Armus: dynamic deadlock verification for barriers

Tiago Cogumbreiro[1] Raymond Hu[1] Francisco Martins[2] Nobuko Yoshida[1]

[1]Imperial College London [2]Universidade de Lisboa

Abstract

This paper presents a graph-based dynamic verification algorithm 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. The synchronisation patterns enabled by current barrier-based abstractions can introduce deadlocks, a major issue in getting parallel applications correct. Barrier deadlocks arise from a cyclic-dependency amongst tasks that participate on multiple barriers. We propose the use of a synchronisation-centric model instead a task-centric model, traditionally used in dynamic deadlock verification.

We present Armus, a platform-agnostic framework for deadlock verification and introduce two applications of this framework: Armus-X10 monitors any unchanged X10 program for deadlocks; JArmus is a library to verify Java programs. To evaluate Armus, we benchmark the runtime execution against the NAS Parallel Benchmark suite and Java Grade Forum Benchmark suite. Results show that the performance overhead of our deadlock detection technique is negligible for usual parallel applications and is independent of the number of the tasks.

1. Introduction

Barrier synchronisation coordinates the execution of independent processing units (*e.g.*, tasks, processors, or computers). The importance of this synchronisation mechanism can be traced back to the first parallel computers ever designed in the 1960's, *e.g.*, Gamma 60 [4].

Barriers are a cornerstone of parallel computing. Their presence is ubiquitous, ranging from hardware [1] to programming models. In particular, the *de facto* standards for shared memory (OpenMP [27]) and message passing (MPI [26]) paradigms are build upon this form of synchronisation. Java 5–8 and .NET 4 incorporate five abstractions based on barriers: latches, cyclic barriers, futures, a fork/join programming model, and stream programming. Futures are also a novelty of C++ version 2011. The cyclic barriers found in Java, are inspired by *clocks* of the X10 language [41], and by *phasers* of Habanero-Java (HJ) [6].

Current barrier-based abstractions enable new synchronisation patterns that can introduce deadlocks, a class of nefarious concurrency failures. Barrier deadlocks arise from a cyclic-dependency amongst tasks that participate on multiple barriers. Literature usually considers strategies to handle deadlocks [17]—prevention, avoidance, and detection.

The focus of this paper is on *dynamic verification*—deadlock avoidance and detection—where the runtime system monitors the application's state and deals with resolving the deadlock. The standard approach is to model any runtime dependencies between tasks and resource requests as a graph and subsume deadlock detection to some graph operation, like cycle detection. The requests of resources corresponds to blocking operations, such as awaiting at a barrier, or acquiring a lock.

When choosing a graph representation, there are three factors to consider: (1) the information stored in the graph, (2) the graph operation that represents the deadlock check, (3) the accuracy of the deadlock check. Each factor influences the other. Representing more (1) information in the graph, usually implies (3) having accurate checks, but slows down the (3) the dynamic checks. There are three main models to choose from: the Wait-For Graph (WFG) [20] that models dependencies (edges) between tasks (nodes), the State Graph (SG) [17] that models dependencies between requests (nodes), and the Resource-Allocation Graph (RAG) [28] that models dependencies between tasks (nodes) and resources (nodes). State-of-the-art in dynamic deadlocks verification for mutual exclusion [5] uses (1) the RAG, the check is (2) cycle detection on a specific path configuration, and (3) is not sound, so there are false positives. There are recent attempts to increase the accuracy of this form of verification [31]. Similarly, related work that checks MPI applications [13] uses (1) the WFG, the check is (2) knot detection in the graph, and (3) it is not sound. The SG is preferred over the WFG and the RAG for applications that need to scale on the number of tasks while the number of resources remains fixed (like HPC applications), as adding more tasks slows down verification. The SG makes the detection problem independent of the number of tasks running in the system, yet, surprisingly, literature surrounding the SG is scarce.

We address the accuracy and the scalability problems of dynamic verification of barrier deadlocks. In summary, our contributions are:

- a novel representation of barrier deadlocks that uses the SG, that subsumes the deadlock check to cycle detection,

```
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     }
12   }
13 } // Join barrier (finish) step: wait on all tasks
14 process(a);
```

Figure 1: Join and cyclic barrier synchronisation in X10.

and the check is sound and complete, *i.e.,* all identified deadlocks are true deadlocks, and all possible deadlocks are detected.

- the first dynamic deadlock verification of barrier deadlocks for Java and X10; and
- a performance evaluation using the NAS Parallel Benchmark [38] (for X10 and Java) and Java Grade Forum Benchmark [9] (§4).

The benchmark results show that Armus runtime overhead is negligible and independent of the number of tasks. Armus is available from [3] with detailed benchmark results and test cases.

The paper is organised as follows. Sections 2 to 4 present the contributions listed above. Section 5 discusses related work and Section 6 concludes the paper.

2. Barrier programs and deadlocks

This section firstly illustrates deadlock errors in two (bugged) implementations of a small parallel algorithm, using X10 barriers [41] and Java phasers [15]. Secondly, we give a more general summary of deadlock situations that may arise in various barrier usage patterns, with X10 examples. As Armus is based on the more general mechanism of phasers, which subsumes barriers, we are able to directly apply its implementation to both X10 and Java.

2.1 X10 barrier programming

The algorithm that we use as a running example has two stages. In the first stage (a simplified parallel 1-dimensional iterative averaging [34]), `I` parallel tasks work on an array `a` of `I+2` numbers. Each task is responsible for updating one of the middle `I` elements to the average of its neighbours repeatedly over a series of synchronised iterations. After these tasks have finished, a single task then performs some final processing on the resulting array values.

Figure 1 lists an X10 implementation. 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 the clock `c`; the parent task was implicitly registered when it created the clock. In the `async` body of each task `i`, the inner loop, repeated `J` times, reads the `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 after reading the current values in the `j`-th step before writing the new values, and again after writing on Line 10 before reading the values in the `(j+1)`-th step.

The second stage is implemented using a *join barrier*. The parent task executes the `finish` statement starting on Line 2 by executing the statement body and waiting for all transitively spawned tasks (*i.e.,* the `I` sub-tasks) to finish, before proceeding to its continuation (Line 14).

The ability to dynamically register (and drop) tasks at any point during the lifetime of the X10 clock, as in the above code, is an expressive feature, but can be a source of subtle deadlocks. The above code deadlocks because all of the `I` tasks are stuck on the first `advance`, since the (implicitly registered) parent thread never calls `advance`. The most straightforward solution is to have the parent task to call `c.drop()` between Lines 13 and 14 to deregister itself from the clock.

As the example shows, `finish` join barriers and `Clock` cyclic barrier are distinct abstractions in X10. With `finish` barriers, the parent task observes when a group of (other) tasks reaches the barrier, while the tasks that can observe a `Clock` barrier are its members. The `finish` barrier is observed by the parent task once, whereas a `Clock` can be reused (as above) to allow the member tasks to wait for each other periodically. Join barriers are commonly used in fork/join patterns and with futures [12] (where tasks can observe the asynchronous evaluation of functions), both of which feature in Java and X10. Cyclic barriers are also found in the SPMD programming model, where "collective" operations are repeatedly invoked by all members of a task group to communicate data with implicit barrier synchronisations.

2.2 Generalised barrier synchronisation using phasers

Phasers [33] were proposed in HJ as an extension of X10 clocks, and a limited form of phasers were later included in Java 7. A phaser is used to count and observe events generated by a group of tasks, akin to event counters [29], a classical synchronisation mechanism.

Figure 2 lists a Java version of the previous example, using the `java.util.concurrent.Phaser` API to encode 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 is ini-

```
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();
5    b.register();
6    new Thread() { // Spawn task i
7      public void run() {
8        for (int j = 1; j <= J; j++) {
9          l = a[i-1];
10         r = a[i+1];
11         c.arriveAndAwaitAdvance();
12         a[i] = (l + r) / 2;
13         c.arriveAndAwaitAdvance();
14       }
15       c.arriveAndDeregister();
16       b.arriveAndDeregister();
17     }
18   }.start();
19 }
20 b.arriveAndAwaitAdvance();
21 handle(a);
```

Figure 2: The example implemented using Java phasers.

tialised to `1` to signify the registration of the parent task to this phaser. On Line 4, each of the `I` tasks (threads) is registered with the `c`-phaser. Intuitively, each task is assigned a non-negative monotonic integer, called the *local phase*, that is incremented when the task *arrives* at an event on the phaser; a phaser event is then *observed* by a task if the local phase of every member is at least at the value of the event. Analogously to the X10 code, the cyclic barrier synchronisations are thus performed by each task invoking `arriveAndAwaitAdvance` on `c` on Lines 11 and 13 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 16 the non-blocking `arriveAndDeregister` method on `b` when finished, which the parent task observes by `arriveAndAwaitAdvance` on Line 20. 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 19 and 20. 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.

Although Java phasers can express X10 cyclic and join barriers, they are limited compared to the "full" phasers supported in HJ (and formalised in Armus). Similarly to X10 clocks, correct usage of Java phasers requires all registered tasks to eventually observe (*i.e.,* await) every synchronisation event. In contrast, HJ phasers (1) decouple a non-blocking operation to arrive at a phase from the blocking operation to await a phase advance (when all registered tasks have arrived at that phase), such that (2) it is not necessary for the task to await the phase after arriving before proceeding to the next. HJ phasers thus permit tasks to await arbitrary phases. Works on phasers include synchronisation algorithms [25, 36], data-flow programming models [34, 35], and OpenMP extensions [37]. By designing Armus to support HJ phasers, we subsume deadlock detection for X10 and Java barrier programs under one central abstraction, which can also be readily applied to the barrier concurrency mechanisms in the other languages summarised in the next subsection.

3. Armus: a framework for dynamic deadlock verification

Armus is a *language-agnostic* framework to perform deadlock verification, optimised for barrier synchronisation. We present two verification tools that use Armus: JArmus to check Java programs, and Armus-X10 to check X10 programs.

The verification tools work by "weaving" the verification in a given program. The input of a verification tool is the compiled program we want to verify (Java bytecode). The output is a verified program (Java bytecode) that includes dynamic checks for deadlock detection/avoidance. The Armus framework manages the SG and checks it for cycles. The JArmus and Armus-X10 layers instrument the program to supply JArmus with edges before any task blocks.

3.1 Overview of Armus

The Armus framework maintains a graph-representation of dependencies between resources, and performs cycle detection. The graph is incrementally built using three operations: (1) before synchronisation tasks put edges in the shared graph; (2) the tool checks for cycles; and (3) after synchronisation, tasks take the added edges out of the graph.

Resources r correspond to synchronisation events. To uniquely represent a synchronisation event we pair the barrier with an event counter, *e.g.*, resource $b\colon 1$ represents the first time barrier b is used for synchronisation. For a given task t that synchronises, the runtime environment must provide the set of resources the task synchronises with, R_1, and the set of blocked resources, R_2. In X10, R_1 is the set of clocks and finish barriers the task is registered with; R_2 is the clock or finish barrier in which the task is blocked. The runtime environment generates an edge (r_1, r_2) for each $r_1 \in R_1$ and $r_2 \in R_2$. Armus-X10 considers finishes to always be at phase 1.

Figure 3 lists a deadlock between two tasks executing the running example, in Figure 1, instrumented with Armus-X10. In Line 13, task 1 is registered with clock `c`, thus $R_1 = \{c\colon 1\}$, and it is blocked at the end at the end of the finish, so we have $R_2 = \{f\colon 1\}$. The generated edge for task 1

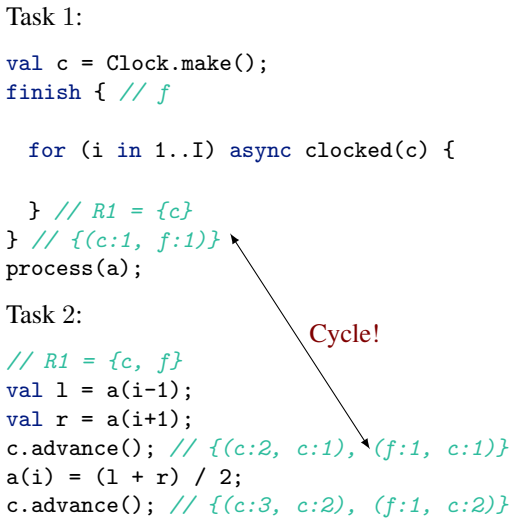


Figure 3: Edge generation of the running example.

is therefore $(c\colon 1, f\colon 1)$. In Line 8, task 2 is registered with clock `c` at phase 2 (since the task advanced the clock) and with finish `f`, so $R_1 = \{c : 2, f : 1\}$; the task is awaiting other tasks to reach phase 1, so $R_2 = \{c : 1\}$. Task 2 generates edges $(c\colon 1, f\colon 1)$ and $(c\colon 2, c\colon 1)$. The cycle happens from edge $(c\colon 1, f\colon 1)$ to edge $(f\colon 1, c\colon 1)$.

SG vs WFG There is a property we note to favour the SG over the WFG for barrier deadlock detection, besides the difference in the complexity of cycle detection. Constructing a SG generates less contention than for the WFG, since building the SG relies entirely on the barriers the task is registered with (task-local information). Instead, constructing a WFG requires the participants registered with the barrier (information spread across tasks).

3.2 Internals of the Armus framework

The Armus framework is divided into (i) the state graph layer and (ii) the verification strategy layer. Layer (i) stores edges and performs cycle detection with an off-the-shelf library called JGraphT [16]. Layer (ii) mediates the use of the graph: it defines when to invoke cycle detection, and provides a façade for tasks to add and to remove edges from the graph. Armus provides two strategies: deadlock avoidance and deadlock detection. The framework is available as a remote service, a necessary feature to support the verification of distributed applications.

Deadlock avoidance strategy only permits tasks to synchronise if it is safe to continue, *i.e.,* the task is not going to deadlock because of the synchronisation it is about to perform. Before blocking on a barrier, the strategy layer places the edges in the graph, checks for cycles, and then throws an exception if a cycle is found. The programmer can catch the exception to recover from the avoided deadlock. The buffer of edges in the graph layer must be able to cope with concurrent requests of adding (removing) edges, and of cycle detection.

Deadlock detection strategy allows tasks to synchronise at will, possibly reaching a deadlocked state. This strategy tackles two factors with a major impact on the performance of the deadlock avoidance strategy, namely (i) cycle detection occurs whenever a task synchronises, and (ii) the state graph is a source of contention. To address (i) we centralise cycle detection in a single task, called the monitor, that periodically checks for cycles. This way, at synchronisation points, tasks only need to add and remove edges from the state graph. To address (ii) we use a double buffer technique. Before synchronising, tasks place their edges in the first buffer. Periodically, the monitor task moves all edges from the first buffer to the graph layer (that acts as a second buffer). Nevertheless, the use of two buffers complicates the removal of edges. When the runtime environment requests for an edge removal, the platform marks the edge as being expired. We include an additional task that periodically garbage collects expired edges from the buffers. Removals from the second buffer only occurs after checking for cycles. The tools is parametric on both the monitoring and the garbage collection time intervals.

3.3 The design of the verification tools

Both JArmus and Armus-X10 share the same usage and design. The implementation each of these verification tools is divided into two components: the resource mapper and the task observer. The resource mapper converts blocking calls and impeding synchronisation as resources. The task observer intercepts blocking calls to inform Armus that the current task is blocked with a set of resource edges. The set of edges is constructed from the blocking call to each impeding resource. The task observer is programmed with Aspect-Oriented programming, through AspectJ [19].

3.4 JArmus

JArmus verifies class operations of `Thread`, `CountDownLatch`, `CyclicBarrier`, `Phaser`, and `ReentrantLock`. In Java, the relationship between the participants of barrier synchronisation and tasks is implicit. For example, when using a `CyclicBarrier` the programmer declares the number of participants and then shares the object with those many tasks. It is not specified which tasks participate in the synchronisation. JArmus has no way of reconstructing this information for the `CountDownLatch`, `CyclicBarrier`, and `Phaser` classes, so the programmer must annotate its code to supply the barriers a task is registered with, invoking `JArmus.register(b)` for barrier `b`. The verification of `Thread` and `ReentrantLock` do not require manual annotations. Support for verifying mutual exclusion deadlocks is still experimental, as it may suffers from potential false deadlocks, a usual limitation of current deadlock verification techniques [5].

3.5 Armus-X10

Armus-X10 can verify any program that uses clocks and finishes. There is support for distributed X10 applications, as long as all parts are instrumented and connect to the same remote Armus service. X10 can be compiled to Java bytecode, called Managed X10, and to machine code, called Native X10. Our tool only supports Managed X10.

The instrumentation in Armus-X10 is completely automatic. This is made possible by the information provided by X10 runtime. To generate the edges, Armus-X10 emits the set of impeding resources, that comprises the clock phases and finishes a in which a given task is registered. We consider that a task is registered with a finish if the task executes within the scope of that finish.

4. Evaluation

The aim of the evaluation process is to ascertain whether the performance impact of Armus scales well with the increase in the number of tasks. For that we benchmark Armus' execution overhead against two suites of parallel applications, as we increment the number of tasks.

4.1 Benchmark suites description

We use the NAS Parallel Benchmark (NPB) [9] and the Java Grande Forum [38] (JGF) benchmark suites to evaluate Armus. All benchmarks check the validity of the produced output. Next, we describe each benchmark suite.

NPB The NPB suite of benchmarks ranges from kernels to pseudo-applications, taken primarily from Computational Fluid Dynamics (CFD) applications. The benchmarks include computational kernels and pseudo-applications that exhibit the communication and computational patterns commonly found in CFD applications. We used the following benchmarks:

`BT` is a pseudo-application that solves 3-dimensional (3-D) compressible Navier-Stokes equations;

`CG` is a computational kernel that uses a Conjugate Gradient method to compute approximations to the smallest eigenvalues of a sparse unstructured matrix;

`FT` is a computational kernel of a 3-D Fast Fourier Transform;

`LU` is a pseudo-application that uses the symmetric successive over-relaxation method to solve the discrete Naive-Stokes equations by splitting it into block lower and upper triangular systems;

`MG` is a computational kernel that uses the V-cycle multi grid method to compute the solution of the 3-D scalar Poisson equation;

`SP` is a pseudo-application that employs the Beam-Warming approximate factorisation in a system of scalar pentadiagonal linear equations.

The synchronisation patterns in the NPB-JAV suite is similar for most cases, except for `LU`. The synchronisation is implemented with condition variables. Their algorithms are iterative, and tasks use a cyclic barrier to synchronised stepwise.

In the `LU` benchmark there is a pipeline dependence between workers, enforced with a relay-race task synchronisation, that uses one barrier per a task.

We converted the `FT` benchmark to X10, and call it NPB-X10 suite. The source code is available through our project's page [3]. The translation is semi-automatic; the biggest differences between Java and X10 amounts to type declarations. For cyclic barrier synchronisation we opt for clocks instead of condition variables.

JGF The Java Grande Forum benchmark suite is divided into three groups of applications: micro-benchmarks, computational kernels, and pseudo-applications. We selected one pseudo-application, `RT`, that emulates a 3-D ray tracer. The algorithm is iterative and tasks use a cyclic barrier to synchronised stepwise.

4.2 Benchmark results

Setup 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 is Ubuntu 13.10 and the OpenJDK Runtime Environment is IcedTea 2.4.4.

We follow the *start-up performance* methodology detailed in [10]. 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. The benchmarks are verified in deadlock detection mode, as it is the most optimised of the two modes.

All benchmarks accept the size of the input as a parameter. We choose the smallest input size such that the application scales (runs faster) as we add more tasks. For the sake of reproducibility we list the code of the input as specified in each benchmark: `BT` uses size A, `CG` uses size C, the Java version of `FT` uses size B, `LU` uses size W, `MG` uses size C, `SP` uses size W, `RT` uses B, the X10 version of `FT` uses size A.

The input set chosen for benchmarks `LU` and `SP` only lets them scale up to 34 and 31 tasks, respectively. For simplicity, in the evaluation we consider that these benchmarks scale up to 32 tasks. Larger input sets take at least one month per benchmark to complete with our hardware configuration, so we opt for using a smaller input set.

Results Figure 4 summarises the comparative study of the execution time for each benchmark. The results show that, for up to 64 tasks, scaling the application has no impact in the verification.

The results for the NPB and JGF benchmark suites are depicted in Figures 4a to 4g. This metric indicates that verification is unaffected when scaling the benchmark. Adding

tasks does *not* suggest an increase of the overhead. For example, the execution overhead between 32 and 64 tasks only increases on 2 out of 5 benchmarks (`BT` and `LU`).

Figure 4c and Figure 4h represent a similar overhead for the same benchmark implemented in Java and X10, respectively. The verification algorithm is the same, so the biggest change is the language implementation. The metrics indicate that there is no impact in X10.

5. Related work

This section lists related work forcussing on deadlock in HPC program languages. For more extensive comparisons, including an historical summary of barrier synchronisations, see [7, § 2].

Deadlock prevention. The literature around source code analysis to prevent global barrier deadlocks is vast, see [7, § 2] for related work on deadlock preventions of MPI, OpenMP and OpenSHMEM.

The fork/join programming model is easily restricted syntactically to prevent deadlocks from happening. Lee and Palsberg presented a calculus for a fork/join programming model [23], suited for inter-procedural analysis through type inference, and establishes the deadlock freedom property. This work also includes a type system that is used to identify may-happen-parallelism, further explored in [2].

Our work in [8] also proposes a static typing system to essure correctness of phased activities. In [8], there is no single await primitive so that any example that has more than two phasers and performs `await(c)` cannot be expressed. Thus none of X10 and Java programs in § 2 can be verified by [8].

There is some work surrounding the formalisation of barrier semantics that considers more complex idioms of barrier synchronisation, but do not establish deadlock-freedom. For example, Saraswat and Jagadeesan formalise a subset of X10 that prevents deadlocks [32], comprising join barriers and clocks. Le *et al.* devise a verification for the correct use of a cyclic barrier in a fork/join programming language [22]. In [39], Vasudevan *et al.* perform static analysis to specialise the implementation and thus improve performance of the synchronisation algorithm.

The tool X10X [11] is a *model checker* for X10. Model checkers perform source code analysis and can be used to discover potential deadlocks. This class of tools suffers from 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 knowledge, techniques that avoid deadlocks, in the context of barrier synchronisation, only handle certain classes of deadlocks. Unlike our proposal, these techniques are *not* complete with regards to our definition. In X10 and HJ, tasks deregister from all barriers upon terminating, this mitigates deadlocks that arise from missing participants. HJ includes two dynamic mechanisms to avoid deadlocks that originate from the interaction between phasers and finish blocks. The runtime disallows tasks spawned within a finish to be registered with phasers defined outside the scope of that finish. Plus, at the end of a finish block (before blocking) tasks automatically deregister from phasers created within a finish block.

Deadlock detection. The deadlock detection techniques that handle most barrier idioms are the ones targeting MPI, as it is only incapable of describing producer-consumer barrier synchronisation. Literature concerning MPI deadlock detection takes a top-bottom approach, the general idea is given but mapping it to the actual semantics of the language is left as an exercise to the reader. Umpire [13] and MUST [14] (a successor of Umpire) use a graph-based deadlock detection algorithm that subsumes deadlock detection to cycle detection in a graph, but omit a formal description on how the graph is actually generated from the language. There are some works on SPMD languages with global collective operations: Titanium [18] (an extension of Java with SPMD support) and UPC [30]. Finally, there are some tools that consider the stated deadlock after a period of inactivity that cannot guarantee correctness: DAMPI [40], Marmot [21], and MPI-CHECK [24].

As far as we are aware, our work is the first deadlock detection/avoidance mechanism which is applicable to representable barriers in Java and X10. The correctness of our implementation is backed up by the soundness and completeness results between graph algorithms and langauge semantics.

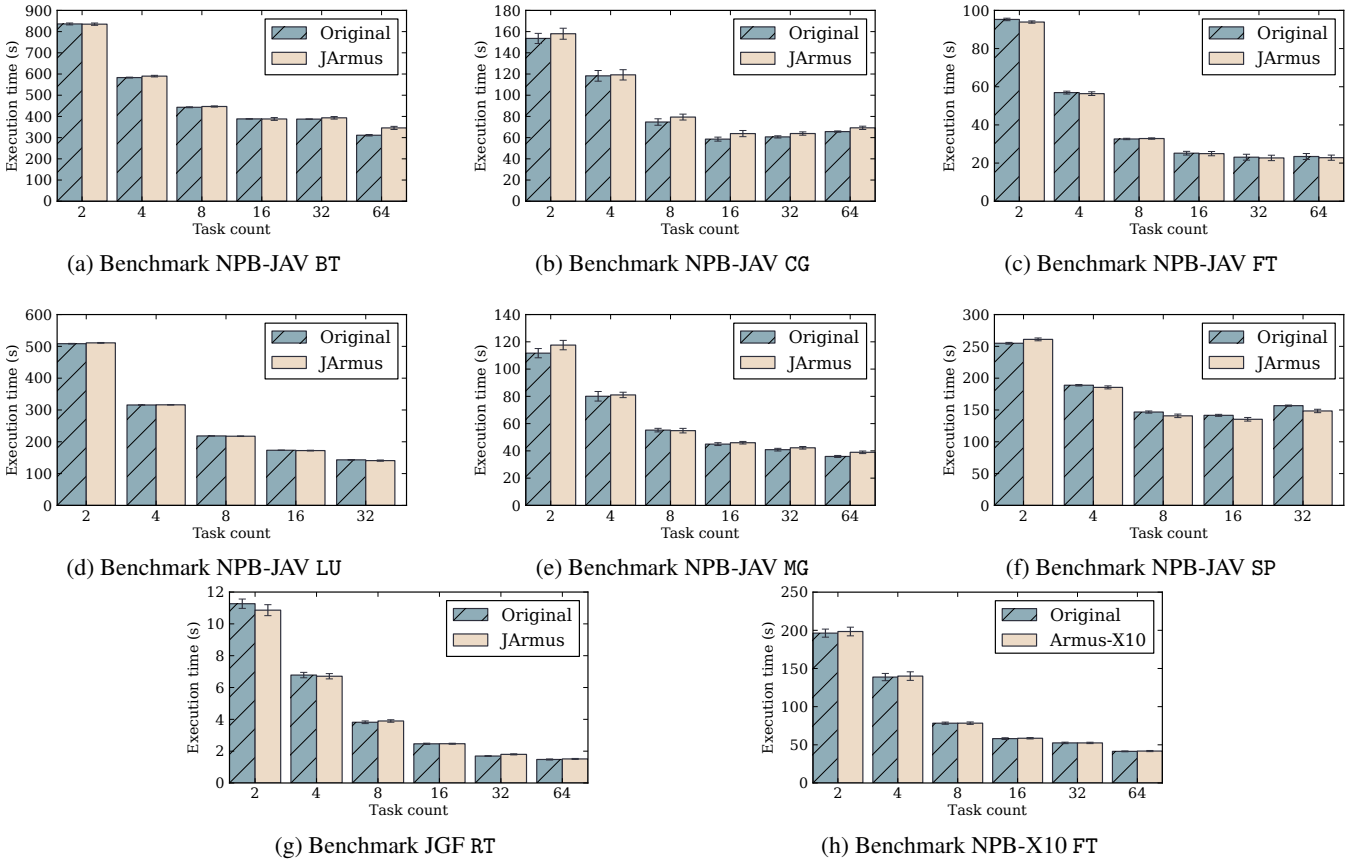
6. Conclusion

We put forward a dynamic verification technique for barrier deadlocks. The verification technique is graph based, and subsumes deadlock identification to cycle detection. The runtime overhead of our deadlock detection algorithm is shown to be negligible and independent of the number of tasks, for up to 64 tasks. We introduce two applications of this framework: Armus-X10 monitors any unchanged X10 program for deadlocks; JArmus is a library to verify Java programs.

Our next step is to add support for the verification of mutual exclusion without suffering from false deadlocks. We intend to build a tool for HJ to exercise the expressiveness of Armus. This language features abstractions with complex synchronisation patterns, such as multiple bounded producer-consumer.

References

[1] J. L. Abellán, J. Fernández, and M. E. Acacio. Efficient hardware barrier synchronization in many-core CMPs. *Transactions on Parallel and Distributed Systems*, 23(8):1453–1466, 2012.



(a) Benchmark NPB-JAV `BT`

(b) Benchmark NPB-JAV `CG`

(c) Benchmark NPB-JAV `FT`

(d) Benchmark NPB-JAV `LU`

(e) Benchmark NPB-JAV `MG`

(f) Benchmark NPB-JAV `SP`

(g) Benchmark JGF `RT`

(h) Benchmark NPB-X10 `FT`

Figure 4: Comparative execution time for each benchmark (lower means faster).

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

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

[4] M. Bataille. Something old: the Gamma 60 the computer that was ahead of its time. *SIGARCH Computer Architecture News*, 1:10–15, 1972.

[5] Y. Cai and W.-K. Chan. Magicfuzzer: Scalable deadlock detection for large-scale applications. In *Proceedings of ICSE'12*, pages 606–616. IEEE, 2012.

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

[7] T. Cogumbreiro. *Programming multicores safely: handling barrier deadlocks*. PhD thesis, University of Lisbon, 2014. To appear. Available at [3].

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

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

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

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

[12] R. H. Halstead, Jr. Implementation of Multilisp: Lisp on a multiprocessor. In *Proceedings of LFP'84*, pages 9–17. ACM, 1984.

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

[14] 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 *Proceedings of SC'12*, pages 1–11. IEEE, 2012.

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

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

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

[18] A. Kamil and K. Yelick. Enforcing textual alignment of collectives using dynamic checks. In *Proceedings of LCPC'09*, volume 5898 of *Lecture Notes in Computer Science*, pages 368–382. Springer, 2009.

[19] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP'01*, volume 2072, pages 327–353. Springer, 2001.

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

[21] B. Krammer, T. Hilbrich, V. Himmler, B. Czink, K. Dichev, and M. S. Müller. MPI correctness checking with Marmot. In *Proceedings of PTW'08*, pages 61–78. Springer, 2008.

[22] 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, pages 231–248, 2013.

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

[24] G. R. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, and Y. Zou. MPI-CHECK: a tool for checking Fortran 90 MPI programs. *Concurrency and Computation: Practice and Experience*, 15(2):93–100, 2003.

[25] 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 *Proceedings of HPCC'10*, pages 130–137. IEEE, 2010.

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

[27] OpenMP homepage. `openmp.org/wp/`.

[28] J. L. Peterson and A. Silberschatz. *Operating System Concepts*. Addison-Wesley, 1985.

[29] D. P. Reed and R. K. Kanodia. Synchronization with eventcounts and sequencers. *Communications of the ACM*, 22(2):115–123, 1979.

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

[31] M. Samak and M. K. Ramanathan. Trace driven dynamic deadlock detection and reproduction. In *Proceedings of PPoPP'14*, pages 29–42. ACM, 2014.

[32] V. Saraswat and R. Jagadeesan. Concurrent clustered programming. In *Proceedings of CONCUR'05*, volume 3653 of *Lecture Notes in Computer Science*, pages 353–367. Springer, 2005.

[33] 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 *Proceedings of ICS'08*, pages 277–288. ACM, 2008.

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

[35] 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.

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

[37] J. Shirako, K. Sharma, and V. Sarkar. Unifying barrier and point-to-point synchronization in openmp with phasers. In *Proceedings of IWOMP'11*, pages 122–137, 2011.

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

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

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

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