reaching the root is the overall loser and has to announce synchronization.

For the data of a helper node in the tree, a simple rule applies: every value represents the aggregated values of its corresponding subtree. From that rule follows property (1): a set wait-only flag indicates that all participants in the subtree are registered in wait-only mode and therefore do not actively participate in synchronization. For the phase counts it means that a helper node always has to indicate the minimum phase count of all participants in its subtree. We define this as property (2).

Fig. 1 shows an example of a synchronization tree. Two participants already dropped from the barrier. One participant is registered in signal-only mode and has already advanced to phase 20. One participant is explicitly registered in wait-only mode and the two remaining ones are registered in the standard signal-and-wait mode. However, one of them has already signaled synchronization in the current phase. When the other does so too, it has to determine global synchronization to allow all participants to proceed to the next phase. The free list enables the reuse of free leaf nodes in the synchronization tree. The list is needed to avoid

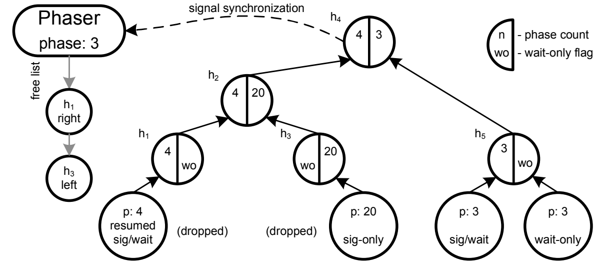

Figure 1. An Insertion Tree Barrier with Four Participants

unnecessary growth of the tree while maintaining a stable tree structure.

A general prerequisite for our approach is that an activity can only be added to the barrier by an activity that already participates in the barrier. Furthermore, the adding participant must not have announced synchronization, i. e., it can not add a new participant to the barrier after reaching the checkpoint. This guarantees that the barrier does not reach global synchronization during an add operation.

B. Insertion Tree

The main problem we had to overcome was building a tree structure that allows the addition of new nodes while maintaining the invariants for synchronization.

Our solution to this problem is called an *insertion tree*. It is an inverted tree, thus child nodes have a pointer to their parent, but the parent does not know its child nodes. This choice was primarily relevant during the design of the algorithm. The strict inverted tree made several problems obvious, but is not essential for the approach.

The tree itself does not support a real remove operation, but relies on a free list to enable the reuse of dropped nodes.

In order to maintain the synchronization invariants and allow activities to announce synchronization in parallel to insert operations, we need to minimize the possible tree modifications. We achieve that by using a node insertion strategy which could be named *complete smallest subtree first*. With this strategy we can insert a node knowing only the last inserted node and the number of leaves in the tree. Fig. 2 illustrates the insertion of nodes. The number of leaves defines exactly where the next insert operation has to be done (`insertNode`) to fill the smallest subtree first. The algorithm is basically walking up the tree as long as the number of nodes for a level indicates that the current subtree is complete.[3]

The insert operation creates and initializes a helper node, which becomes the parent of the new participant node. Then it sets the parent pointer of the helper node to the parent of `insertNode`, and the inserted participant points to the helper node. The operation is completed by changing the parent pointer of `insertNode` to the helper node.

[3] i = numLeaves; while (i mod 2 == 0) {insN = insN.parent; i /= 2;}

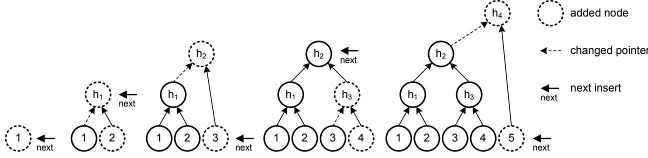

Figure 2. Steps of a Growing Insertion Tree

```
Barrier
numLeave  :int
lastNode:Participant
phase     :int
insertLock:Lock
firstFree :FreeNode

FreeNode
next   :FreeNode
parent :HelperNode
isLeft :bool

HelperNode
parent   :HelperNode
leftChild:addr
sync.waitOnlyL:bool
sync.waitOnlyR:bool
sync.phaseLeft :int
sync.phaseRight:int

Participant
mode        :Mode
resumed     :bool
phase       :int
globalPhase:int*
barrier:Barrier
parent:HelperNode
```

Figure 3. Used Data Structures

C. Data Structure

Fig. 3 visualizes the used data structures. The main components of the *helper nodes* are the parent pointer, an identifier of the left child node and its synchronization state. The `leftChild` identifier is used to determine from which subtree a participant traversed up the tree. This is necessary to be able to add nodes (cf. Sec. IV-D3). The synchronization state, i. e., the phase counts and wait-only flags need to be changeable atomically to avoid data races and read-inconsistencies. Notice that is is important to take measures to prevent problems that can be caused by phase count overflows. The wait-only flags indicate whether a subtree's participants actively take part in the synchronization.

The *participant node* represents an activity taking part in the barrier. The first field is the `mode` with which the participant was registered on the barrier. The `resumed` field indicates whether the participant already announced synchronization. It is not relevant for participants in *signal-only* mode. The pointer to the global phase count is used to check for completion of a phase, i. e., synchronization.

The *barrier* maintains a count of the number of leaves, a pointer to the last leaf for determining the insert node, a pointer to the first free-list element of dropped participants, and a global lock to sequentialize insert and drop operations. Furthermore, it maintains the global phase count. It is used to initialize new participants correctly, and to notify all participants of global synchronization.

To allow drop operations from the barrier, we maintain a free list. The elements of this free list store the parent of the free place, a flag to remember whether it was the left child, and a pointer to the next free-list element.

D. Algorithm

In this section, we will discuss the details of the synchronization algorithm and the approaches to solve the problems caused by adding and dropping nodes. The used pseudo-code is inspired by Python and uses significant whitespace for conciseness.

1) Announcing Synchronization: When a participant reaches the *checkpoint* it needs to announce this. Lst. 1 provides the corresponding pseudo-code.

First the participant checks whether it already announced synchronization or it is registered in wait-only mode. In either case, it does not need to proceed.

Otherwise, it has to remember it already announced synchronization and can then start to propagate the announcement to its parent node. Activities that only participate by signaling disregard the resumed flag altogether, but also increase their local phase counter.

A participant can only win at a helper node if the opponent is not registered in wait-only mode. Furthermore, the corresponding phase count in the helper node needs to be smaller than the value the participant wants to announce to be able to win. If the participant is too late and loses, it has to walk up the tree and propagate synchronization further.

In the event that the participant reaches the top of the tree as the last participant, it becomes the overall loser, and needs to indicate global synchronization.

Before doing so, the single statement needs to be executed. It is safe to do that here, since there is only one participant that can reach this point. Thus, it is guaranteed to be executed only once during a phase transition. Furthermore, all activities have already announced synchronization and adding new participants to the barrier is not allowed at this point in time.

```
def resume(p):   # p is participant
  if p.resumed or p.mode == WAIT_ONLY:
    return false

  p.resumed = p.mode != SIGNAL_ONLY # resume flag
  p.phase++                # increment local phase
  n    = p.parent
  prev = p
  propagate = true

  while n and propagate:
    atomic:
      if n.leftChild == prev:
        propagate = n.flags.waitOnlyR or
              (n.flags.phaseRight >= p.phase)
        n.flags.phaseLeft = p.phase
      else:
        propagate = n.flags.waitOnlyL or
              (n.flags.phaseLeft >= p.phase)
        n.flags.phaseRight = p.phase
    prev = n
    n    = n.parent
  if propagate: # overall loser notifies all
    p.barrier.executeSingleStmt()
    p.globalPhase++
  return propagate
```

Listing 1. Announcing Synchronization

2) Dropping Participants: Notice that a dropped participant is equal in its semantics to a participant which is registered in wait-only mode and hence the tree does not need to be changed. For participants in other modes, we need to announce synchronization and propagate the wait-

only mode (cf Lst. 2). Announcing synchronization works the same as in the general case. If both subtrees are marked wait-only, this information has to be announced to the parent node to mark the whole subtree as wait-only. After dropping a participant, an item is added to the free list which describes the free spot in the tree.

```
def announceDrop(p):
  p.phase++    # increase local phase count
  propagateResume = propagateWaitOnly = true
  n    = p.parent
  prev = p

  while n and (propagateWaitOnly or propagateResume):
    atomic:
      if propagateResume:
        if n.leftChild == prev:
          propagateResume = (n.flags.waitOnlyR
            or (n.flags.phaseRight >= p.phase))
          n.flags.phaseLeft = p.phase
        else:
          propagateResume = (n.flags.waitOnlyL
            or (n.flags.phaseLeft >= p.phase))
          n.flags.phaseRight = p.phase
      if propagateWaitOnly:
        if n.leftChild == prev:
          n.flags.waitOnlyL = true
        else:
          n.flags.waitOnlyR = true
        propagateWaitOnly =
          (n.flags.waitOnlyL and n.flags.waitOnlyR)
    prev = n
    n    = n.parent
  if propagateResume: # overall loser notifies all
    p.barrier.executeSingleStmt()
    p.globalPhase++
  return propagateResumeFurther
```

Listing 2. Announcing Dropped Participant

3) Adding Participants: New participants can join the barrier either by reusing an unoccupied leaf or adding a new leaf to the tree.

If a new node is added to the tree, additional care has to be taken with the phase counts to ensure property (2). Lst. 3 contains the code for adding a new node. First, the newly created helper node is initialized properly, then the `insertNode` will become the left subtree for the new helper node. The participant itself will be the right subtree (cf. Sec IV-B). The correct wait-only flags for the helper node depend on the mode of the participant and of the `insertNode` subtree.

```
def insertNewIntoTree(insertNode, p):
  insertParent     = insertNode.parent
  helper           = new HelperNode
  helper.parent    = insertParent
  helper.leftChild = insertNode
  helper.initWaitOnly(insertNode, p)
  helper.setPhases(p.phase)

  insertNode.parent = helper # modify tree

  # move phase count
  atomic:
    diff = insertParent.flags.phaseRight - p.phase
    insertParent.flags.phaseRight   = p.phase
    insertParent.flags.waitOnlyR  &&= p.mode
  atomic:
    if helper.flags.phaseLeft == phase:
      helper.flags.phaseLeft = phase + diff
  p.parent = helper
```

Listing 3. Added New Participant Node

By initializing the helper node with the current minimum phase count, i. e., the global phase count, it is guaranteed that the left subtree will win at the helper node and will not traverse up the tree to announce synchronization.

After modifying the tree, the old phase count of `insertParent`, the old parent of `insertNode`, needs to be preserved since it represents the synchronization of that subtree. Thus, in an atomic operation, the difference to the global phase count has to be determined and `phaseRight` has to be set to the minimum phase count of the subtree, which is equal to the global phase count. Furthermore, the wait-only flag should be set correctly.

In a second atomic operation, the difference is added to the helper node's `phaseLeft` counter. However, it is only to be set if the value is still equal to the global phase count, otherwise signals from that subtree could get lost.

After completing the insertion into the tree, the overall consistency properties (1) and (2) need to be restored.

If a participant is added in place of one which was dropped before and the new participant is registered in wait-only mode, no action has to be performed. If the participant is added as a new node but uses the wait-only mode, the wait-only mode needs to be propagated identical to dropping a node. For all other cases, the code in Lst. 4 gives an overview of the necessary actions.

```
def announceAdd(node, phase):
  n    = node.parent
  prev = node
  expectedP = None
  while n:
    # ensure racing subtree was here
    while expectedP:
      if ((n.leftChild == prev and
          n.flags.phaseLeft  == expectedP)
        or (n.leftChild != prev and
          n.flags.phaseRight == expectedP)):
        expectedP = None

    atomic(n.flags):
      if n.leftChild == prev:
        n.flags.phaseLeft = phase
        expectedP = n.flags.phaseRight
      else:
        n.flags.phaseRight = phase
        expectedP = n.flags.phaseRight
      if n.leftChild == prev:
        n.flags.waitOnlyL = false
      else:
        n.flags.waitOnlyR = false
    prev = n
    n    = n.parent
```

Listing 4. Announcing Added Participant

When a dropped node has been replaced, the function `announceAdd` is called with this participant node and the global phase count. If the participant was added as a new node, the function is called with `insertParent` instead of the participant node.

While propagating the minimum phase count, there is one critical data race. If synchronization is already complete for the subtree before we as p_1 add the node, it is possible that another participant p_2 loses at a node n_1 to which

synchronization was propagated. Now, it can happen that we (p_1) reach that particular node n_1, reset the phase count, proceed up the tree to n_2 and reset it too. A third participant p_3 could then win at the reset node n_2 before the p_2 reaches it. This would mean that p_2 would propagate a wrong phase count, possibly even announcing global synchronization. To avoid that, we need to wait for p_2, i. e., until the expected phase count is set on the node we are going to reset.

Lastly, the wait-only flags along the tree have to be set correctly together with the phase count.

V. Performance Evaluation

The objective of our performance evaluation is threefold. First, we evaluate the scalability and whether the two-phase support hides the barrier overhead in typical load imbalances. Second, we verify that the insertion tree phaser can compete with standard barrier implementations in terms of its barrier overhead to allow it to be used as a drop-in replacement for existing applications. Third, we evaluate the performance of dynamic task creation.

The literature convincingly shows the benefits of the advanced barrier and phaser concepts [3]–[6], [8], [10]. Thus, we will primarily focus on the actual synchronization overhead.

A. Experimental Environment and Methodology

We run our experiments mainly on a Tilera TILExpressPro64 using the Tilera MDE 2.1.0 Linux software stack.[4] The TilePro64 CPU has 64 cores clocked at 700MHz. We use the standard configuration for our benchmarks. In this setting only 59 cores are available for user programs. Since the caches on each core are rather small compared to typical desktop CPUs, the manual recommends `-Os` as the standard optimization setting, which we use together with deactivated assertions (`-DNDEBUG`). All experiments are also executed on an Intel machine with two Intel Xeon E5520 CPUs at 2.27GHz with hyper-threading enabled. However, the results do not show significant differences between the barrier algorithms and thus are left out.

As barrier implementations, we use a classic central spinning barrier [7], a dissemination barrier [7], a tournament barrier, a C++ port of Habanero's phasers, and our own insertion tree phaser implementation. Furthermore, we make comparisons to a spinning barrier, which is part of the Tilera libraries.

Since modern systems provide many optimizations which can affect benchmark results in unexpected ways, we follow the suggestions of Georges et al. [11]. Furthermore, to avoid the influence of thread migration and scheduling overhead, all threads are pinned to a separate physical core.

The benchmarks are executed and statistically analyzed by ReBench, a benchmark execution tool.[5] Every benchmark is executed at least 100 times, and up to 200 times to reach a confidence interval of 0.95. We always report the median of the measured samples.

[4] http://www.tilera.com/

[5] http://rebench.googlecode.com/

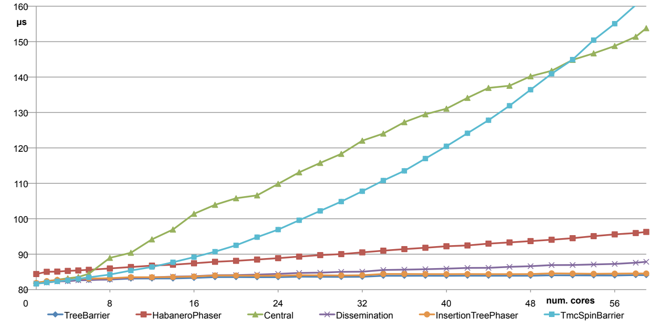

Figure 4. Two-Phase Barrier Microbenchmark

B. Barrier Microbenchmark

For our first experiment, we use a slightly adapted EPCC microbenchmark to compare the pure barrier performance of our approach to other barrier implementations [12].

```
for 1 to 10000:
  delay(500)


  next
```

Listing 5. EPCC Benchmark

```
for 1 to 10000:
  delay(500)
  signal
  delay(250)
  next
```

Listing 6. For Two-Phase Barriers

Lst. 5 shows the standard version, meant to measure the synchronization overhead of a classic barrier. Lst. 6 presents an adapted version, which is meant for two-phase barriers.

Fig. 4 shows that the TreeBarrier, InsertionTreePhaser, and the dissemination barrier have good scalability characteristics. The average size of the confidence intervals within 0.15%, therefore it is not significant and thus not indicated.

To assess the influence of proper two-phase barrier support, we set the results of the EPCC microbenchmarks using one-phase synchronization in relation with a version using two-phase synchronization. A barrier which does not support two-phase synchronization should exhibit the same performance characteristics in both benchmarks. In a two-phase barrier, part of the overhead of the checkpointing should be hidden in the execution imbalances of the different cores. Thus, the observable barrier overhead should become smaller. Fig. 5 shows the corresponding graph. The graph is obtained by dividing the barrier overhead measured in the two-phase microbenchmark by the overhead measured in the classic barrier microbenchmark. The TmcSpinBarrier and the central barrier behave as is expected,. They show the same overhead in both measurements. The original phaser exhibits a minimally smaller overhead for the two-phase barrier benchmark, but since most of the synchronization work is done at the decision-point, it does not hide the barrier

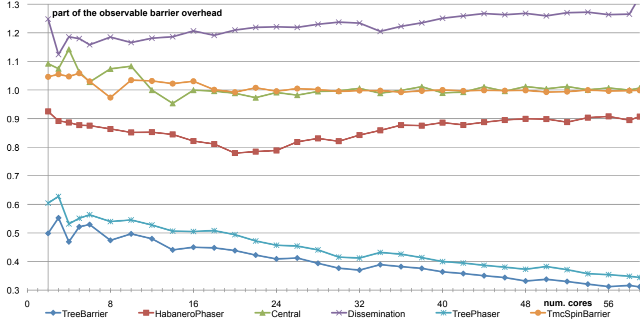

Figure 5. Two-Phase Barrier Overhead vs. Classic Barrier Overhead

overhead as well as our TreeBarrier and InsertionTreePhaser implementations. For the TreeBarrier, only 31% of its overhead is observable, and for the TreePhaser, the observable overhead is 34%.

The microbenchmarks show that our approach scales as is expected of a tree barrier. Furthermore, it is able to benefit fully from its support for two-phase barrier synchronization by hiding a major portion of the barrier overhead.

C. Application Benchmarks

To get a better impression of the influence on application performance, we chose the *Modified SPLASH-2* benchmarks[6] for our evaluation. We use the recommended core benchmarks and, also included the LU benchmark, since it uses only barriers as a synchronization mechanism [13].

The goals of these benchmarks is to measure the classic barrier performance of our approach and to verify its usability as a drop-in replacement for existing algorithms. Thus, the applications are not adapted, but use classic barrier operations only.

Depending on their characteristics and requirements with respect to the allowed number of threads, we run them with 4, 8, 16, 32, and 58 threads. To investigate the scalability and the influence of different barrier algorithms, we report the results for the largest number of threads. Fig. 6 shows

[6]http://www.capsl.udel.edu/splash/

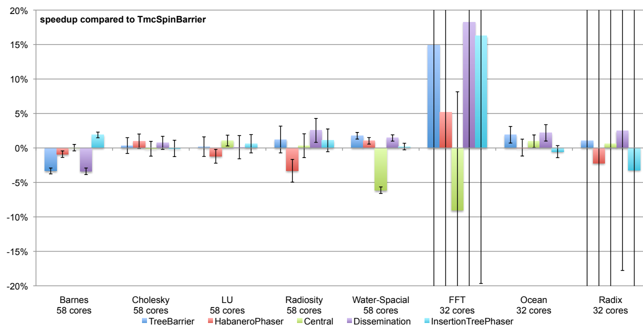

Figure 6. SPLASH-2 on Tilera

Table I
DYNAMIC BARRIER MICROBENCHMARK USING UP TO 59 THREADS

Implementation	mean in ms	std. deviation in ms
Central(Dynamic)	338.6	2.9
HabaneroPhaser	339.4	2.1
InsertionTreePhaser	339.1	2.4

the relative speedup to the TmcSpinBarrier. We chose it as the base for comparison since it is the standard barrier implementation provided with the system. Furthermore, the chart indicates two times the standard deviation, which includes at least 75% of the measured samples. For the interpretation of the chart, it can be noted that the results for FFT and the radix sort benchmark are not significant, but are governed by caching and timing effects in the parallel system.

Our conclusion is that our insertion tree phaser can be used directly as a drop-in replacement for a standard barrier without harming performance. Thus, there is no measurable overhead in supporting the additional phaser capabilities. On the contrary, it provides additional optimization potential by exploiting the two-phase and point-to-point synchronization support, which is not done.

D. Dynamic Barrier Microbenchmark

One important point Shiroko and Sarkar evaluated for their hierarchical phasers was the overhead of adding and dropping activities. Due to additional synchronization, it is more costly than a normal phaser [8].

They used a benchmark based on the EPCC microbenchmarks to measure the barrier overhead in the case of dynamic task creation. We reran their benchmark to investigate whether our approach introduces significant overheads by dynamic task creation. The results can be found in Table I. No significant differences can be found between the three implementations since the main portion of the dynamic overhead comes from thread creation. Thus, it is safe to conclude that our strategy is at least as good as the hierarchical phasers, which need more synchronization than the normal phasers used in our comparison.

VI. DISCUSSION

This section compares our approach briefly with hierarchical phasers.

With respect to the performance characteristics, the current phaser and hierarchical phaser proposals have difficulties with hiding latency in proper two-phase barrier support. As we have shown in Fig. 5, it is possible to hide a large portion of the barrier overhead. The original phasers, which to our knowledge, use the same strategy as hierarchical phasers, hide less than 23% of their overhead while our approach is able to hide up to 66% and thus, can benefit better from two-phaser barrier support.

Another issue with hierarchical phasers is the upfront definition of their tree shape. By defining tiers and the degree of the tree, the overall number of participants for which scalability is given is statically limited. This limits the application in situations where the number of concurrent activities can not be determined upfront, which is a common case for fine-grained fork/join parallelism. Furthermore, it is not clear how graceful their proposal scales from a small number of participants up to very large numbers of participants since the strategy for the tree construction is left open.

For very fine-grained parallelism it becomes important that add and drop operations on a phaser can be executed concurrently. From our understanding of the original phaser implementation, add and drop operations are completely concurrent. The only need for sequentialization is during an add when the synchronization object is added to the list in the phaser. However, the drop operation leaves the synchronization object in the phaser's list. This will lead to scalability problems if drop operations occur frequently. For hierarchical phasers, the authors indicate additional overhead for add operations. However, they do not detail their strategy and thus, it is not clear whether add operations could run in parallel. Drops are probably performed as in the original phasers, and the *garbage collection*, i. e., scalability problem, remains. For insertion tree phasers, the situation is different in that the modification of the tree structure needs to be sequentialized. Thus, the portion of add and drop operations, which changes any pointers in the tree, is protected by a lock. Hence, add and drop operations are only partially concurrent in an insertion tree phaser. It is possible not to use a free list for dropped participants and to allow concurrent drop operations similar to the original phasers, but this would lead to additional growth of the tree which we decided to avoid.

VII. Conclusion and Future Work

We presented in this paper the first completely documented strategy to implement a scalable phaser synchronization construct. Our approach uses the classic strategies of tree-based barriers to achieve their good scalability properties. A novel insertion tree structure enables dynamic parallelism by allowing activities to join and leave the barrier as needed to facilitate fine-grained fork/join parallelism. This paper provides the necessary details to implement an insertion tree phaser straightforwardly. It discusses the data structures, the properties of the synchronization tree, and the potential race conditions that must be considered.

Furthermore, the presented approach overcomes a number of limitations of the hierarchical phaser approach. For example, it does not require a preset tree size, which limits scalability, and it has sufficient two-phase barrier characteristics to hide the barrier overhead in inevitable load imbalances between participating activities. Our evaluation also demonstrates the usability for legacy applications as a drop-in replacement without negatively impacting performance, despite the added complexity to provide the extend phaser capabilities.

Future work is the scalability evaluation for more than 64 cores. Depending on the systems, n-ary trees [8] and node-local spinning [7] could improve performance properties.

References

[1] J. Torrellas, "Architectures for extreme-scale computing," *IEEE Computer*, vol. 42, no. 11, pp. 28–35, November 2009.

[2] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 2005.

[3] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer, "Phasers: A unified deadlock-free construct for collective and point-to-point synchronization," in *ICS 08: Proceedings of the 22nd annual international conference on Supercomputing*. New York, NY, USA: ACM, 2008, pp. 277–288.

[4] R. Gupta and M. Epstein, "High speed synchronization of processors using fuzzy barriers," *International Journal of Parallel Programming*, vol. 19, no. 1, February 1990.

[5] I. Jung, J. Hyun, J. Lee, and J. Ma, "Two-phase barrier: A synchronization primitive for improving the processor utilization," *International Journal of Parallel Programming*, vol. 29, no. 6, pp. 607–627, December 2001.

[6] 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, pp. 161–180, June 1989.

[7] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, 1991.

[8] J. Shirako and V. Sarkar, "Hierarchical phasers for scalable synchronization and reductions in dynamic parallelism," in *24th IEEE International Parallel and Distributed Processing Symposium*, 2010.

[9] D. Hensgen, R. Finkel, and U. Manber, "Two algorithms for barrier synchronization," *International Journal of Parallel Programming*, vol. 17, no. 1, pp. 1–17, February 1988.

[10] 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, August 1994.

[11] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," *SIGPLAN Not.*, vol. 42, no. 10, pp. 57–76, 2007.

[12] J. M. Bull, "Measuring synchronisation and scheduling overheads in openmp," in *In Proceedings of First European Workshop on OpenMP*, 1999, pp. 99–105.

[13] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*. New York, NY, USA: ACM, 1995, pp. 24–36.