Dynamic deadlock verification for general barrier synchronisation

Tiago Cogumbreiro
Imperial College London
cogumbreiro@imperial.ac.uk

Raymond Hu
Imperial College London
raymond.hu05@imperial.ac.uk

Francisco Martins
University of Lisbon
fmartins@fc.ul.pt

Nobuko Yoshida
Imperial College London
n.yoshida@imperial.ac.uk

Abstract

We present Armus, a dynamic verification tool for deadlock detection and avoidance specialised in barrier synchronisation. Barriers are used to coordinate the execution of groups of tasks, and serve as a building block of parallel computing. Our tool verifies more barrier synchronisation patterns than current state-of-the-art. To improve the scalability of verification, we introduce a novel event-based representation of concurrency constraints, and a graph-based technique for deadlock analysis. The implementation is distributed and fault-tolerant, and can verify X10 and Java programs.

To formalise the notion of barrier deadlock, we introduce a core language expressive enough to represent the three most widespread barrier synchronisation patterns: group, split-phase, and dynamic membership. We propose a graph analysis technique that selects from two alternative graph representations: the Wait-For Graph, that favours programs with more tasks than barriers; and the State Graph, optimised for programs with more barriers than tasks. We prove that finding a deadlock in either representation is equivalent, and that the verification algorithm is sound and complete with respect to the notion of deadlock in our core language.

Armus is evaluated with three benchmark suites in local and distributed scenarios. The benchmarks show that graph analysis with automatic graph-representation selection can record a 7-fold execution increase versus the traditional fixed graph representation. The performance measurements for distributed deadlock detection between 64 processes show negligible overheads.

Categories and Subject Descriptors D.4.1 [*Process Management*]: Deadlocks—detection, avoidance; D.3.3 [*Language Constructs and Features*]: Concurrent programming structures—barriers, phasers; F.3.2 [*Semantics of Programming Languages*]: Operational semantics

General Terms Verification

Keywords barrier synchronisation, phasers, deadlock detection, deadlock avoidance, X10, Java

1. Introduction

The rise of multicore processors has pushed mainstream programming languages to incorporate various parallel programming and concurrency features. An important class of these are barriers and their associated techniques. Java 5–8 and .NET 4 introduced several abstractions that expose barriers explicitly or otherwise use implicit barrier synchronisations: latches, cyclic barriers, fork/join, futures, and stream programming. The basic functionality of a barrier is to coordinate the execution of a group of tasks: it allows them to wait for each other at an execution point. Recent languages feature more advanced abstractions, such as clocks in X10 [45] and phasers in Habanero-Java/C (HJ) [7], that support highly dynamic coordination of control flow and synchronisations. In particular, phasers unify the barrier synchronisation patterns available in Java, X10, and .NET under a single abstraction.

As for many other concurrency mechanisms, deadlocks—in which two tasks blocked on distinct barriers are (indirectly) waiting for each other—are one of the primary errors related to barrier synchronisation. Historically, the approach to counter barrier deadlocks has been to restrict the available barrier synchronisation patterns such that programs are barrier-deadlock free by construction, *e.g.*, OpenMP [30] restricts barrier composition to syntactic nesting. To date there are no available tools for barrier-deadlock verification in X10 or HJ, nor for mainstream libraries, such as the Java Phaser [17] and the .NET Barrier [25] APIs.

Two key characteristics exacerbate barrier-deadlock verification in recent languages and systems: barriers can be created dynamically and communicated among tasks as values (called *first-class barriers* [41]); and the group of participants can change over time (*dynamic membership* [29]). Due to the difficulty of statically analysing the usage of first-class barriers (*e.g.*, dealing with aliases and non-determinism statically), the state-of-the-art in barrier-deadlock verification is based on dynamic techniques that monitor the run-time execution of programs (existing tools for static verification handle only a simplistic model where synchronisation is global; see Section 7). Dynamic membership, found in Java, .NET, X10, and HJ, is, however, simply not handled by any existing barrier-deadlock verification tools.

The state-of-the-art in dynamic barrier-deadlock verification follows graph-based techniques and originates from MPI [28] and UPC [42]. Graph-based approaches work by maintaining a representation of the concurrency constraints between running tasks (*e.g.*, the synchronisation state of blocked tasks), and by performing a graph analysis on this state (*e.g.*, cyclic dependencies). While the existing graph-based tools, such as [14, 15], precisely identify deadlocks in systems featuring multiple barriers, these approaches are too limited in the presence of dynamic membership.

The main limitations of the current tools are: (i) a representation of concurrency constraints that assumes static barrier membership, and (ii) committing to the Wait-For Graph [20] analysis technique that is optimised for concurrency constraints with more tasks than barriers (a rare situation for classical parallel programs). Naive extensions to resolve (i) face the problem of maintaining the

PPoPP'15, February 7–11, 2015, San Francisco, CA, USA.

http://dx.doi.org/10.1145/2688500.2688519

membership status of barriers consistently and efficiently; this issue is compounded in the distributed setting, which is a key design point of deadlock verification for languages like X10/HJ. Issue (ii) is related to the dynamic nature of such barrier applications, where the number of tasks and barrier synchronisations may not be known until run-time and may vary during execution. Committing to a particular graph model can thus hinder the scalability of dynamic verification. The State Graph [19] is an alternative model that favours scenarios with more tasks than barriers. In the general case we cannot determine which model is most suitable statically; moreover, this property may change as execution proceeds.

Armus This paper presents Armus, to our knowledge, the first deadlock verification tool for phasers [7, 34]. The contributions of this work are as follows.

- Armus leverages phasers to represent concurrency constraints among *synchronisation events*. This representation enables the analysis of first-class barriers with dynamic membership, and simplifies the algorithm of distributed deadlock detection.
- Armus introduces a technique to improve the scalability of graph-based verification, by automatically and dynamically selecting and transforming between the commonly used Wait-For Graph (WFG) and the alternative State Graph (SG) models.
- We formalise an operational model for a concurrent language with phasers (subsuming the barriers of [17, 25, 45]) and the Armus deadlock verification algorithm. We show that our deadlock verification is sound and complete with respect to the characterisation of deadlock in our language, and establish the WFG-SG equivalence.
- The Armus-X10 and JArmus applications are the first deadlock verification tools for X10 clocks and the Java phaser API, featuring: distributed *deadlock detection* where the tool reports existing deadlocks, and local *deadlock avoidance* where the tool raises an exception before entering a deadlock.

To address (i), Armus introduces a novel representation of concurrency dependencies, based on events in the sense of Lamport's *logical clocks* [22]. A major part of deadlock analysis is the generation of concurrency constraints, attained by bookkeeping the status of tasks and barriers. The insight of our technique is to interpret the operation of waiting for a phase as observing a timestamp; and then to assert a dependency from the phase a task is blocked to any (future) phase the task participates. With this representation Armus simplifies the analysis of dynamic membership, since it avoids tracking the arrival and departure of participants on a synchronisation; bookkeeping the participants of a barrier in a distributed setting is a global state, thus a challenging procedure to maintain.

Armus addresses (ii) with a novel technique that automatically selects between two graph models according to the monitored concurrency constraints. The standard graph model used in graph analysis, the WFG, comes from distributed databases [20], a setting with a fixed number of tasks and dynamic resource creation. The WFG was therefore optimised for concurrency constraints with many resources and few tasks. The underlying assumptions of the WFG no longer hold for languages with dynamic tasks and dynamic barrier creation (first-class barriers), such as X10 and Java. For these applications, Armus proposes a technique that selects either the WFG or the SG depending on the ratio between tasks and barriers. The difference on the size of the graph can be dramatic. For instance, in benchmark `PS` [46], the average edge count decreases from 781 edges to 6 edges (see Section 6.3). The automatic model selection performs at least *as fast* as the usual approach of a fixed graph representation.

We outline the sections of this paper. The following section illustrates the subtleties of barrier deadlock verification regarding first-class barriers with dynamic membership and how Armus addresses these challenges, with connections to later sections. Section 3 presents a programming language abstraction (PL) for general barrier constructs, expressive enough to capture all surveyed barrier synchronisation patterns. In Section 4, we introduce a novel representation of barrier-dependency state based on synchronisation events and define the construction of Wait-For Graphs and State Graphs for Armus deadlock verification. We show that the WFG and SG constructed from a given state are equivalent wrt. the presence of cycles, and that cycle detection gives sound and complete deadlock verification wrt. a deadlock in PL. Next, Section 5 presents the implementation details of Armus, including the distributed deadlock detection algorithm, and applications to Java (JArmus) and to X10 (Armus-X10). Section 6 performs an extensive performance evaluation of Armus in Java and X10, using the NAS Parallel Benchmark, the Java Grade Forum Benchmark suite, and the HPC Challenge benchmark suite. The results show that overall deadlock detection is below 13% in local applications, and negligible in distributed applications. Furthermore, automatic graph model selection outperforms the fixed graph model selection: deadlock detection can have an overhead impact difference of 9%; deadlock avoidance have an overhead impact difference of 518%. Finally, Section 7 discusses related work and Section 8 concludes the paper. ***The Armus web page [3] includes the full proofs of the theorems, the implementation, and the benchmark data.***

```
1  val c = Clock.make();
2  finish {
3    // Spawn I tasks, looping together J times
4    for (i in 1..I) async clocked(c) {
5      for (j in 1..J) {
6        val l = a(i-1);
7        val r = a(i+1);
8        c.advance(); // Cyclic barrier (clock) step
9        a(i) = (l + r) / 2;
10       c.advance();}}
11 } // Join barrier (finish) step: wait on all tasks
12 handle(a);
```

Figure 1: Join and cyclic barrier synchronisation in X10.

2. Barrier programs and deadlocks

This section illustrates the use of different kinds of barriers in X10 and Java, and examples of deadlocks that arise in such programs. We motivate the design of Armus for the more general mechanism of phasers, which has allowed us to directly apply Armus to handle the different forms of barrier programming and barrier synchronisation patterns in languages like X10 and Java. Since a comprehensive deadlock verification for X10 applications must also consider distributed barrier coordination, we discuss the challenges of distributed deadlock detection and motivate the event-based barrier-dependency state used in Armus. Finally, we discuss how the choice of graph model impacts deadlock verification scalability in various scenarios of barrier applications.

2.1 Dynamic barrier membership using X10 clocks

Our running example is a simplified parallel 1-dimensional iterative averaging [35], divided into two stages. The first stage spawns `I` tasks, in parallel, to work on an array `a` of `I+2` numbers. Each task is responsible for updating an element with the average of its neighbours, repeatedly over a series of synchronised iterations. After these tasks have finished, a single task then performs the second stage, some final processing on the resulting array values.

Figure 1 lists an X10 implementation of the running example. The first stage is implemented using a *cyclic barrier*, represented by

the *clock* created and assigned to `c` (an immutable `val`) on Line 1. The `for` loop starting on Line 4 spawns `1..I` parallel tasks (called *activities* in X10) using the `async` statement. All `I` child tasks are registered (`clocked`) with clock `c`; the parent task is implicitly registered upon clock creation. In the `async` body of each task `i`, the inner loop, repeated `J` times, reads `a(i-1)` and `a(i+1)` array values and assigns the average to `a(i)`. Stepwise looping by these parallel tasks is coordinated using the blocking `advance` operation on the clock. A task executes an `advance` by waiting until every task registered with that clock has done so, and then all tasks may proceed. On Line 8, the tasks synchronise between reading the current values in the `j`-th step and writing the new values. Another synchronisation takes place between writing on Line 10 and reading the values in the `(j+1)`-th step.

The second stage is implemented using a *join barrier*. The parent awaits on Line 11 the termination of the `I` tasks spawned in the body of the finish.

Deadlock verification for dynamic membership Deadlock verification for this example must take into account two properties of barrier semantics: group synchronisation, and dynamic membership. The former capability lets groups of tasks synchronise independently, in contrast with global synchronisation. The verification must identify any transitive dependencies among the participants of different groups through the chosen graph analysis. The latter capability lets tasks register and revoke their membership in a barrier, which may introduce subtle deadlocks such as the one affecting the program in Figure 1. In X10, two operations register a task with `c`: creating clock `c` and `clocked(c)`; while `c.drop()` revokes the membership with `c`. The code deadlocks because all of the `I` tasks are stuck on the first `advance`, since the registered parent task never calls `advance`. Armus can monitor the program's execution to detect deadlocks, reporting to the user after the error occurs. A straightforward fix to the deadlock is to insert `c.drop()` immediately before Line 11 to break the circular dependency. Alternatively, Armus can perform deadlock avoidance and an exception is raised in Lines 8 and 11 and the tasks become deregistered from clock `c`.

The X10 language was developed with the goal of simplifying the migration of single-threaded prototype programs to distributed implementations running across, *e.g.*, multiple SMP clusters. Programs refer to each site (*e.g.*, cluster nodes) of the distributed system as *places*. Any X10 statement may be prefixed with `at(p)` to execute that statement at the site referred by place `p` (a value). Clocks work across multiple places, so invoking `at(p) async clocked(c)` spawns a task that runs at place `p` registered with a distributed clock `c`.

We argue that the techniques available in the literature are not adapted to the *distributed* analysis of barrier deadlocks, because of the effort to monitor dynamic membership across all sites of the distributed system. Such techniques, developed mostly to verify lock-based deadlocks, track the status of each blocked operation with the objective of obtaining dependencies between tasks. Since barrier synchronisation is a collective operation, a distributed algorithm incurs in the additional effort of aggregating the arrival status of each participant [14, 15]. Essentially, the analysis ends up recreating a significant part of the actual synchronisation protocol, which Agarwal *et al.* [2] observe to be the non-trivial part of distributed barrier synchronisation with dynamic membership. Armus proposes an alternative representation that is oblivious of the status of the barrier operation—a global property—and instead considers the local impact of each task in the global ordering of barrier synchronisation.

```
1  c = new Phaser(1); // "Cyclic barrier" phaser
2  b = new Phaser(1); // "Join barrier" phaser
3  for (int i = 1; i <= I; i++) {
4    c.register(); b.register();
5    new Thread() { // Spawn task i
6      public void run() {
7        for (int j = 1; j <= J; j++) {
8          l = a[i-1];
9          r = a[i+1];
10         c.arriveAndAwaitAdvance();
11         a[i] = (l + r) / 2;
12         c.arriveAndAwaitAdvance();
13       }
14       c.arriveAndDeregister();b.arriveAndDeregister();
15   }}.start();
16 }
17 b.arriveAndAwaitAdvance();
18 handle(a);
```

Figure 2: The example implemented using Java phasers.

2.2 Generalised barrier synchronisation using phasers

Phasers generalise barrier synchronisation by introducing the notion of barrier *phase*, allowing a task to await a future barrier step (*i.e.,* ahead of the other members), rather than just the immediately pending step. This abstraction was first introduced in HJ as an extension of X10 clocks. A limited form of phasers was later included in Java 7. Figure 2 lists a Java version of the running example, using the `java.util.concurrent.Phaser` API to represent both the cyclic and join barriers. The cyclic barrier is managed by the phaser assigned to `c`. The integer constructor argument (Line 1) creates the phaser with an initial count of pre-registered tasks for the first phase: here, the count, initialised to `1`, signifies the registration of the parent task with this phaser. On Line 4, each of the `I` tasks (threads) is registered with the `c`-phaser. Intuitively, a non-negative monotonic integer is assigned to each task, called the *local phase*, that is incremented when the task *arrives* at phaser; the phase is *observed* when all participants advance their local phase. Analogously to the X10 code, the cyclic barrier synchronisations are thus performed by each task invoking `arriveAndAwaitAdvance` on `c` on Lines 10 and 12 to arrive at and observe each synchronisation event.

The join barrier is managed by the phaser assigned to `b`. The join synchronisation is achieved by each task `i` invoking on Line 14 the non-blocking `arriveAndDeregister` method on `b` when finished, which the parent task observes by `arriveAndAwaitAdvance` on Line 17. Corresponding to Figure 1, this Java implementation will deadlock at the first `c`-phaser synchronisation because the registered parent task does not arrive at this event; the fix is to have the parent task do `c.arriveAndDeregister()` between Lines 16 and 17. Note that avoiding this deadlock by changing the code to simply not register the parent task with the `c`-phaser (*i.e.,* by setting its constructor argument to `0`) is not sufficient: in this case, the synchronisations on `c` would proceed non-deterministically between already running threads and those that have yet to be started.

Unlike in X10/HJ/MPI, Java phasers lack the information to identify which tasks are participating in a synchronisation, *e.g.*, method `Phaser.register` does not target a thread. To verify Figure 2 the programmer must insert the code `JArmus.register(c,b)` before Line 7.

HJ phasers permit tasks to await arbitrary phases, and split-phase synchronisation. The former enables collective producer-consumer synchronisations. The latter (also present in X10 and Java) enables the barrier synchronisation to be conducted over two steps: the task firstly initiates the synchronisation as a non-blocking background operation (*i.e.,* concurrently with the task), and can wait for the operation to conclude at a later point. By designing Armus to support HJ phasers, we subsume deadlock detection for X10

and Java barrier programs under one central abstraction. Works on phasers include synchronisation algorithms [27, 37], data-flow programming models [35, 36], and OpenMP extensions [38].

Event-based concurrency dependencies To be able to define the concurrency dependencies between tasks and phasers, state-of-the-art analysis uses information about the blocked tasks and the participants of each phaser. Instead, Armus defines these dependencies by reasoning about synchronisation events in the sense of Lamport logical clocks, a mechanism that can order events by associating a different timestamp (a monotonic integer) per event.

There is a natural representation between a phaser and a logical clock: when tasks synchronise on a phase number n of a phaser p each participant observes a synchronisation event that occurred at timestamp n of the logical clock associated with phaser p. Under this view, blocked tasks *wait* for a specific event to be observed. But since a waiting task cannot arrive at other registered phasers, then waiting tasks also *impede* the observation of events. Thus, any event that a tasks is waiting must *precede* an event the task impedes. A deadlock corresponds to any circular dependencies found in this ordering of events. Our novel representation dramatically improves the scalability of verification for two reasons.

1. Our representation has enough information to generate different graph models (Wait-For Graph and State Graph).
2. When performing distributed verification, the consistency of the dependencies is local to the task. Each site can independently collect the dependencies generated by each of its blocked task, which means that the various sites do not need to agree upon a certain global view, see Section 5.2.

Graph-based deadlock analysis Graph-based approaches perform cycle detection on the concurrency dependencies between tasks and synchronisation events. The Wait-For Graph (WFG) only models dependencies between tasks. The State Graph (SG) only models dependencies between synchronisation events. Since the scalability of cycle detection depends on the size of the graph, the ratio between the number of synchronisation events and the number of tasks impacts the best graph model choice. We discuss three scenarios of applications that use barrier synchronisation.

Parallel applications designed following the Single Program Multiple Data (SPMD) programming model share two characteristics: there is a fixed number of tasks and a fixed number of cyclic barriers throughout the whole computation; and the number of tasks is a parameter of the program, but the number of cyclic barriers is not. All of the benchmarks found in Section 6.1 share these characteristics. Scaling an SPMD program usually involves adding more tasks, whilst maintaining the same number of cyclic barriers; hence SG becomes beneficial at a larger scale.

The appropriate graph model for fork/join applications is harder to predict. For instance, in *nested* fork/join programming models, such as in X10, where join barriers (finishes) are lexically scoped, each task is registered with all join barriers that are enclosing its spawn location, *e.g.*, an X10 task spawned within the scope of three finishes is registered with three join barriers. The case complicates when join barriers are created dynamically in a recursive function call. For instance, languages with futures turn each function call into a join barrier, so it can happen that there are as many join barriers (resources) as there are tasks. In general, it is not possible to statically predict the ratio between resources and tasks in fork/join (and future) applications.

Java and X10 include multiple barrier abstractions to let applications choose from different programming models. Recent proposals of abstractions that use barrier synchronisation, in the context of X10 programming, make the case difficult for a fixed graph representation (be it the WFG or the SG). Atkins *et al.* design and implement *clocked variables* [4] that mediate the access of shared memory cells with barrier synchronisation in the context of X10. We benchmark three parallel algorithms that use clocked variables in Section 6.3 and the average edge count of each is different: in `SE` the edge count is similar between WFG and the SG; in `FI` the SG is on average twice as small; and in `FT` the average edge count of the WFG is ten times as small. Additionally, in the context of HJ, Shirako *et al.* propose using phasers for point-to-point synchronisation [34], so we expect the WFG to be more beneficial, and for the implementation of parallel reduction operations [35] that should favour the SG model.

3. PL: a core phaser-based language for general barrier synchronisations

This section introduces the syntax and semantics of a core language (PL) that abstracts user-level programs with barriers. The verification requires the state of the phaser data structure, and the set of blocked tasks to characterise a deadlock. The formalisation serves two purposes: defines the required information to characterise a deadlock, and precises the operations that change this information. Since our runtime verification works by sampling the state of phasers and blocked tasks, the analysis is oblivious to control flow mechanisms. Thus, language constructs that do not affect barrier synchronisation, like data manipulation, are left out or simplified, *e.g.*, the loop.

Phasers We first formalise the semantics of phasers in one predicate and three operations. Let P denote a *phaser* that maps *task names* $t \in \mathcal{T}$ into *local phases* $n \in \mathcal{N}$. Predicate $\mathrm{await}(P, n)$, used by tasks to observe a phaser, holds if every member of the phaser has a local phase of at least n.

$$\forall t \in \mathrm{dom}(P)\colon P(t) \geq n \implies \mathrm{await}(P, n)$$

We define the operational semantics for phasers in Figure 4. Three operations ϕ mutate a phaser: $\mathrm{reg}(t, n)$ adds a new member task t whose local phase is n; $\mathrm{dereg}(t)$ revokes the membership of t; and $\mathrm{adv}(t)$ increments the local phase of t.

Let M map phaser names $p \in \mathcal{P}$ to phasers, used to represent all phasers in the system. There are two operators o over phaser maps: $p := P$ that creates a phaser P named p, and $p.\phi$ that manipulates the phaser named p.

Syntax PL abstracts a user-level program as an instruction sequence s defined with instructions c, generated by the grammar:

$$s ::= c; s \mid \texttt{end}$$
$$c ::= t = \texttt{newTid}() \mid \texttt{fork}(t)\ s \mid p = \texttt{newPhaser}() \mid \texttt{reg}(t, p) \mid \texttt{dereg}(p) \mid \texttt{adv}(p) \mid \texttt{await}(p) \mid \texttt{loop}\ s \mid \texttt{skip}$$

We explain the syntax and its informal semantics by giving, in Figure 3, the PL representation of the running example, from Figure 1. Launching a task encompasses two instructions: create task name t with `newTid` (Line 4), and then `fork` a task t whose body is an instruction sequence s (Lines 6 to 15).

On task membership we have: `newPhaser` that creates a phaser and registers the current task at phase zero; `reg` that registers task t with a phaser p (the registered task in the parameter inherits the phase number of the current task); and `dereg` that deregisters the current task from phaser p. In our example the driver task creates a join barrier p_b in Line 2, registers worker tasks t with p_b in Line 5, which deregister from it to signal task termination in Line 14. For synchronisation we have phase advance with instruction `adv`, and `await` to wait for the phase the current task is registered with. While averaging the array each task advances its phase and then awaits the others to do the same in Lines 9 and 11.

Finally, regarding control flow we have `skip` that does nothing (used to represents data-related operations), and `loop` that unfolds

```
1  p_c = newPhaser();
2  p_b = newPhaser();
3  loop
4    t = newTid();
5    reg(p_c, t); reg(p_b, t);
6    fork(t)
7      loop
8        skip;
9        adv(p_c); await(p_c); // Cyclic barrier steps
10       skip;
11       adv(p_c); await(p_c);
12     end;
13     dereg(p_c);
14     dereg(p_b); // Notify finish
15   end;
16 end;
17 adv(p_b); await(p_b); // Join barrier step
18 skip;
19 end
```

Figure 3: PL for the example in Figure 1.

its body an arbitrary number of times (possibly zero), capturing while-loops, for-loops, and conditional branches. In Lines 8 and 10 we abstract the reading and writing to shared memory with the `skip`. In Lines 3 to 16 we abstract the for-loop to spawn tasks, and in Lines 7 to 12 we abstract the for-loop to average the array.

PL semantics We define the operational semantics for PL in **Instructions** and **States** in Figure 4. The rules for instruction sequences (`skip` and `loop`) are standard. A state $S ::= (M, T)$ of the system pairs phaser maps M with task maps T. A task map $T ::= \{t_1 : s_1, \ldots, t_m : s_m\}$ captures the state of the running tasks: it relates task t_i to instruction sequence s_i. Given any map, say X, we write $\mathrm{dom}(X)$ for the domain of X. When $\mathrm{dom}(X) \cap \mathrm{dom}(Y) = \emptyset$, we write $X \uplus Y$ for the disjoint union of X and Y.

In the context of dynamic verification, the semantics of PL plays two roles. First, with rule [sync] we can define the notion of blocked tasks, which allows us to characterise deadlocked states and establish the results in Section 4.3. Second, the remaining rules serve as a specification of how to maintain the status of phasers when verifying X10 and Java applications, see Section 5.3.

Definition 3.1 (Deadlocked state)**.** *State $(M, T' \uplus T)$ is* deadlocked *on task map T if, and only if, state (M, T) satisfies, for all $t \in \mathrm{dom}(T)$ we have that $T(t) = \texttt{await}(p); s$, $M(p)(t) = n$, and there is a task $t' \in \mathrm{dom}(T)$ where $M(p)(t') < n$.*

4. Deadlock verification algorithm

We adapt the classical notion of resource [19] to a phase in PL and use two alternative graph models to analyse concurrency constraints: Wait-For Graph [20] (WFG) and the State Graph [19] (SG).

The algorithm consists of three steps. First, by abstracting a PL state as a resource-dependency state, which expresses the dependencies between tasks and resources. Second, by translating this resource-dependency state into a WFG, or a SG. Third, by applying the standard cycle detection on the resulting graph.

4.1 Resource-dependency state construction

A resource-dependency state D consists of a pair (I, W). The map of impeding tasks I maps resources to the set of task names that impede synchronisation; in the case of barriers this set denotes the tasks that have not arrived at the barrier. The map of waiting resources W maps task names to the set of resources the task is blocked on. In PL, tasks can only await on a single phaser so we get singleton sets.

Phasers

$$\frac{\exists t' : P(t') \leq n}{P \xrightarrow{\mathrm{reg}(t,n)} P \uplus \{t : n\}} \quad [\text{reg}]$$

$$P \uplus \{t : n\} \xrightarrow{\mathrm{dereg}(t)} P \quad [\text{dereg}]$$

$$P \uplus \{t : n\} \xrightarrow{\mathrm{adv}(t)} P \uplus \{t : n+1\} \quad [\text{adv}]$$

Phaser maps

$$M \xrightarrow{p:=P} M \uplus \{p : P\} \quad [\text{new-p}]$$

$$\frac{P \xrightarrow{\phi} P'}{M \uplus \{p : P\} \xrightarrow{p.\phi} M \uplus \{p : P'\}} \quad [\text{new-t}]$$

Instructions

$$\texttt{skip}; s \to s \quad [\text{skip}]$$

$$\frac{s' = c_1; ..; c_n; \texttt{end}}{\texttt{loop}\ s'; s \to c_1; ..; c_n; (\texttt{loop}\ s'; s)} \quad [\text{i-loop}]$$

$$\texttt{loop}\ s'; s \to s \quad [\text{e-loop}]$$

States

$$\frac{t'' \notin \mathrm{fv}(s)}{(M, T \uplus \{t : t' = \texttt{newTid}(); s\}) \to (M, T \uplus \{t : s[t''/t']\} \uplus \{t'' : \texttt{end}\})} \quad [\text{new-t}]$$

$$(M, T \uplus \{t : \texttt{fork}(t')\ s'; s\} \uplus \{t' : \texttt{end}\}) \to (M, T \uplus \{t : s\} \uplus \{t' : s'\}) \quad [\text{fork}]$$

$$\frac{M \xrightarrow{q:=P} M' \qquad P = \{t : 0\} \qquad q \notin \mathrm{fv}(s)}{(M, T \uplus \{t : p = \texttt{newPhaser}(); s\}) \to (M', T \uplus \{t : s[q/p]\})} \quad [\text{new-ph}]$$

$$\frac{M(p)(t) = n \qquad M \xrightarrow{p.\mathrm{reg}(t',n)} M'}{(M, T \uplus \{t : \texttt{reg}(t', p); s\}) \to (M', T \uplus \{t : s\})} \quad [\text{reg}]$$

$$\frac{M \xrightarrow{p.\mathrm{dereg}(t)} M'}{(M, T \uplus \{t : \texttt{dereg}(p); s\}) \to (M', T \uplus \{t : s\})} \quad [\text{dereg}]$$

$$\frac{M \xrightarrow{p.\mathrm{adv}(t)} M'}{(M, T \uplus \{t : \texttt{adv}(p); s\}) \to (M', T \uplus \{t : s\})} \quad [\text{adv}]$$

$$\frac{M(p)(t) = n \qquad \mathrm{await}(P, n)}{(M, T \uplus \{t : \texttt{await}(p); s\}) \to (M, T \uplus \{t : s\})} \quad [\text{sync}]$$

$$\frac{s \to s'}{(M, T \uplus \{t : s\}) \to (M, T \uplus \{t : s'\})} \quad [\text{c-flow}]$$

Figure 4: Operational semantics of PL.

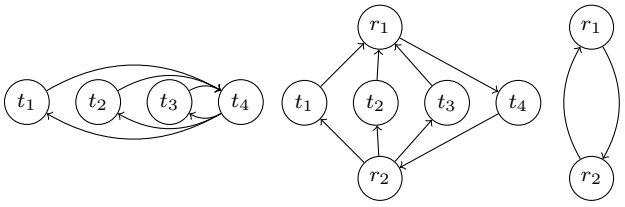
Example 4.1 (Resource dependency)**.** Consider the deadlocked state (M_1, T_1) defined below, derived from the running example considering `I` to be 3. Tasks t_1, t_2, and t_3 are the worker tasks blocked at the cyclic barrier p_c. Driver task t_4 is at the join barrier p_b.

$$M_1 = \{p_c : \{t_1 : 1, t_2 : 1, t_3 : 1, t_4 : 0\}, p_b : \{t_1 : 0, t_2 : 0, t_3 : 0, t_4 : 1\}\},$$
$$T_1 = \{t_1 : \texttt{await}(p_c); s_1, t_2 : \texttt{await}(p_c); s_2,$$
$$t_3 : \texttt{await}(p_c); s_3, t_4 : \texttt{await}(p_b); s_4\}$$

To construct a resource-dependency (I_1, W_1) from (M_1, T_1) we look into the tasks, to identify the resources. Let resource r_1 represent awaiting on phaser p_c at phase 1 and resource r_2 represent awaiting on phaser p_b at phase 1. Hence,

$$W_1 = \{t_1 : \{r_1\}, t_2 : \{r_1\}, t_3 : \{r_1\}, t_4 : \{r_2\}\}$$

To construct the structure of impeding tasks we inspect the phaser map according to each resource (r_1 and r_2). Task t_4 impedes any



(a) WFG (b) GRG (c) SG

Figure 5: Graphs of concurrency constraints in Example 4.1.

task blocked on resource r_1 (phaser p_c at phase 1). Similarly, tasks t_1, t_2, and t_3 impede task t_4 via resource r_2 (phaser p_b at phase 1), since the former are registered with a phase below 1.

$$I_1 = \{r_1 \colon \{t_4\}, r_2 \colon \{t_1, t_2, t_3\}\}$$

Below we define this notion: let res be a bijective function that maps resources r to pairs of phaser names and naturals (the phase).

Definition 4.1 (Resource-dependency)**.** *Let φ be a function from states into resource-dependencies, where $\varphi(M,T) \stackrel{\text{def}}{=} (I,W)$ is defined as:*

$$W \stackrel{\text{def}}{=} \{t \colon \{\text{res}(p,n)\} \mid \forall t \in \mathcal{T} \colon T(t) = \texttt{await}(p); s \wedge M(p)(t) = n\}$$

$$I \stackrel{\text{def}}{=} \{\text{res}(p,n) \colon R \mid \forall t' \in \mathcal{T} \colon T(t') = \texttt{await}(p); s \wedge M(p)(t') = n\}$$

$$\textit{where } R = \{t \mid \forall t \in \mathcal{T} \colon M(p)(t) < n\}$$

By Definition 4.1, we have that $\varphi(M_1,T_1) = (I_1,W_1)$.

We can view a resource-dependency as a graph by simply considering nodes to be tasks and resources, and use W and I to create the edges. The General Resource Graph [16] (GRG) models dependencies between tasks and resources. For each task t_1 that waits on a resource r_1, that is $r_1 \in W(t_1)$, then we have an edge (t_1, r_1). For each resource r_1 that impedes task t_4, that is $t_4 \in I(r_1)$, then we have an edge (r_1, t_4). Figure 5b depicts the GRG for resource-dependency (I_1, W_1).

WFG and SG can be obtained from a GRG by simple edge contraction. We give an intuition on the construction of the WFG and of the SG. The WFG is task-centric, so an edge (t_1, t_4) represents that task t_1 waits for task t_4 to synchronise, meaning that there exists a resource r_1 such that $r_1 \in W(t_1)$ and $t_4 \in I(r_1)$. Figure 5a illustrates the WFG for resource-dependency (I_1, W_1). The SG is resource-centric, so an edge (r_1, r_2) represents that resource r_1 impedes any task from synchronising via resource r_2, meaning that there exists a task t_4 such that $r_1 \in W(t_4)$ and $t_4 \in I(r_2)$. Figure 5c depicts the SG for resource-dependency (I_1, W_1).

4.2 WFG and SG construction

Graph analysis is the last step of verification. The resource-dependency state serves as a general representation of concurrency constraints, translatable to WFG and to SG.

First, some notions about graph theory [6]. A (directed) *graph* $G = (V, E)$ consists of a nonempty finite set of vertices V (where $r \in V$), and of a finite set of edges E (where $e \in E$). An *edge* $e = (r, r')$ is directed from the head r to the tail r'.

Graph (V, E) is a *subgraph* of graph (V', E') if (i) $V \subset V'$, (ii) $E \subset E'$, and (iii) $\forall (r, r') \in E \implies r \in V \wedge r' \in V$. A *walk* w on (V, E) is an alternating sequence $r_1 \cdots r_{n-1} r_n$ of vertices $r_i \in V$ such that $n > 1$ and $(r_i, r_{i+1}) \in E$ for $i = 1, \ldots, n-1$. We may specify the first and last vertices of a walk by saying a r-r' walk, for the walk $r \cdots r'$. A *cycle* is a walk $r \cdots r'$ where $r = r'$. We may specify the first and last vertex of a cycle by saying a r-cycle, for the cycle $r \cdots r$. The *length* of a walk corresponds to the number of its edges. We say that $r \in w$ if, and only if, $w = r_1 \cdots r_n$ and there exists a r_i such that $r = r_i$ and $1 \leq i \leq n$. We say that $(r, r') \in w$ if, and only if, $w = r_1 \cdots r_n$ and there exists a r_i and r_{i+1} such that $r = r_i$, $r' = r_{i+1}$, and $1 \leq i < n$.

The in-degree n of a vertex r counts the number of edges whose tail is r. The out-degree n of a vertex r counts the number of edges whose head is r. We say that vertex r' is *reachable* from r, or vertex r *reaches* r', if there exists a r-r' walk on graph G.

Next we formalise the notions of constructing a WFG and an SG from a resource-dependency.

Definition 4.2 (WFG construction)**.** *Let* wfg *be a function from resource-dependencies into WFG's:*

$$\text{wfg}(I,W) \stackrel{\text{def}}{=} \big(\mathcal{T}, \{(t_1,t_2) \mid \forall t_1, r, t_2 \colon r \in W(t_1) \wedge t_2 \in I(r)\}\big)$$

Formula $\text{wfg}(I_1, W_1)$ yields the graph in Figure 5a:

$$\big(\mathcal{T}, \{(t_1,t_4),(t_2,t_4),(t_3,t_4),(t_4,t_1),(t_4,t_2),(t_4,t_3)\}\big)$$

Definition 4.3 (SG construction)**.** *Let* sg *be a function from resource-dependencies into SG's:*

$$\text{sg}(I,W) \stackrel{\text{def}}{=} \big(\mathcal{R}, \{(r_1,r_2) \mid \forall t, r_1, r_2 \colon t \in I(r_1) \wedge r_2 \in W(t)\}\big)$$

We apply Definition 4.3 and get the graph in Figure 5c:

$$\text{sg}(I_1,W_1) = \big(\mathcal{R}, \{(r_1,r_2),(r_2,r_1)\}\big)$$

To prove the equivalence in finding a cycle in the WFG and finding a cycle in the SG, we define the GRG, that bridges the WFG and the SG.

Definition 4.4 (GRG construction)**.** *Let* grg *be a function from resource-dependencies into GRG's:*

$$\text{grg}(I,W) \stackrel{\text{def}}{=} \big(\mathcal{R} \cup \mathcal{T}, \{(t,r) \mid \forall t, r \colon r \in W(t)\} \cup \{(r,t) \mid \forall t, r \colon t \in I(r)\}\big)$$

Formula $\text{grg}(I_1, W_1)$ yields the graph in Figure 5b:

$$\big(\mathcal{R} \cup \mathcal{T}, \{(t_1,r_1),(t_2,r_1),(t_3,r_1),(t_4,r_2),(r_1,r_4),(r_2,r_1),(r_2,r_2),(r_2,r_3),\}\big)$$

4.3 Complexity and correctness of the deadlock verification algorithm

We motivate the importance of selecting from alternative graph models by discussing various synchronisation scenarios. To this end we introduce the complexity of graph analysis with the WFG and with the SG. We conclude the section by establishing the main results of our paper.

Proposition 4.2 (Complexity)**.** *Given a resource dependency state (I,W), a cycle detection based on WFG is O($|W|^2 + |W|$), while a cycle detection based on SG is O($|I|^2 + |I|$).*

Proof. A cycle detection in a graph has a complexity of O($e + v$) [40], for a graph with e edges and v vertices. We know that [6] for any graph $e \leq v^2$, thus we can simplify the complexity to O($v+v^2$). Since the WFG vertices are tasks, a deadlock verification that uses a WFG over a resource-dependency state with $|W|$ tasks has a complexity of O($|W|^2 + |W|$). Similarly, for the SG, we have O($|I|^2 + |I|$). □

The correctness of our verification implementation, which mixes both the WFGs and SGs (see Section 6.3), requires three theorems: equivalence, soundness, and completeness. The first theorem shows the WFG and the SG are interchangeable. The second

and third theorems state soundness and completeness of the deadlock verification. Soundness means that a cycle detection based on a WFG corresponds to a deadlocked PL state; completeness means that every deadlocked state yields a WFG with a cycle. ***The full version of this paper includes the full proofs [3, 8].***

Theorem 4.5 (Equivalence between WFG and SG)**.** *There exists a cycle w on graph* $\mathrm{wfg}\,(D)$ *if, and only if, there exists a cycle w' on graph* $\mathrm{sg}\,(D)$.

We note that, although the correctness of our approach and implementation only immediately concerns the relationship between WFGs and SGs, the proof of Theorem 4.5 [8] involves an extended property: a cycle in the WFG (resp. SG) implies a cycle in the GRG, which in turns implies a cycle in the SG (resp. WFG).

Theorem 4.6 (Soundness)**.** *If w is closed on* $\mathrm{wfg}\,(\varphi\,(M,T))$ *with a positive length, then there exists task map T' and T'' such that $T = T' \uplus T''$,* $\mathrm{dom}(T') = \{t \mid \forall t \in w\}$*, state (M,T) is deadlocked on T'.*

Theorem 4.7 (Completeness)**.** *If state S is deadlocked on T and $t \in \mathrm{dom}(T)$, then there exists a t'-cycle on* $\mathrm{wfg}\,(\varphi\,(S))$ *such that t' is reachable from t.*

5. The Armus tool

The architecture of Armus is divided into two layers: the *application layer* generates concurrency constraints for each task, and the *verification layer* that manages the resource-dependency state and checks for deadlocks. The application layer is specific to each language we check.

Our verification algorithm can be used to *avoid* and to *detect* deadlocks. In the former, Armus checks for deadlocks before the task blocks and interrupts the blocking operation with an exception if the deadlock is found. The programmer can treat the exceptional situation to develop applications resilient to deadlocks. In the latter, verification is performed periodically and can only report already existing deadlocks, with the benefit of a lower performance overhead.

5.1 Resource-dependency deadlock verification library

Armus' deadlock verification library implements the theory described in Section 4.2. The main features of the library are (i) a deadlock detection algorithm that is fault-tolerant and distributed; and (ii) a scalable deadlock verification technique (*i.e.,* the adaptive graph representation).

Essentially, whenever a task of the program blocks the application layer invokes the verification library by producing its blocked status: a set of waiting $W(t)$ and set of impeding resources $\{r \mid \forall r\colon t \in I(r)\}$. The library is divided into two services: an implementation of the resource-dependencies; and the *deadlock checker* that analyses the resource-dependencies for deadlocks, using Definition 4.2 and Definition 4.3. Maintaining the blocked status is more frequent than checking for deadlocks, so the resource-dependencies are rearranged per task to optimise updates. The deadlock checker internally transforms the dependencies into a graph and then performs cycle detection with JGraphT [18].

The verification library provides two graph selection modes: fixed or automatic. In the former, the verification always uses the same graph model. State-of-the-art tools are fixed to the WFG model. In the automatic mode, the verification library selects the graph model according to the ratio between blocked tasks and registered phasers. This means that the graph model used for cycle detection can change over time.

We briefly describe the implementation of each mode. In the *fixed to WFG mode* (see Definition 4.2), the algorithm iterates over a copy of the blocked tasks twice. First, uses the impeding resource of each blocked task to construct map I. Second, generates a WFG-edge from each waiting resource r to each task in $I(r)$. In the *fixed to SG mode* (see Definition 4.3), it iterates over each blocked task (available in the resource-dependencies) and generates an SG-edge from each impeding resource to each blocked resource. The *adaptive mode* tries to build an SG first; if during the construction of the SG it reaches a size threshold, then it builds a WFG instead. The size threshold is reached if at any time there are more SG-edges than twice the number of tasks processed thus far. The value of the threshold was obtained based on experiments on the available benchmarks.

5.2 Distributed deadlock detection

Armus adapts the traditional one-phase deadlock detection [21] to barrier synchronisation and introduces support for fault tolerance. A distributed program is composed of various *sites* that communicate among each other, each runs an instance of Armus that has remote access to a global resource-dependency (implemented as a data store server Redis [31]). The various instances of Armus periodically update a disjoint portion of the global resource-dependency with the contents of their local resource-dependencies. The deadlock checker requires a global view of the system, so it operates on the blocked status of the global resource-dependency.

There are two differences with reference to the original algorithm in [21]. Armus uses logical clocks to represent barrier synchronisations and maintain global data consistency; the original algorithm requires vector clocks to represent lock synchronisations. For fault-tolerance concerns, the global status of Armus is maintained in a dedicated server, and all sites check for deadlocks. In contrast, in [21] there is a designated control site that collects the global status and performs graph analysis. Our benchmarks, in Section 6.2, show that the verification overhead has a negligible impact for 64 tasks.

The verification algorithm is fault-tolerant, since it continues executing despite (i) site-failures and (ii) data store-failures. Such feature is of special interest for checking fault-tolerant applications, like Resilient X10 [10]. The algorithm resists (i) because the deadlock checker executes at each site and does not depend on the cooperation of other sites to function. The algorithm resists (ii) because Redis itself is fault-tolerant.

5.3 Verifying barrier deadlocks in X10 and in Java

We present two verification applications to check for barrier deadlocks: JArmus for Java programs and Armus-X10 for X10 programs. These tools work by "weaving" the verification into programs. The input is a compiled program to be verified (Java bytecode); the output is a verified program (Java bytecode) that includes dynamic checks for deadlock verification. JArmus and Armus-X10 layers implement the resource-dependency construction from Section 4.1.

JArmus and Armus-X10 share the same usage and design. The implementation of each of these verification tools is divided into two components: the resource mapper and the task observer. The resource mapper converts synchronisation events to resources. The task observer intercepts blocking calls to inform Armus that the current task is blocked with a set of resource edges.

Armus-X10 Armus-X10 can verify any program written in X10 that uses: clocks, finishes, and the `SPMDBarrier`; the tool can verify distributed applications. Unlike in Java, automatic instrumentation is possible. The X10 runtime provides information about the registered clocks and registered finishes of a given task, which is required to construct the concurrency dependencies of each task. X10 can be compiled to Java bytecode, called Managed X10, and

to machine code, called Native X10. Currently, our application only supports Managed X10.

JArmus JArmus supports `CountDownLatch`, `CyclicBarrier`, `Phaser`, and `ReentrantLock` class operations of the standard Java API. Unlike in X10 and HJ, the relationship between the participants of barrier synchronisation and tasks in Java is left implicit. For example, when using a `CyclicBarrier` the programmer declares the number of participants and then shares the object with those many tasks, but it is not specified which tasks participate in the synchronisation. This missing information, which the Armus analysis requires, is also necessary to extend the Java implementation of phasers to support the full range of features in the original phaser semantics [34]. For instance, only by knowing exactly which tasks are participants can phasers allow some tasks to advance without waiting. Since JArmus has no way of reconstructing this information for the `CountDownLatch`, `CyclicBarrier`, and `Phaser` classes, then the programmer must annotate its code to supply the barriers each task is registered with. Each task, upon starting up, must invoke `JArmus.register(b)` per barrier `b` it uses (similarly to the X10 `clocked`). Instances of the class `ReentrantLock` do not require annotations.

6. Evaluation

The aim of the evaluation process is to 1) ascertain whether the performance impact of Armus scales with the increase in the number of tasks, 2) evaluate the performance overhead of distributed deadlock detection, and 3) compare execution impact the SG with the WFG and with adaptive approach.

The hardware used to run the benchmarks has four AMD Opteron 6376 processors, each with 16 cores, making a total of 64 cores. There are 64GB of available RAM. The operating system used is Ubuntu 13.10. For the languages, we used Java build 1.8.0_05-b13, and X10 version 2.4.3. For compiling and running we used the defaults flags with no additional parameters, except in the case of the NPB suite that is compiled with `-O`.

We follow the *start-up performance* methodology detailed in [12]. We take 31 samples of the execution time of each benchmark and discard the first sample. Next, we compute the mean of the 30 samples with a confidence interval of 95%, using the standard normal z-statistic.

6.1 Impact of non-distributed verification

The two goals of this evaluation are: to measure the impact of verification on standard Java benchmarks, and ii) to measure whether the verification scales with the increase of the number of tasks. We run the verification algorithm against a set of standard parallel benchmarks available for Java. JArmus is run in the detection mode (every 100 milliseconds) and in the avoidance mode, both use the adaptive graph model. Note that the Java applications we checked are not distributed.

We select benchmarks from the NASA Parallel Benchmark (NPB) suite [11] and the Java Grande Forum (JGF) [39] benchmark suite. The NPB ranges from kernels to pseudo-applications, taken primarily from representative Computational Fluid Dynamics (CFD) parallel applications. The JGF is divided into three groups of applications: micro-benchmarks, computational kernels, and pseudo-applications. All benchmarks proceed iteratively, and use a fixed number of cyclic barriers to synchronise stepwise. Furthermore, all benchmarks check the validity of the produced output.

For the sake of reproducibility we list the parameters of the benchmarks run as specified in [11, 39]: `BT` uses size A, `CG` uses size C, the Java version of `FT` uses size B, `MG` uses size C, `RT` uses B, and `SP` uses size W. The input set chosen for benchmark `SP` only allows it to scale up to 31 tasks. For simplicity, in the evaluation we consider that this benchmark scales up to 32 tasks.

Table 1: Relative execution overhead in detection mode.

Threads	**2**	**4**	**8**	**16**	**32**	**64**
`BT`	3%	-4%	0%	-5%	0%	7%
`CG`	7%	0%	7%	15%	12%	9%
`FT`	1%	0%	-1%	-7%	0%	0%
`MG`	-5%	0%	0%	0%	11%	13%
`RT`	-4%	0%	0%	0%	0%	8%
`SP`	-1%	4%	4%	2%	0%	

Table 2: Relative execution overhead in avoidance mode.

Threads	**2**	**4**	**8**	**16**	**32**	**64**
`BT`	5%	0%	0%	0%	11%	8%
`CG`	0%	9%	20%	34%	46%	50%
`FT`	1%	4%	0%	0%	7%	25%
`MG`	8%	7%	21%	27%	27%	30%
`RT`	-5%	0%	0%	0%	5%	16%
`SP`	2%	9%	8%	22%	28%	

Figure 6 summarises the comparative study of the execution time for each benchmark. Tables 1 and 2 list the relative runtime overhead in detection and in avoidance. The results for the NPB and JGF benchmark suites are depicted in Figures 6a to 6f. In detection mode, since there is a dedicated task to perform verification, we observe that the overhead does not increase linearly as we add more tasks. The relative runtime overhead sits below 15% and in most cases is negligible. In avoidance mode, each task checks the graph whenever it blocks, so as we add more tasks, the execution overhead increases. Still, in the worst case, benchmark `CG`, the overhead is 50%, which is acceptable for application testing purposes.

6.2 Impact of distributed verification

The goal of the evaluation is to measure the runtime overhead of deadlock detection in available X10 distributed applications. Armus-X10 is configured with the distributed deadlock detection mode, running the verification algorithm every 200 milliseconds. The chosen benchmarks are available via the X10 source code repository [45]. Deadlock avoidance is unavailable in the distributed setting.

Benchmarks `FT` and `STREAM` come from the HPC Challenge benchmark [26], `SSAC2` is an HPCS Graph Analysis Benchmark [5], `JACOBI` and `KMEANS` are available from the X10's website. For reproducibility purposes the non-default parameters we select are: `FT` magnitude 11; `KMEANS` 25k points, 3k clusters to find, and 5 iterations; `JACOBI` matrix of size 40, maximum iterations are 40; `SSCA2` 2^{15} vertices, a with a probability of 7%, and no permutations; `STREAM` with size of 524k.

Figure 7 depicts the execution time of each benchmark with and without verification. There is no statistical evidence of an execution overhead with running deadlock detection mode.

6.3 Impact of the graph model choice

The goal of this evaluation is to measure the impact of the graph model in the verification procedure. To this end we analyse the worst case behaviour: programs that generate graphs with thousands of edges. In particular, we evaluate our adaptive model selection against the usual fixed model selection (WFG and SG).

We select a suite of programs that spawn tasks and create barriers as needed, depending on the size of the program, unlike the classical parallel applications we benchmark in Sections 6.1 and 6.2 where the number of tasks should correspond to the number of available processing units (cores). The suite of programs exercises

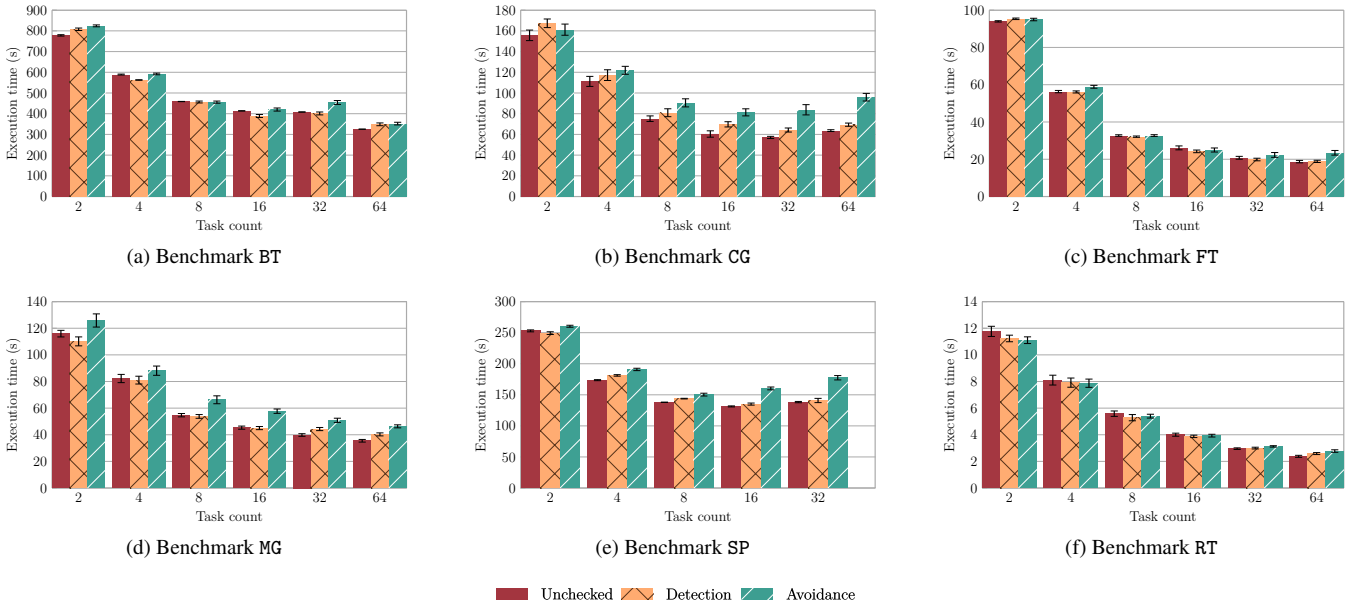


Figure 6: Comparative execution time for non-distributed benchmarks (lower means faster).

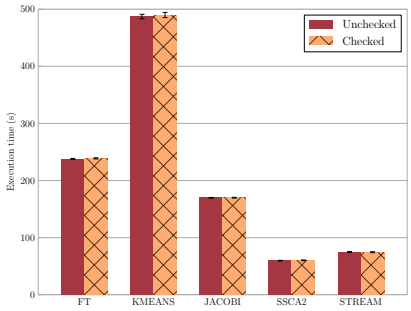


Figure 7: Comparative execution time for distributed deadlock detection (lower means faster).

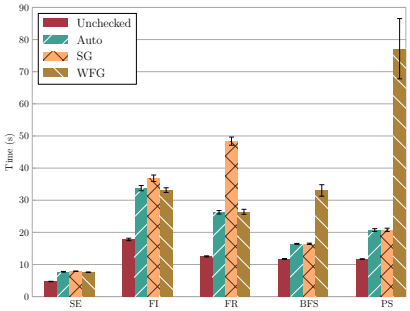


Figure 8: Comparative execution time for different graph model choices (lower means faster), using deadlock avoidance.

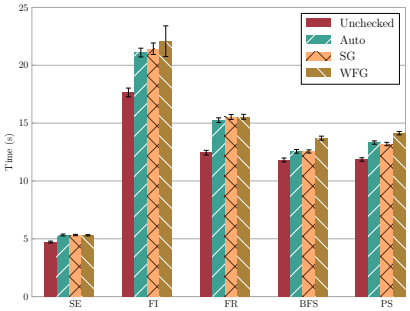


Figure 9: Comparative execution time for different graph model choices (lower means faster), using deadlock detection.

different worst case scenarios for the verification algorithm: many tasks *versus* many barriers.

The chosen benchmarks are educative programs taken from the course on *Principles and Practice of Parallel Programming*, taught by Martha A. Kim and Vijay A. Saraswat, Fall 2013 [46]. `BFS` performs a parallel breadth-first search on a randomly generated graph. There is a task per node being visited and a barrier per depth-level of the graph. `FI` computes a Fibonacci number iteratively with a shared array of *clocked variables* (each pairs a barrier with a number). Each element of the array holds the outcome of a Fibonacci number. When the program starts it launches n tasks. The i-th task stores its Fibonacci number in the i-th clocked variable and synchronises with task $i+1$ and task $i+2$ that read the produced value. `FR` computes a Fibonacci number recursively. Recursive calls are executed in parallel and a clocked variable synchronises the caller with the callee. `SE` implements the Sieve of Eratosthenes using clocked variables. There is a task per prime number and one clocked variable per task. `PS` computes the prefix sum—or cumulative sum—for a given number of tasks. Given an input array with as many elements as there are tasks, the outcome of task i is the partial sum of the array up to the i-th element. All tasks proceed stepwise and are synchronised by a global barrier.

Figures 8 and 9 depict the execution time of each benchmark verified by Armus-X10 in avoidance and detection modes (respectively) where we vary the selection method of the graph model. Table 3 lists the average number of edges used in verification and the relative execution time overhead of each benchmark.

We can classify the benchmarks in three groups according to the ratio between the number of tasks and the number of resources: i) similar count of tasks and resources, benchmark `SE`; ii) much more resources than tasks, benchmarks `FI` and `FT`; and iii) much more tasks than resources, benchmarks `BFS` and `PS`. When i) there are as many resources as there are tasks, then all graph models perform equally well. When ii) there are more resources than tasks, and iii) vice-versa, the choice of the graph model is of major importance for a verification with low impact on the execution time.

Even in the worst case behaviour for analysis the largest verification overhead with deadlock detection is 25%; for deadlock avoidance the largest is 117%. For both cases we consider adaptive graph selection. Overall, the approach of the adaptive graph model outperforms the fixed graph model approach. The adaptive approach can save up to 9% of execution overhead in deadlock detection *versus* a fixed model. The graph model choice severely amplifies the verification overhead in deadlock avoidance. The case

Table 3: Edge count and verification overhead per benchmark per graph mode.

		SE	FI	FR	BFS	PS
Auto	Edges	23	1074	140	5	7
	Avoidance	75%	94%	117%	45%	82%
	Detection	25%	24%	25%	9%	18%
SG	Edges	51	2137	1643	7	6
	Avoidance	75%	112%	300%	45%	82%
	Detection	25%	24%	25%	9%	18%
WFG	Edges	23	1281	94	579	781
	Avoidance	75%	94%	117%	200%	600%
	Detection	25%	29%	25%	18%	27%

in point is benchmark PS, where the verification overhead ranges from 600% (fixed) down to 82% (adaptive).

7. Related work

This section lists related work focusing on deadlock verification in parallel programming languages.

Deadlock prevention The literature around source code analysis to prevent barrier related deadlocks is vast. The fork/join programming model is easily restricted syntactically to prevent deadlocks from happening. Lee and Palsberg present a calculus for a fork/join programming model [24], suited for inter-procedural analysis through type inference, and establish a deadlock freedom property. The work also includes a type system that is used to identify may-happen-parallelism, further explored in [1].

Cogumbreiro *et al.* [9] propose a static typing system to ensure the correctness of phased activities for a fragment of X10 that disallows awaiting on a particular clock. Therefore, programs that involve more than one clock and that perform single waits cannot be expressed, nor verified (*cf.* the X10 and Java programs we present in Section 2).

Other works on the formalisation of barrier semantics are not concerned with deadlock-freedom. Saraswat and Jagadeesan [33] formalise the concurrency primitives of X10. Le *et al.* devise a verification for the correct use of a cyclic barrier in a fork/join programming language [23]. Vasudevan *et al.* [43] perform static analysis to improve performance of synchronisation mechanisms.

The tool X10X [13] is a *model checker* for X10. Model checkers perform source code analysis and can be used to discover potential deadlocks. This class of tools is affected by the state explosion problem: the analysis grows exponentially with the possible interleaves of the program. Thus, X10X may not be able to verify complex programs. In general, prevention is too limiting to be applied to the whole system, so language designers use this strategy to eliminate just a class of deadlocks.

Deadlock avoidance To our best knowledge, techniques that avoid deadlocks in the context of barrier synchronisation only handle a few situations of barrier deadlocks, unlike our proposal that is complete (with reference to Theorem 4.7). For instance, in X10 and HJ, tasks deregister from all barriers upon termination; this mitigates deadlocks that arise from missing participants. HJ avoids deadlocks that originate from the interaction between phasers and finish blocks by limiting the use of phasers to the scope of finish blocks.

Deadlock detection UPC-CHECK [32] deals with deadlock detection, but in a simpler setting where barriers are global; in contrast, our work can handle group synchronisation. Literature concerning MPI deadlock detection takes a top-down approach: the general idea is given, but mapping it to the actual MPI semantics is left out. DAMPI [44] reports a program as deadlocked after a period of inactivity, so it may indicate false positives, *i.e.,* it can misidentify a slow program as a deadlock. Umpire [14] and MUST [15] (a successor of Umpire) use a graph-based deadlock detection algorithm that subsumes deadlock detection to cycle detection, but omit a formal description on how the graph is actually generated from the language, *cf.* Theorems 4.6 and 4.7. We summarise the distributed detection technique of MUST. First, all sites collaborate to generate a single stream of events to a central site. The difficulty lays in ordering and aggregating the events generated by the various tasks. Then, the central site processes the stream of events to perform the *collective checking*, where, among other things, it identifies any completed barrier synchronisations. Finally, since MUST maintains a distributed *wait state*, then the site performing the collective checking must broadcast the status of terminated synchronisations back to the various sites of the application. The wait state is required to delay the graph analysis as much as possible. In our approach, tasks only requires local information to maintain data consistency, which means that, in a distributed setting, Armus does not require the last synchronisation step that MUST performs. Furthermore, unlike MUST, Armus is capable of verifying split-phase synchronisation, known in MPI as non-blocking collective operations.

8. Conclusion

We put forward Armus, a dynamic verification tool for barrier deadlocks that features distributed deadlock detection and a scalable graph analysis technique (based on automatic graph model selection). The target of verification is the core language PL, introduced to represent programs with various barrier synchronisation patterns. The graph-based deadlock verification of Armus is formalised and shown to be sound and complete against PL. We establish an equivalence theorem between utilising the graph models WFG and SG for deadlock detection; this result enables us to use the standard WFG to prove our results, and choose automatically between the WFG and the SG during verification. Our adaptive model selection dramatically increases the performance against the fixed model selection. The runtime overhead of the deadlock detection is low for up to 64 tasks, in most cases negligible. We present two applications: Armus-X10 monitors any unchanged X10 program for deadlocks; JArmus is a library to verify Java programs. To the best of our knowledge, our work is the first dynamic verification tool that can correctly detect Java and X10 barrier deadlocks.

For future work, we intend to verify HJ programs, as it will exercise the expressiveness of Armus. This language features abstractions with complex synchronisation patterns, such as the bounded producer-consumer. Another direction is the verification of MPI programs that introduce complex patterns of point-to-point synchronisation and enable a direct comparison with state-of-the-art in barrier deadlock detection.

Acknowledgements We thank Olivier Tardieu and PPoPP reviewers for their comments and suggestions. The work is partially supported by EPSRC KTS with Cognizant, EP/K034413/1, EP/K011715/1 and EP/L00058X/1, EU project FP7-612985 UpScale and ICT COST Action 1201 BETTY.

References

[1] S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel analysis of X10 programs. In *PPoPP'10*, pages 183–193. ACM, 2007.

[2] S. Agarwal, S. Joshi, and R. K. Shyamasundar. Distributed generalized dynamic barrier synchronization. In *ICDCN'11*, pages 143–154. Springer, 2011.

[3] Armus homepage. `bitbucket.org/cogumbreiro/armus/wiki/PPoPP15`.

[4] D. Atkins, A. Potanin, and L. Groves. The design and implementation of clocked variables in X10. In *ACSC'13*, pages 87–95. Australian Computer Society, 2013.

[5] D. A. Bader and K. Madduri. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. In *HiPC'05*, volume 3769 of *LNCS*, pages 465–476. Springer, 2005.

[6] J. Bang-Jensen and G. Z. Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer, 2nd edition, 2009.

[7] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: the new adventures of old X10. In *PPPJ'11*, pages 51–61. ACM, 2011.

[8] T. Cogumbreiro, R. Hu, F. Martins, and N. Yoshida. Dynamic deadlock verification for general barrier synchronisation. Technical Report DTR14-12, Imperial College London, 2014.

[9] T. Cogumbreiro, F. Martins, and V. T. Vasconcelos. Coordinating phased activities while maintaining progress. In *COORDINATION'13*, volume 7890 of *LNCS*, pages 31–44. Springer, 2013.

[10] D. Cunningham, D. Grove, B. Herta, A. Iyengar, K. Kawachiya, H. Murata, V. Saraswat, M. Takeuchi, and O. Tardieu. Resilient X10: Efficient failure-aware programming. In *PPoPP'14*, pages 67–80. ACM, 2014.

[11] M. A. Frumkin, M. Schultz, H. Jin, and J. Yan. Performance and scalability of the NAS Parallel Benchmarks in Java. In *IPDPS'03*. IEEE, 2003.

[12] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *OOPSLA'07*, pages 57–76. ACM, 2007.

[13] M. Gligoric, P. C. Mehlitz, and D. Marinov. X10X: Model checking a new programming language with an "old" model checker. In *ICST'12*, pages 11–20. IEEE, 2012.

[14] T. Hilbrich, B. R. de Supinski, M. Schulz, and M. S. Müller. A graph based approach for MPI deadlock detection. In *ICS'09*, pages 296–305. ACM, 2009.

[15] T. Hilbrich, J. Protze, M. Schulz, B. R. de Supinski, and M. S. Müller. MPI runtime error detection with MUST: advances in deadlock detection. In *SC'12*, pages 1–11. IEEE, 2012.

[16] R. C. Holt. Some deadlock properties of computer systems. *ACM Computing Surveys*, 4(3):179–196, Sept. 1972.

[17] Java 7 Phaser API. `docs.oracle.com/javase/7/docs/api/java/util/concurrent/Phaser.html`.

[18] JGraphT homepage. `jgrapht.org`.

[19] E. G. C. Jr., M. J. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys*, 3(2):67–78, 1971.

[20] E. Knapp. Deadlock detection in distributed databases. *ACM Computing Survey*, 19(4):303–328, 1987.

[21] A. D. Kshemkalyani and M. Singhal. Correct two-phase and one-phase deadlock detection algorithms for distributed systems. In *SPDP'90*, pages 126–129, 1990.

[22] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[23] D.-K. Le, W.-N. Chin, and Y.-M. Teo. Verification of static and dynamic barrier synchronization using bounded permissions. In *ICFEM'13*, volume 8144 of *LNCS*, pages 231–248. Springer, 2013.

[24] J. K. Lee and J. Palsberg. Featherweight X10: a core calculus for async-finish parallelism. In *PPoPP'10*, pages 25–36. ACM, 2010.

[25] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *OOPSLA'09*, pages 227–242. ACM, 2009.

[26] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi. The HPC Challenge (HPCC) benchmark suite. In *SC'06*. ACM, 2006.

[27] S. Marr, S. Verhaegen, B. D. Fraine, T. D'Hondt, and W. D. Meuter. Insertion tree phasers: Efficient and scalable barrier synchronization for fine-grained parallelism. In *HPCC'10*, pages 130–137. IEEE, 2010.

[28] Message Passing Interface (MPI) homepage. `mpi-forum.org`.

[29] M. T. O'Keefe and H. G. Dietz. Hardware barrier synchronization: Dynamic barrier MIMD (DBM). In *ICPP'90*, pages 43–46. Pennsylvania State University, 1990.

[30] OpenMP homepage. `openmp.org`.

[31] Redis homepage. `redis.io`.

[32] I. Roy, G. R. Luecke, J. Coyle, and M. Kraeva. A scalable deadlock detection algorithm for UPC collective operations. In *PGAS'13*, pages 2–15. The University of Edinburgh, 2013.

[33] V. Saraswat and R. Jagadeesan. Concurrent clustered programming. In *CONCUR'05*, volume 3653 of *LNCS*, pages 353–367. Springer, 2005.

[34] 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*, pages 277–288. ACM, 2008.

[35] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. Phaser accumulators: A new reduction construct for dynamic parallelism. In *IPDPS'09*, pages 1–12. IEEE, 2009.

[36] J. Shirako, D. M. Peixotto, D.-D. Sbîrlea, and V. Sarkar. Phaser beams: Integrating stream parallelism with task parallelism. Presented at the X10'11, 2011.

[37] J. Shirako and V. Sarkar. Hierarchical phasers for scalable synchronization and reductions in dynamic parallelism. In *IPDPS'10*, pages 1–12. IEEE, 2010.

[38] J. Shirako, K. Sharma, and V. Sarkar. Unifying barrier and point-to-point synchronization in OpenMP with Phasers. In *IWOMP'11*, volume 6665 of *LNCS*, pages 122–137. Springer, 2011.

[39] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel Java Grande benchmark suite. In *SC'01*. ACM, 2001.

[40] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[41] F. Turbak. First-class synchronization barriers. In *ICFP'96*, pages 157–168. ACM, 1996.

[42] UPC homepage. `upc-lang.org`.

[43] N. Vasudevan, O. Tardieu, J. Dolby, and S. A. Edwards. Compile-time analysis and specialization of clocks in concurrent programs. In *CC'09*, volume 5501 of *LNCS*, pages 48–62. Springer, 2009.

[44] A. Vo. *Scalable Formal Dynamic Verification of MPI Programs Through Distributed Causality Tracking*. PhD thesis, University of Utah, 2011. AAI3454168.

[45] X10 homepage. `x10-lang.org`.

[46] Course materials of *principles and practice of parallel programming*. `www.cs.columbia.edu/~martha/courses/4130/au13/`, 2013.