Deadlock Avoidance in Parallel Programs with Futures

Why parallel tasks should not wait for strangers

TIAGO COGUMBREIRO, Rice University, USA
RISHI SURENDRAN, Rice University, USA
FRANCISCO MARTINS, LaSIGE, Faculty of Sciences, University of Lisbon, Portugal
VIVEK SARKAR, Rice University, USA
VASCO T. VASCONCELOS, LaSIGE, Faculty of Sciences, University of Lisbon, Portugal
MAX GROSSMAN, Rice University, USA

Futures are an elegant approach to expressing parallelism in functional programs. However, combining futures with imperative programming (as in C++ or in Java) can lead to pernicious bugs in the form of data races and deadlocks, as a consequence of uncontrolled data flow through mutable shared memory.

In this paper we introduce the *Known Joins* (KJ) property for parallel programs with futures, and relate it to the *Deadlock Freedom* (DF) and the *Data-Race Freedom* (DRF) properties. Our paper offers two key theoretical results: 1) DRF implies KJ, and 2) KJ implies DF. These results show that data-race freedom is sufficient to guarantee deadlock freedom in programs with futures that only manipulate unsynchronized shared variables. To the best of our knowledge, these are the first theoretical results to establish sufficient conditions for deadlock freedom in imperative parallel programs with futures, and to characterize the subset of data races that can trigger deadlocks (those that violate the KJ property).

From result 2), we developed a tool that avoids deadlocks in linear time and space when KJ holds, i.e., when there are no data races among references to futures. When KJ fails, the tool reports the data race and optionally falls back to a standard deadlock avoidance algorithm by cycle detection. Our tool verified a dataset of ~2,300 student's homework solutions and found one deadlocked program. The performance results obtained from our tool are very encouraging: a maximum slowdown of 1.06× on a 16-core machine, always outperforming deadlock avoidance via cycle-detection. Proofs of the two main results were formalized using the Coq proof assistant.

CCS Concepts: • **Software and its engineering → Deadlocks**; **Software verification**; **Dynamic analysis**; *Concurrent programming structures*; *Semantics*;

Additional Key Words and Phrases: futures, deadlock avoidance, Habanero-Java, Java

ACM Reference Format:
Tiago Cogumbreiro, Rishi Surendran, Francisco Martins, Vivek Sarkar, Vasco T. Vasconcelos, and Max Grossman. 2017. Deadlock Avoidance in Parallel Programs with Futures: Why parallel tasks should not wait for strangers. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 103 (October 2017), 26 pages.
https://doi.org/10.1145/3133927

Authors' addresses: Tiago Cogumbreiro, Rice University, USA, cogumbreiro@rice.edu; Rishi Surendran, Rice University, USA, rishi@rice.edu; Francisco Martins, LaSIGE, Faculty of Sciences, University of Lisbon, Portugal, fcmartins@fc.ul.pt; Vivek Sarkar, Rice University, USA, vsarkar@rice.edu; Vasco T. Vasconcelos, LaSIGE, Faculty of Sciences, University of Lisbon, Portugal, vmvasconcelos@fc.ul.pt; Max Grossman, Rice University, USA, max.grossman@rice.edu.

2475-1421/2017/10-ART103
https://doi.org/10.1145/3133927

1 INTRODUCTION

Futures [Halstead 1985] offer a simple model of parallelism. Programmers annotate certain function calls as being asynchronous. Calling one such function immediately returns a future object that acts as a placeholder for the value computed by that call. The *get* primitive on a future object retrieves the value computed by the asynchronous function call. An invocation of the primitive blocks its caller's execution until the asynchronous call terminates. We focus on a parallel programming model where asynchronous calls fork new tasks, and obtaining function values via futures awaits task termination.

As with any source of concurrency, the interaction with shared memory must be carefully regarded. Since future objects can be freely communicated via shared memory, program execution may introduce circular dependencies among tasks waiting on future values, which constitutes a form of deadlock. The focus of this paper is twofold: to formally study the relationship between deadlocks on futures and accesses to shared memory, and to propose an efficient runtime technique to ensure that tasks manipulating futures never reach deadlocked situations.

The deadlock problem [Isloor and Marsland 1980] can be handled in one of three ways: by statically *preventing* the deadlock from happening at all [Boyapati et al. 2002a; Williams et al. 2005], by monitoring the runtime execution while *detecting* already deadlocked situations [Hilbrich et al. 2009, 2012; Krammer et al. 2008; Luecke et al. 2003; Vo 2011], and by dynamically *avoiding* deadlocks from happening [Boudol 2009; Cogumbreiro et al. 2015; Dijkstra 1965; Gerakios et al. 2011; Li et al. 2010]. Some works delay, or replay, schedules to avoid deadlocks [Boudol 2009; Gerakios et al. 2011; Navabi et al. 2008; Welc et al. 2005]; in our model as in Cogumbreiro et al. [2015], waiting on a deadlocked future raises an exception.

The problem of deadlock avoidance for task termination (joining) is known to have quadratic time and space complexity on the number of running tasks [Reveliotis et al. 1997a]. This fact alone makes deadlock avoidance on parallel runtimes prohibitive to use in practice. To circumvent the problem, programming language researchers proposed different syntactic restrictions on programs that prevent deadlocks from ever happening. For example, the X10 language [Charles et al. 2005] includes an async/finish construct for task forking and joining: at the end of a finish scope the task running the finish operation joins with all tasks forked in that scope, directly or indirectly. X10 programs are deadlock free by construction for tasks joining with a finish construct. Unfortunately, syntactic restrictions are inapplicable to programming models with futures.

We propose a theory of futures and shared memory programming to reason about deadlocks caused by synchronization on task termination (henceforth called joining). We introduce a runtime policy on joining, called *Known Joins* (KJ), that guarantees absence of deadlocks. The two ways for a task to augment its knowledge is by forking and by joining. When forking a new task, the forking task (the parent) becomes acquainted with forked task (the child). The child inherits the knowledge of the parent but does not know its own identity (thereby ensuring tasks cannot trivially deadlock on themselves). After joining, the waiting task augments its knowledge with that of the terminated task. The Known Joins policy says that tasks should only join with known tasks, and never with tasks obtained by other means. KJ is applicable to any programming model where tasks can await the termination of other tasks, including models with shared memory multithreading, fork-join, and futures. This is the essence of the first part of the formal development, section 3, which proves that the root cause of deadlocks is the voiding of the Known Joins policy.

The second part of the formal development, section 4, concentrates on programs that manipulate futures as the sole synchronization mechanism. We show that, for this subset, only racy programs can violate the Known Joins policy, or, conversely, that all data-race-free programs follow the Known Joins policy. In this setting, the chain of events for a deadlock is as follows: a racy read

causes a task to introduce an unknown task name, which causes a join with an unknown task, which leads to a deadlock. However, for programs that include other synchronization mechanisms beyond futures, such as mutexes and critical sections, data races are not the core issue (since deadlock is possible in data-race-free programs with these mechanisms). Instead, as summarized in section 3, the root cause of deadlocks with futures is the voiding of the Known Joins policy.

We show that the Known Joins property avoids task-termination deadlocks in linear time and space in the presence of data-race free computations, surpassing state-of-the art deadlock avoidance tools that run in quadratic time and space. By showing that the KJ property can be formulated as a causality test, we set an upper bound on the time and space complexities of KJ testing—a state-of-the-art causality test uses vector clocks and is linear in time and quadratic in space [Fidge 1988; Mattern 1989]. As an alternative to vector clocks, we propose an algorithm based on *snapshot-sets* that achieves both linear space and time complexities. In our solution, each task maintains its set of known tasks, extending this knowledge at forking and joining points. With vector clocks, extending knowledge means duplicating the data from one task to another. A snapshot-set is a concurrent graph-like data structure holding the known tasks. Extending knowledge means duplicating references, thereby avoiding copying the snapshots, thus saving memory.

Our technique can be effectively used in programming models with different parallel constructs. The KJ property is oblivious to data movements (e.g., via volatiles, atomics, and mutual exclusion) and can be integrated with most synchronization mechanisms we are aware of. To this end, we add KJ checks to the Habanero runtime [Cavé et al. 2011]. The deadlock-free Habanero programming model consists of three constructs: *isolated* [Imam et al. 2015] for mutual exclusion; *async-finish* [Charles et al. 2005], a generalization of the fork-join model where a task can await the termination of its descendants; and, *phasers* [Shirako et al. 2008], a generalization of barriers with producer-consumer synchronization. By adding futures to this mix, deadlocks can only be caused by futures. Our extended runtime therefore supports deadlock-freedom with futures, mutual exclusion, fork-join, and barriers.

The contributions of this paper are:

Theoretical. A formalization of the KJ property for parallel programs with future tasks as the sole form of synchronization that enjoys *Deadlock Freedom* (DF) and *Data Race Freedom* (DRF). Our paper offers two key theoretical results: 1) DRF $\implies$ KJ, and 2) KJ $\implies$ DF; hence, DRF $\implies$ DF. To the best of our knowledge we are the first to build a mathematical argument on the root cause of deadlocks in this context. A novelty of our approach is to use causality to reason about deadlock avoidance. Our theory and results are fully mechanized using the Coq proof assistant.[1]

Algorithmic. A solution to the deadlock avoidance problem in $O(n)$ time and space, where n is the number of tasks forked. Known solutions take $O(m^2)$ time and space where m is the number of running tasks. By using the finish construct we have that $n \approx m$. When compared to other causality analysis algorithms, ours solution uses $O(n)$ space, whereas traditional solutions (e.g., vector clocks) require $O(n^2)$.

Implementation. An extension of the Habanero for Java runtime that grants deadlock avoidance for futures, mutual exclusion, finish blocks, and phasers. Gorn—the implementation of KJ—outperforms standard deadlock avoidance based on cycle detection.[2] Gorn verified a dataset of ~2,300 student's homework solutions and found one deadlocked program. We also evaluated the overheads of our tool against a series of standard parallel programs and witnessed a maximum slowdown of 1.08×, and a maximum memory overhead of 2.34×. The

[1] Our Coq mechanization can be downloaded at: https://gitlab.com/cogumbreiro/gorn-coq/tree/oopsla17

[2] Gorn can be downloaded at: https://gitlab.com/cogumbreiro/gorn/tree/oopsla17

Listing 2.1 The program invokes `A()` and `B()` concurrently. This program exhibits neither deadlocks nor races.

```
1 // Task f
2 future<future<int>> a = async {          // Task g
3   future<int> b = async { return B();}; // Task h
4   A(); return b; };
5 future<int> c = a.get();
6 c.get();
```

benchmarks scale up to 1 million tasks and exercise distinct synchronization patterns using futures, such as a parent waiting for many children, recursive parent-children wait, sibling waits, and consecutive fork-joins.

The rest of the paper is organized as follows. Section 2 introduces futures and its programming model. Section 3 abstracts program execution by means of traces and presents the KJ property. Next, section 4 establishes the root cause of future-based deadlocks. Section 5 discusses Gorn along with its implementation details, and is followed by section 6 that evaluates the tool. Finally, section 7 covers related work and section 8 concludes the paper.

2 PROGRAMMING WITH FUTURES

This section introduces programming with futures in the Habanero-Java (HJ) language [Cavé et al. 2011], an extension of Java with constructs for task parallelism. In the notation used in this paper, expression **async** `S` forks a task that evaluates statement `S` in parallel with the forking task and returns a future of type **future**<`T`>, where `T` is the type of the value returned by statement `S`. The future includes a **get**() method that can be invoked to await the termination of the task forked by the **async**, thus retrieving the value returned by `S`. Subsequent invocations of **get**() are nonblocking and always return the same value. For the sake of presentation, variables declared with a **shared** modifier can be shared among tasks, otherwise they are local to a single task.

The program in listing 2.1 calls functions `A()` and `B()` in parallel. This program illustrates three tasks communicating task names via futures values (obtained via **get**), as depicted in figure 1. Task f forks task g and stores task name g in variable `a`, in line 2. Task g forks h, in line 3, and then calls `A()`. Concurrently, task h invokes `B()`. After forking g, task f awaits the termination of task g to retrieve h, in line 5. Finally, task f awaits the termination of task h, in line 6. The type of an **async** expression is a **future** of the type of the expression returned by that forked task. For instance, in line 3, task h returns `B()` (an **int**), so variable `b` is of type **future**<**int**>.

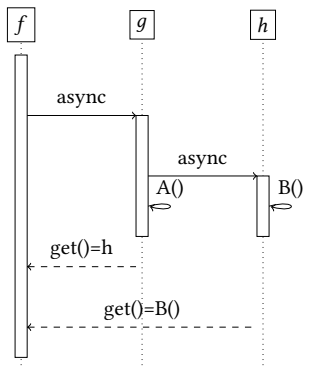

Fig. 1. Sequence diagram of listing 2.1.

Futures can be communicated as values or stored in variables, allowing for complex synchronization patterns. Listing 2.2 illustrates the nefarious concurrency errors that can be triggered by the use of shared memory. Task f creates two shared variables, `x` and `y`. Next, f forks two tasks g and h, and stores the futures in shared variables `x` and `y`, respectively. Concurrently, task g awaits the value of task h in future variable `y`, while task h awaits the value of g in future variable `x`. There is a race condition between reading from `y` in line 3, and writing to `y` in line 4, which leads to two possible

Listing 2.2 Example program with a data race and a deadlock.

```
1 // Task f
2 shared future<int> x, y;
3 x = async {return y.get();};       //Task g
4 y = async {return x.get();};       //Task h
```

Listing 2.3 Program in listing 2.2 rewritten in C++.

```
1 std::future<int> x; std::future<int> y;
2 x = std::async(std::launch::async, [&](){return y.get();});
3 y = std::async(std::launch::async, [&](){return x.get();});
```

interleavings: either the task g observes (reads) the initial value of y, which in Habanero-Java defaults to **null**, or task g observes the write of h in line 4. In the former interleaving, task g invokes **get** on a **null** reference which triggers a NullPointerException. In the latter interleaving, the writes to x and to y complete before the reads from x and from y, thus futures g and h left in a circular dependency, deadlocked, each waiting for the other to terminate.

While this paper motivates its discussion with Habanero-Java examples, the results and techniques translate to most languages that use futures. For instance, we can easily rewrite listing 2.2 into C++ in listing 2.3. The deadlock and the data race also exist in the C++ version.

3 THE KNOWN JOINS PROPERTY IMPLIES DEADLOCK FREDOM

Our approach to avoid deadlocks is to analyze the history of asyncs and gets to derive a partial order on task names. We introduce a trace language to model the history of computations in section 3.1, and our deadlock avoidance technique in section 3.2.

3.1 Trace Language

We define the language of *instruction traces* via the grammar below. An instruction trace $t \in \mathcal{T}$, henceforth just trace, is a sequence of actions. An *action* $a \in A$ pairs a task name $f \in \mathcal{F}$ with an operation $o \in \mathcal{O}$. Operations include the typical primitives in a fork-join model of computation. The values $d \in \mathcal{D}$ of interest for our trace language are task names f and memory locations $r \in \mathcal{R}$. We note the absence of value **null** in the trace language: we are only interested in operations that execute error free.

$$
\begin{array}{rcll}
t & ::= & \epsilon \mid t; a & \textit{Traces} \\
a & ::= & (f, o) & \textit{Actions} \\
o & ::= & \texttt{init} \mid m \mid \texttt{async}\ f \mid \texttt{get}\ f & \textit{Operations} \\
m & ::= & \texttt{new}\ r \mid \texttt{wr}\ r\ d \mid \texttt{rd}\ r\ d & \textit{Memory Operations} \\
d & ::= & f \mid r & \textit{Values}
\end{array}
$$

The trace language includes six operations: init is the first operation any program executes (invoked by the root task); async f forks a task named f; get f joins with task name f; new r creates memory location r (available to all tasks); wr r d writes value d to location r; and rd r d reads value d from location r.

Example 1. We revisit the example of listing 2.1 by presenting its trace (on the right-hand side).

```
1 // Task f                                                    (f, init)
2 future<future<int>> a = async {           // Task g          (f, async g)
3   future<int> b = async { return B(); }; // Task h          (g, async h)
4   A(); return b;};
5 future<future<int>> c = a.get();                            (f, get g)
6 c.get();                                                    (f, get h)
```

Traces are listed left-to-right from the least recent to the most recent action, and include only the relevant primitives in the language. For instance, function calls `A()` or `B()`, are not included in traces. The only possible trace for this example is the following:

$$t_1 = \epsilon; (f, \texttt{init}); (f, \texttt{async}\ g); (g, \texttt{async}\ h); (f, \texttt{get}\ g); (f, \texttt{get}\ h) \tag{1}$$

Next we discuss memory access operations.

Example 2. A trace of the code in listing 2.2 can be written as follows.

$$\begin{aligned} t_2 = \epsilon;&(f, \texttt{init}); (f, \texttt{new}\ r); (f, \texttt{new}\ q); \\ &(f, \texttt{async}\ g); (f, \texttt{wr}\ r\ g); (f, \texttt{async}\ h); (f, \texttt{wr}\ q\ h); (h, \texttt{rd}\ r\ g); (g, \texttt{rd}\ q\ h) \end{aligned} \tag{2}$$

The program starts with task named f running operation `init`. Then it creates memory locations r and q for shared program variables `x` and `y` via two `new` operations. The task continues by initializing memory reference r with the result of forking task named g (the first two actions of the second line). Then it initializes memory location q with the result of forking task named h. The trace ends with task h reading g from r and task g reading h from q. The effect of these reads is that expression `x.get()` evaluates to task h awaiting the termination of task g, and expression `y.get()` evaluates to task g awaiting the termination of task h, thus tasks g and h deadlock. Note that *a circular wait never shows up in a trace*, since traces only record instructions that were actually executed, and not the ones under execution.

3.2 Safety Rules

The theory of Known Joins (KJ) enforces that tasks should only await the termination of *known* tasks. The two ways for a task to augment its knowledge is by forking (**async**) or by joining (**get**).

We introduce some operations on maps, that is, partial functions of finite domain. Functions $\text{dom}(M)$ and $\text{range}(M)$ represent the domain and range of map M, respectively. Function $M[k := v]$ associates entry k to value v in map M, replacing the previous value of k in M if $k \in \text{dom}(M)$.

Knowledge, denoted by K, maps tasks into sets of tasks. We call $K(f)$ the *known tasks* of f. If $g \in K(f)$, then we say that f *knows* g. For example, consider the knowledge associated to the code in listing 2.1:

$$\{f : \{g, h\}, g : \{h\}, h : \emptyset\}$$

We can see that task f knows tasks g and h, that task g knows task h, and that task h does not know any task.

The rules for enforcing the Known Joins policy are defined in figure 2. The judgment $\vdash t : K$ asserts that trace t enjoys the KJ property and produces a knowledge K. The trace is evaluated left-to-right. Rule T-NIL states that the empty trace yields an empty knowledge. Rule T-INIT assigns the empty knowledge set to the initial task. Rule T-MEM states that memory operations m have no effect on the knowledge. Rule T-ASYNC ensures that the forked task name g is fresh; the outcome is that the child g knows what the parent f knew before forking, i.e., $K(f)$, and that the parent f adds the child g to its knowledge. Rule T-GET states that task f can only await the termination of task g if f knows g (from $g \in K(f)$); the outcome is that task f inherits the knowledge of task g.

$$\vdash \epsilon : \emptyset \qquad \frac{\vdash t : K \qquad f \notin \mathrm{dom}(K)}{\vdash t;(f,\texttt{init}) : K[f := \emptyset]} \qquad \frac{\vdash t : K}{\vdash t;(f,m) : K} \qquad \text{(T-NIL,T-INIT,T-MEM)}$$

$$\frac{\vdash t : K \qquad g \notin \mathrm{dom}(K)}{\vdash t;(f,\texttt{async}\ g) : K[g := K(f)][f := K(f) \cup \{g\}]} \qquad \frac{\vdash t : K \qquad g \in K(f)}{\vdash t;(f,\texttt{get}\ g) : K[f := K(f) \cup K(g)]} \qquad \text{(T-ASYNC,T-GET)}$$

Fig. 2. Rules for checking whether a trace complies with Known Joins.

We can easily show that the domain of a knowledge map obtained from the system in figure 2 contains its range.

Lemma 3.1. *If* $\vdash t : K$, *then* $\mathrm{range}(K) \subseteq \mathrm{dom}(K)$.

Example 3. Trace t_1, obtained from listing 2.1, enjoys the *KJ* property.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \epsilon : \emptyset}}{\vdash \epsilon;(f,\texttt{init}) : \{f : \emptyset\}}}{\vdash \epsilon;(f,\texttt{init});(f,\texttt{async}\ g) : \{f : \{g\}, g : \emptyset\}}}{\vdash \epsilon;(f,\texttt{init});(f,\texttt{async}\ g);(g,\texttt{async}\ h) : \{f : \{g\}, g : \{h\}, h : \emptyset\}}}{\vdash \epsilon;(f,\texttt{init});(f,\texttt{async}\ g);(g,\texttt{async}\ h);(f,\texttt{get}\ g) : \{f : \{g,h\}, g : \{h\}, h : \emptyset\}}}{\vdash \epsilon;(f,\texttt{init});(f,\texttt{async}\ g);(g,\texttt{async}\ h);(f,\texttt{get}\ g);(f,\texttt{get}\ h) : \{f : \{g,h\}, g : \{h\}, h : \emptyset\}}$$

Example 4. Similarly, we can easily show that t_2, obtained from listing 2.2, enjoys the *KJ* property.

$$\vdash t_2 : \underbrace{\{f : \{g,h\}, g : \emptyset, h : \{g\}\}}_{K_2}$$

Given trace t_2, task f can await the termination of g and h, it is unsafe for task g await any task, and h can safely await the termination of g. If we inspect trace t_2, we note that task g holds a reference to task h (obtained from location q) and h holds a reference to task g (obtained from location r). Referring back to listing 2.2, at this point of the execution there are two possible actions to be executed: (i) task h invokes a get on g, action $(g, \texttt{get}\ h)$; and (ii) task h invokes a get on g, action $(h, \texttt{get}\ g)$. KJ rejects (i), that is, $\nvdash t_2;(g, \texttt{get}\ h)$, since g does not know h, i.e., $h \notin K_2(g) = \emptyset$. However, (ii) is safe. According to trace t_2, it is safe for task h to wait for task g, since the former knows the latter.

$$\frac{\vdash t_2 : K_2 \qquad g \in K_2(h) = \{g\}}{\vdash t_2;(h,\texttt{get}\ g) : K_2}$$

To reason about deadlocked tasks, we introduce the notion of *knowledge graph*. The nodes of a knowledge graph represent the tasks forked in the trace; the edges represent all possible joins on known tasks.

Definition 3.2 (Knowledge graph). Given a knowledge map K such that $\mathrm{range}(K) \subseteq \mathrm{dom}(K)$, the knowledge graph associated to K, notation $\mathrm{kg}(K)$, is a pair (V, E) where $V = \mathrm{dom}(K)$ and $E = \{(f, g) \mid g \in K(f)\}$.

The condition $\mathrm{range}(K) \subseteq \mathrm{dom}(K)$ ensures that knowledge graphs are well-formed. Given the result of lemma 3.1, we know that we can build a graph from the knowledge map obtained by

the rules in figure 2. The main result of this section states that graphs obtained from traces that conform to the KJ property (figure 2) are acyclic.

Theorem 3.3. *If* $\vdash t : K$, *then* $\mathrm{kg}(K)$ *is acyclic.*

Proof. We proceed by induction in the structure of the trace, and analyze two representative cases. When the trace ends with a `get` operation, task f joins with g, and given an edge, say (g, h), we need to show that adding edge (f, h) preserves the acyclic property. But we know that (f, g) from $g \in K(f)$. From (f, g) and (g, h) we have that f already reaches h, thus adding the edge (f, h) does not introduce a cycle. When the trace ends with $(f, \mathsf{async}\ g)$, we are adding an edge (g, h) given an edge (f, h) and we need to prove that h does not reach g. Suppose the that h reaches g. This means that h also has to reach f. But by hypothesis the edge (f, h) is in the graph, which means that there is a cycle passing through h, reaching a contradiction. □

The deadlock avoidance problem checks whether a given blocking operation leads to a deadlock. A common way to handle this problem is by means of a Wait-For Graph [Knapp 1987], a graph whose nodes are tasks and whose edges are wait-for dependencies. In the context of this paper, a task f waits for a task g if f invokes a **`get`** operation on g. Then, checking for a deadlock corresponds to checking whether the Wait-For Graph is cyclic.

Let us relate Wait-For Graphs (WFGs) with the knowledge graph introduced before. When the edges in the WFG only target known tasks, the WFG is a subgraph of the knowledge graph (as the latter contains all possible known joins). Since the knowledge graph is acyclic (theorem 3.3), we know that the WFG is acyclic. Using Known Joins, the deadlock avoidance algorithm becomes a simple membership test: task f can perform `get` g if, and only if, f knows g. If the test fails we cannot be sure whether the WFG is acyclic, in which case the unsafe **`get`** may be handled by a standard cycle-based deadlock avoidance. Testing for known tasks in this way corresponds to checking whether the WFG is a subgraph of the knowledge graph. Section 4.3, and in particular definition 4.7, makes precise the relationship between Wait-For graphs and traces.

Revisiting our running example, recall that $\vdash t_2 : K_2$. We can build the knowledge graph of the trace, $\mathrm{kg}(K_2)$, as the pair $(\{f, g, h\}, \{(f, g), (f, h), (h, g)\})$. Consider a deadlocked Wait-For Graph G obtained from an execution of the code in listing 2.2, namely $G = (\{g, h\}, \{(g, h), (h, g)\})$. Since (g, h) is not an edge in $\mathrm{kg}(K_2)$, task g cannot wait for task h according to KJ. In contrast, it is safe for task h to wait for g according to KJ.

Some traces are rejected by KJ and yet are deadlock free.[3] For instance, we can remove the **`get`** operation in task h of listing 2.2 so that there is no possibility for deadlock. Trace t_2 is a feasible trace of the program below, but the **`get`** in task g is still considered unsafe according to KJ.

```
// Task f
shared future<int> x, y;
x = async {return y.get();};      //Task g
y = async {return 0;};            //Task h
```

Can we characterize the class of programs accepted by KJ? When considering our stream language, section 4 precisely characterizes the set of programs accepted by KJ as the set of data-race-free programs. The benefit of using KJ is that the majority of the tests will succeed and conclude that joining is safe, since most code respects our joining policy, cf. the empirical data in section 6.

[3] In the implementation, such cases are handled by cycle-based deadlock avoidance.

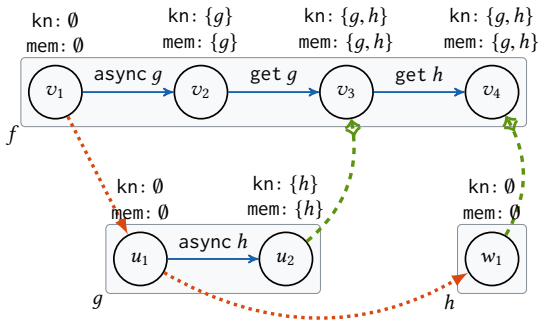

Fig. 3. Computation graph of the code in listing 2.1.

4 DATA-RACE FREEDOM IMPLIES DEADLOCK FREEDOM

Known Joins imposes a discipline on the communication of task names via forking and joining. Since tasks can also communicate through shared memory, the question arises as how do shared memory accesses affect knowledge. In this section, we relate data-race-free accesses with the preservation of KJ. The standard model to reason about data races at runtime is through causality analysis, which can be modeled with computation graphs. We opted by initiating our formal development with traces since it provides a simpler setting that is sufficient to power our tool. Computation graphs are not computed at runtime; they are however needed to show one of the main results, namely that data-race freedom implies deadlock freedom (theorem 4.6). Section 4.1 introduces computation graphs, section 4.2 shows how to extract a computation graph from a trace, and section 4.3 presents the main result of this paper: DRF implies DF (corollary 4.8).

4.1 Computation Graphs

Lamport [1978] pioneered the idea of using a partial order to reason about causality among concurrent events. Such a partial order is often called a happens-before relation. The intuition behind the idea that an event u happened before an event v is that the former caused, or influenced, the latter [Schwarz and Mattern 1994]. In distributed systems, researchers developed several representations for causality relations [Baquero and Preguiça 2016], including vector clocks [Fidge 1988; Mattern 1989] and causal histories [Birman and Joseph 1987].

Researchers on the dynamic analysis of parallel and multithreaded systems introduced graph representations of the happens-before relation, and called them computation graphs [Blumofe and Leiserson 1998], computation DAGs [Cormen et al. 2009], or partial order execution graphs [Dinning and Schonberg 1991; Mellor-Crummey 1991]. The nodes of these graphs denote sequential computation steps; the edges describe scheduling constraints, in such a way that an edge (u, v) indicates that node u must execute before node v. We say that u happens before w if there is a path connecting u to w.

We now introduce the notion of computation graphs for a programming model with futures. As an example, from the code in listing 2.1 we can extract the computation graph illustrated in figure 3. Exactly how is made explicit in the next subsection. Nodes in computation graphs contain information on the task the node belongs to, the set `kn` of known tasks, and the set `mem` of values accessible in the local memory of the node. There are three different kinds of edges:

Continue edges capture the sequencing of steps within a task. These are graphically depicted by solid arrows. All steps in a task are connected by continue edges. For instance, nodes v_1

and v_2 of figure 3 represent the execution of statement `future<future<int>> a = async {...}` in line 2 of listing 2.1; node v_1 represents the event before running the statement and node v_2 represents the event when the statement completed.

Fork edges represent the parent-child relationship among tasks, and are depicted by dotted arrows. When a task forks another task from a given step v, we draw an edge from v to the first node of forked task. For instance, since node v_1 forks a new task, the graph in figure 3 includes a fork-edge from node v_1 to u_1 representing the execution of statement `future<future<int>> a = async {...}` in line 2 of listing 2.1; node v_1 is the step before forking task g and step u_1 is the start of task g.

Join edges represent synchronization among tasks, and are depicted by dashed lines. When a task performs a `get` on task f to end in step v, we draw an edge from the last node of f to step v. For instance, there is a join-edge from step u_2 to step v_3 in figure 3 that represent the execution of statement `future<int> c = a.get()` in line 5 of listing 2.1; node u_2 is the last step of task g and node v_3 is the continuation of the get operation, which started at v_2.

Given that the computation graphs for our programming model have a restricted structure, we define an algebra of computation graphs, cf. Mokhov and Khomenko [2014]. For the sake of analysis, each node is a triple that includes a task name, a set of known task identifiers, and a set of values that represents the local memory of the node. We denote by $v_{\mathtt{tid}}$, and $v_{\mathtt{kn}}$ and $v_{\mathtt{mem}}$ the three *projection* functions on node v. Let $F \subseteq \mathcal{F}$ be a set of task identifiers, and $D \subseteq \mathcal{D}$ be a set of values. The algebra of graphs is derived from the grammar below, where m denotes a memory operation as in the grammar of the trace language, section 3.1.

$$v ::= (f, F, D) \qquad \textit{Nodes}$$

$$G ::= \epsilon \mid G + v \mid G + u \xrightarrow{m} v \mid G + v \xleftarrow{\mathsf{fork}} u \xrightarrow{\mathsf{async}\ f} w \mid G + v \xrightarrow{\mathsf{get}\ f} u \xleftarrow{\mathsf{join}} w \qquad \textit{Graphs}$$

The graph constructors are subject to a few restrictions, namely: in $G + v$, node v does not occur in G; in all cases, nodes u and w are sink nodes in G (that is, they have zero outgoing edges); in addition, node w, the source of the `join` edge, does not contain outgoing m-labeled edges in G.

The usual notion of the happens-before relation can be easily defined on top of computation graphs.

Definition 4.1 (Nodes, Edges, Happens-before). The *nodes* in a term G are the set of subterms v of G; the *edges* in G are the set of subterms of G of the form $u \xrightarrow{l} v$ where label l is either `fork`, `join` or an operation o (as in the grammar in section 3.1), and the `fork` and `join`-labeled edges are re-oriented to become left-to-right. We then say that u *happens before* v in a computation graph G, notation $u \preceq_G v$, when $u = u_0 \xrightarrow{l_0} u_1 \xrightarrow{l_1} \cdots u_{n-1} \xrightarrow{l_{n-1}} u_n = v$ are edges in G and $n \geq 0$.

The following graph is graphically depicted in figure 3.

$$G_1 = \epsilon + v_1 + u_1 \xleftarrow{\mathsf{fork}} v_1 \xrightarrow{\mathsf{async}\ g} v_2 + w_1 \xleftarrow{\mathsf{fork}} u_1 \xrightarrow{\mathsf{async}\ h} u_2 + v_2 \xrightarrow{\mathsf{get}\ g} v_3 \xleftarrow{\mathsf{join}} u_2 + v_3 \xrightarrow{\mathsf{get}\ h} v_4 \xleftarrow{\mathsf{join}} w_1$$

The nodes of G_1 are $\{v_1, u_1, v_2, w_1, u_2, v_3, v_4\}$, and the edges of G_1 are $\{v_1 \xrightarrow{\mathsf{fork}} u_1, v_1 \xrightarrow{\mathsf{async}\ g} v_2, u_1 \xrightarrow{\mathsf{fork}} w_1, u_1 \xrightarrow{\mathsf{async}\ h} u_2, v_2 \xrightarrow{\mathsf{get}\ g} v_3, u_2 \xrightarrow{\mathsf{join}} v_3, v_3 \xrightarrow{\mathsf{get}\ h} v_4, w_1 \xrightarrow{\mathsf{join}} v_4\}$. Forking task g, represented by node v_1, must happen-before the execution of task h, represented by node w_1, since we have $v_1 \xrightarrow{\mathsf{fork}} u_1 \xrightarrow{\mathsf{fork}} w_1$. The first step of task g, node u_1, happens before the execution of get on task g in node v_3, given that $u_1 \xrightarrow{\mathsf{async}\ h} u_2 \xrightarrow{\mathsf{join}} v_3$. Finally, the continuation of forking v_2 and the first node of task g, node u_1 can run in parallel, as there is no happens before relation between these two nodes, that is $v_2 \not\preceq u_1$ and $u_1 \not\preceq v_2$.

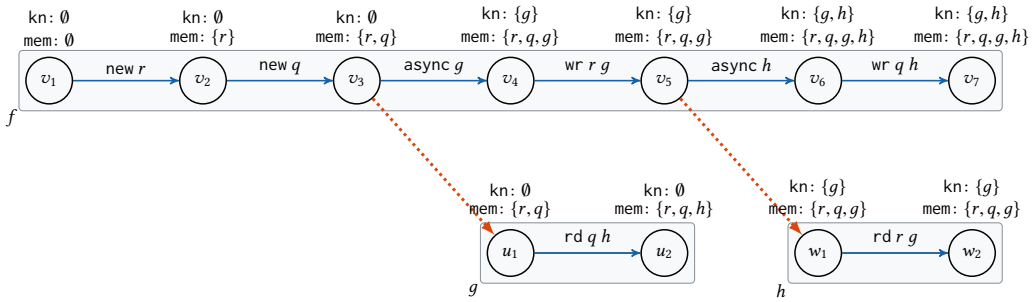

Fig. 4. Computation graph of the code in listing 2.2.

Definition 4.2 (Data visibility, Data-race freedom). We say that node v *reads* d *from* memory location r in graph G if $u \xrightarrow{\texttt{rd}\ r\ d} v$ is an edge in G. Similarly, we say that v *writes* d *to* r in G if $u \xrightarrow{\texttt{wr}\ r\ d} v$ is an edge in G. The *write-set* of a memory location r in graph G, notation $\texttt{W}_G(r)$, is the set $\{v \text{ is a node in } G \mid v \text{ writes } d \text{ to } r, \text{ for some } d\}$. When v writes to r or v reads from r we say that v *accesses* r. We say that there is a *data race* between two distinct nodes u and v accessing r in G, when at least one of these node writes to r, $v \not\preceq_G u$, and $u \not\preceq_G v$. We say that a computation graph is *data-race free* if for every pair of nodes v and u there is no data race between them. We say that *node* v *produces a visible side-effect* d *in* r if v writes d in r and every node in $\texttt{W}_G(r)$ happens before v. Finally, we say that *value* d *is visible in memory location* r if some node produces a visible side-effect d in r.

Figure 4 depicts a computation graph G_2 derived from the execution of the code in listing 2.2.

$$
\begin{aligned}
G_2 = \epsilon + v_1 + v_1 &\xrightarrow{\texttt{new}\ r} v_2 + v_2 \xrightarrow{\texttt{new}\ q} v_3 \\
+ u_1 &\xleftarrow{\texttt{fork}} v_3 \xrightarrow{\texttt{async}\ g} v_4 + v_4 \xrightarrow{\texttt{wr}\ r\ g} v_5 + w_1 \xleftarrow{\texttt{fork}} v_5 \xrightarrow{\texttt{async}\ h} v_6 + v_6 \xrightarrow{\texttt{wr}\ q\ h} v_7 \\
+ w_1 &\xrightarrow{\texttt{rd}\ r\ g} w_2 + u_1 \xrightarrow{\texttt{rd}\ q\ h} u_2
\end{aligned}
$$

In this graph, the write-set of r is $\{v_5\}$. There is a single data-race in G_2, the one between nodes v_7 and u_2, since both nodes are accessing r, v_7 is a write, and there is no path between these nodes ($u_2 \not\preceq v_7$ and $v_7 \not\preceq u_2$). Furthermore, node v_5 produces a visible side-effect g in r, as v_7 is the only write to r, which is observed (read) by node w_2.

4.2 Building Computation Graphs from Traces

We map a trace into a computation graph, in such a way that the computation graph captures the scheduling constraints of the trace. Function $[\![\cdot]\!]$, defined in figure 5, takes a trace and yields a computation graph.

We briefly describe the various rules. Rule C-NIL builds the empty graph ϵ from the empty trace ϵ. Rule C-INIT adds a node tagged with task name f, an empty knowledge, and an empty local memory. Memory operations (rules C-NEW, C-WRITE, and C-READ) produce continuation nodes the same task name f and same knowledge $v_{\texttt{kn}}$ (cf. rule T-MEM). Rule C-NEW ensures that the newly created location r is indeed new, and extends the local memory of the node with r. Rule C-WRITE ensures that location r and value d are already in the local memory of node v, and leaves the local memory unchanged. Rule C-READ says that reading extends the local memory with value d loaded from location r. The location from which f reads must be in the local memory of the node, $v_{\texttt{mem}}$.

$$\frac{}{[\![\epsilon]\!] = \epsilon} \qquad \frac{f \text{ does not occur in } t}{[\![t;(f,\texttt{init})]\!] = [\![t]\!] + (f, \emptyset, \emptyset)} \tag{C-NIL,C-INIT}$$

$$\frac{v_{\texttt{tid}} = f \qquad r \text{ does not occur in } t}{[\![t;(f,\texttt{new}\ r)]\!] = [\![t]\!] + v \xrightarrow{\texttt{new}\ r} (f, v_{\texttt{kn}}, v_{\texttt{mem}} \cup \{r\})} \tag{C-NEW}$$

$$\frac{v_{\texttt{tid}} = f \qquad r, d \in v_{\texttt{mem}}}{[\![t;(f,\texttt{wr}\ r\ d)]\!] = [\![t]\!] + v \xrightarrow{\texttt{wr}\ r\ d} (f, v_{\texttt{kn}}, v_{\texttt{mem}})} \tag{C-WRITE}$$

$$\frac{v_{\texttt{tid}} = f \qquad r \in v_{\texttt{mem}} \qquad d \text{ is visible in } r}{[\![t;(f,\texttt{rd}\ r\ d)]\!] = [\![t]\!] + v \xrightarrow{\texttt{rd}\ r\ d} (f, v_{\texttt{kn}}, v_{\texttt{mem}} \cup \{d\})} \tag{C-READ}$$

$$\frac{v_{\texttt{tid}} = f \qquad g \text{ does not occur in } t}{[\![t;(f,\texttt{async}\ f)]\!] = [\![t]\!] + (g, v_{\texttt{kn}}, v_{\texttt{mem}}) \xleftarrow{\texttt{fork}} v \xrightarrow{\texttt{async}\ g} (f, v_{\texttt{kn}} \cup \{g\}, v_{\texttt{mem}} \cup \{g\})} \tag{C-ASYNC}$$

$$\frac{v_{\texttt{tid}} = f \qquad u_{\texttt{tid}} = g \qquad g \in v_{\texttt{mem}}}{[\![t;(f,\texttt{get}\ g)]\!] = [\![t]\!] + v \xrightarrow{\texttt{get}\ g} (f, v_{\texttt{kn}} \cup u_{\texttt{kn}}, v_{\texttt{mem}} \cup u_{\texttt{mem}}) \xleftarrow{\texttt{join}} u} \tag{C-GET}$$

Fig. 5. Function $[\![\cdot]\!]$ builds a computation graph G from a trace t.

The standard notion of visibility (definition 4.2) ensures that a value d can only be read when the last write that happened before this read produced d. Rule C-ASYNC ensures that the forked task is labeled with a fresh identifier f. The continuation node of f extends its knowledge (as per rule T-ASYNC) and local memory with the new task identifier g. Rule C-GET states that when task g joins with task f, the value returned by future f must be in $v_{\texttt{mem}}$. Then the continuation node extends its local memory with that of $v_{\texttt{mem}}$. Similarly, it extends its knowledge with $u_{\texttt{kn}}$ (following rule T-GET).

The computation graphs associated with traces t_1 and t_2 are the graphs $G_1 = [\![t_1]\!]$ and graphs $G_1 = [\![t_2]\!]$, respectively, both presented in the previous section.

4.3 Results

The main result of this paper, corollary 4.8, states that data-race freedom implies deadlock freedom.

An important result says that if node v knows a task f and v happens before a node u, then u also knows f.

LEMMA 4.3. *If $[\![t]\!] = G$, $f \in v_{\texttt{kn}}$, and $v \preceq_G u$, then $f \in u_{\texttt{kn}}$.*

PROOF. The proof follows by induction on the structure of trace t. We show that for each edge $v \xrightarrow{l} u$ in graph $[\![t]\!]$ if $f \in v_{\texttt{kn}}$, then $f \in u_{\texttt{kn}}$, which holds since in every rule of figure 5 the outgoing nodes of each new edge extend the knowledge of the source node v. □

We can now use lemma 4.3 to establish another important result: in a DRF computation graph, any task in the memory of a node is known. Such result can be proved if we simultaneously show that any visible task name is known by the writer node.

LEMMA 4.4. *Let $[\![t]\!] = G$ be data-race free graph.*

(1) *If $v \in G$ and $g \in v_{\texttt{mem}}$, then $g \in v_{\texttt{kn}}$.*

(2) *If v produces a visible side-effect f in r, then $f \in v_{\mathsf{kn}}$.*

Proof. The proof of the two cases is by induction on the structure of the trace t. We show a few cases.

Part (1). When $[\![t]\!] = [\![t']\!] + v \xrightarrow{\mathtt{rd}\ r\ g} u$, given $g \in u_{\mathtt{mem}}$ we want to show that $g \in u_{\mathsf{kn}}$. From Rule C-read, we have that some node w produced a visible side-effect g in r. Hence, from (2) we have that $g \in w_{\mathsf{kn}}$. But since $[\![t]\!]$ is DRF, and nodes w and u are accessing r, then we have that $w \preceq_G u$. So, from $g \in w_{\mathsf{kn}}$, $w \preceq_G u$, and lemma 4.3, we have that $g \in u_{\mathsf{kn}}$.

Part (2). When $[\![t]\!] = [\![t']\!] + v \xrightarrow{\mathtt{wr}\ r\ g} u$, we have that u writes g visible in r and we are showing that $g \in u_{\mathsf{kn}}$. From Rule C-write, we have that $g \in u_{\mathtt{mem}}$. And, from (1), since we have $g \in u_{\mathtt{mem}}$, then we have $g \in u_{\mathsf{kn}}$, thus concluding this case. When $[\![t]\!] = [\![t']\!] + v \xrightarrow{\mathtt{rd}\ r\ d} u = G$, we have that w produces a visible side-effect g in r, and we want to show that $g \in w_{\mathsf{kn}}$. From Rule C-read we have there exists some node w' that writes d in r. But since $[\![t]\!]$ is DRF, we can show that the visible side effects to r are unique, which means that $w' = w$ and $d = g$. Finally, we show that w produces the same visible side-effect in $[\![t']\!]$, and conclude with the induction hypothesis. □

Next, we establish a relation between the knowledge in a DRF graph and $\vdash t : K$.

Lemma 4.5. *Let t be a trace such that $[\![t]\!]$ is data-race free and $\vdash t : K$. Then $v_{\mathsf{kn}} \subseteq K(v_{\mathtt{tid}})$, for all node v in $[\![t]\!]$.*

Proof. The proof is by induction on the structure of trace t. □

Theorem 4.6 (DRF implies KJ). *If $[\![t]\!]$ is data-race free then $\vdash t : K$, for some K.*

Proof. The proof is by induction on the structure of trace t. The interesting case is when t is of the form $t'; (f, \mathtt{get}\ g)$, hence $[\![t]\!] = [\![t']\!] + v \xrightarrow{\mathtt{get}\ f} u \xleftarrow{\mathtt{join}} w$ where $v_{\mathtt{tid}} = g$ and $f \in v_{\mathtt{mem}}$ (w.r.t. Rule T-get). We want to show that $\vdash t'; (g, \mathtt{get}\ f) : K[g := K(f) \cup K(f)]$, given that $\vdash t' : K$ is our induction hypothesis. Applying T-get we are left with showing that $f \in K(f)$. From the hypothesis that $f \in v_{\mathtt{mem}}$ and Lemma 4.4 we have that $f \in v_{\mathsf{kn}}$. So, since $g = v_{\mathtt{tid}}$ and $f \in v_{\mathsf{kn}}$, then can conclude with Lemma 4.5. □

It is important to note that the contrapositive of theorem 4.6 can be read as: if KJ fails, then DRF fails. But since not-DRF means that there is a data race, we conclude that *if KJ fails, then there is a data race.*

To discuss deadlock avoidance, we concentrate ourselves on a certain class of wait-for graphs, namely *well-formed graphs*. The intuition is that, in a well-formed WFG, a task f can only await task g if g is in the local memory of f. Since computation graphs record memory operations we have enough information to specify the meaning of well formed graphs.

Definition 4.7 (Well-formed WFG). We say that (V, E) is well-formed WFG w.r.t. a trace t, if $(f, g) \in E$ implies that there is some sink node $v \in [\![t]\!]$ such that $v_{\mathtt{tid}} = f$ and $g \in v_{\mathtt{mem}}$.

Corollary 4.8 (DRF implies DF). *Let (V, E) be a well-formed WFG w.r.t. trace t. If $[\![t]\!]$ is data-race free, then (V, E) is acyclic.*

Proof. From theorem 4.6, we have that $\vdash t : K$. Next, we show that $(V, E) \subseteq \mathrm{kg}(K)$: since (V, E) is well formed. If f joins with g, then from lemma 4.4 there exists some node v such that $v_{\mathtt{tid}} = f$ and $g \in v_{\mathsf{kn}}$. Thus, from theorem 3.3 we have that (V, E) is acyclic. □

The contrapositive of corollary 4.8 lets us conclude that *if a program deadlocks, then there must have been a data race during its execution.* This means that we now have two methods of proving

that a computation graph, say $[\![t_2]\!]$, has a data race: either by providing an evidence of a data race (definition 4.2), e.g., between nodes v_7 and u_2, or by applying the contrapositive of corollary 4.8 to a cyclic, well-formed WFG (definition 4.7). For instance, the following WFG

$$(\{g, h\}, \{(g, h), (h, g)\})$$

is cyclic and well-formed with respect to $[\![t_2]\!]$, thus $[\![t_2]\!]$ has a data race. The WFG is well formed because edge (g, h) is derived from $(u_2)_{\texttt{tid}} = g$ and $h \in (u_2)_{\texttt{mem}}$, and edge (h, g) is obtained from $(w_2)_{\texttt{tid}} = h$ and $g \in (w_2)_{\texttt{mem}}$.

5 GORN: EXTENDING HABANERO-JAVA WITH KNOWN JOINS CHECKS

This section introduces Gorn, our implementation of Known Joins. First, we discuss the integration of Gorn in the Habanero Java language, and then delve into implementation details on checking the Known Joins property.

5.1 Integrating Gorn in Habanero Java

We recall that the result that data-race freedom implies KJ only holds for programs that use futures as the sole synchronization mechanism. Habanero-Java programs that synchronize via isolated blocks or volatile variables void such guarantee. However, the result that KJ implies deadlock freedom can be extended to the deadlock-free programming model of Habanero.

Integrating Gorn in Habanero Java (HJ) corresponds to extending the parallel runtime so that it includes calls to our verification algorithm. We integrated Gorn in two different flavors of HJ, so as to maximize the set of programs that can be checked: Habanero-Java [Cavé et al. 2011], an extension of the Java language with the Habanero primitives as language constructs, and hj-lib [Imam and Sarkar 2014], a Java 8 library that offers the Habanero primitives as library calls. Every HJ program runs as-is in a HJ runtime verified by Gorn. Deadlocks are treated as runtime errors, so tasks may catch the exception and try to recover from deadlocked situations.

Deadlock avoidance. The deadlock-free Habanero programming model consists of the following synchronization constructs. When adding futures to such a programming model, deadlocks can only arise from the interaction among future tasks.

Isolated blocks: instruction `isolated B` runs a block of code `B` atomically. Isolated blocks define critical sections.
Deadlock safety: The code block `B` cannot invoke (directly or indirectly) blocking operations. Thus, this form of mutual exclusion cannot deadlock against other synchronization mechanisms.

Phasers: represent a logical group of tasks that synchronize together repeatedly in a stepwise fashion, generalizing the classical all-to-all barrier synchronization. The set of tasks partaking in a phaser (i.e., its members) can change dynamically. Synchronization advances stepwise in all-to-all synchronization. The static method `nextAll` synchronizes the execution of the members of a phaser: it blocks until every member invokes `nextAll`, or leaves the group.
Deadlock safety: Instruction `nextAll` synchronizes on all phasers the task is a member of to avoid deadlocks among different groups of tasks (phasers). Additionally, futures cannot be registered with phasers, forbidding their interaction.

Finish blocks: instruction `finish B` executes a block of code `B` and declares a syntactic scope at the end of which the current task awaits the termination of any task forked within `B`. This includes any descendant of the task executing `B`. Finish blocks can be nested and allow programmers to write fork-join computation, e.g., recursive divide and conquer.
Deadlock safety: Finish blocks do not interact with futures nor with mutual exclusion. A

Table 1. Comparing worst-case time and space complexities, where n is the number of tasks forked.

	Vector Clock	**Snapshot-Set**
Time before fork	$O(n)$	$O(1)$
Time after fork	$O(1)$	$O(1)$
Time before join	$O(n)$	$O(n)$
Time after join	$O(n)$	$O(1)$
Space	$O(n^2)$	$O(n)$

task drops all phasers at the end of a finish block, avoiding deadlocks between these two mechanisms.

Leveraging the finish construct. For programs checked by Gorn, a finish block acts as a scope of verification: any history captured in the block is garbage collected at the end of the scope. This lets the programmer trim the size of the history (known-set). In section 6.2, we change a benchmark program and surround a computation stage with a finish block, thus lowering the memory usage overhead of verification.

Integration details. The HJ compiler adds calls to Gorn runtime as a bytecode-level transformation pass implemented on HJ's Parallel Intermediate Representation (PIR) [Nandivada et al. 2013]. The PIR extends Soot's Jimple IR [Vallée-Rai et al. 1999] with parallel constructs such as async, finish, and future. The instrumentation pass adds the necessary calls to Gorn at future task forks, future get operations, and at the start and end of finish blocks. In the HJ-LIB version, the instrumentation pass is not needed, as there is no code generation stage. Instead, the various calls to Gorn are included in the runtime itself.

5.2 Implementation

As an HJ program runs, Gorn performs the following four kinds of operations: i) before forking, the parent task copies its known tasks to the child task; ii) after forking, the parent task registers the child in its known tasks; iii) before task f joins with task g, a membership test of task g in the known tasks of task f is performed; and iv) after task f joins with task g, task f merges its known tasks with that of g.

When KJ is false Gorn notifies the user of a data race and then invokes Armus [Cogumbreiro et al. 2015], a standard deadlock avoidance algorithm. In case of a deadlock, Armus throws an exception, disallowing the task from entering the deadly embrace; the user may catch the exception to recover from the synchronization error. Additionally, before testing KJ, our implementation tests if the task to join is still alive; terminated tasks do not introduce deadlocks and are therefore ignored. This means that a task only checks if it knows another task *at most* once. Gorn can also be run in a strict-mode, where any get of an unknown task throws an exception, i.e., when KJ is false an exception is raised and Armus is *not* invoked.

We present two implementations of KJ: one uses vector clocks and another one uses snapshot-sets (a data structure we introduce). Table 1 compares the worst-case time and space complexities of the two algorithms. We experimented with other algorithms, namely interval-tree clocks [Almeida et al. 2008], but their performance was unsatisfactory.

Implementing KJ with vector clocks. A node v can join with task g if the continuation of forking g happens-before node v.

Definition 5.1 (Can-join). Node u is the *fork-point* of task f in a graph G if $v \xrightarrow{\text{async } f} u \in G$. Node w *can join* with task f if v is the fork-point of f and $v \preceq_G w$.

Theorem 5.2. *Let* $\vdash t : K$ *and* $v \in [\![t]\!]$. *Then* $f \in K(v_{\text{tid}})$ *if and only if* v *can join with* f.

Theorem 5.2 allows us to implement KJ using vector clocks. The importance of this result is twofold: sets an expectation on the time and space bounds of the problem, and lets us benefit from the vast literature on causality analysis. A vector clock is an efficient representation of the causality relation, hence equivalent to using computation graphs, see Schwarz and Mattern [1994, Theorem 3.3]. This means that we can implement definition 5.1 to test (using e.g., vector clocks) whether f can join with g, as it can be formulated in terms of a happens-before relation.

A vector clock is a map from tasks to integers that represents (i) a node in the computation graph, and (ii) the reachability relation of the predecessors of (i). The worst-case time complexity for vector clocks is linear in the number of tasks, while for graphs is quadratic in the number of nodes, cf. section 7. The lower bound of the number of nodes is always the number of tasks. The worst-case space complexity for vector clocks is quadratic in the number of tasks; for graphs is linear in the number of nodes. The rules for maintaining causality using clocks are standard [Flanagan and Freund 2009].

Implementing KJ with snapshot-sets. This implementation pairs a hash-set of *children* with the *joins*, a binary tree of sets of tasks representing unions. The algorithm snippets are written in a Haskell-inspired language. A `KnownSet` pairs two fields: field `forks` is the set of (forked) tasks of type `Children`, and field `joins` holds a join tree. A branch in a join tree holds either a set of children or merges join trees.

```
data Children = Set Task
data Joins = Leaf Children | Branch Joins Joins
type KnownSet = {forks::Children, joins::Joins}
```

Before task f forks task g. The set known tasks of g is initialized with a copy of the tasks known to f: the forks of the task start empty, and the joins compose a copy of the forks of the parent, `copy ks.forks`, with the joins of the parent task.

```
child :: KnownSet -> KnownSet
child ks = {forks: empty, joins: Branch (Leaf (copy ks.forks)) ks.joins}
```

The worst-case time complexity for this operation depends on the worst-case time complexity of copying the hash-set `ks.forks`. Instead of duplicating `forks`, our implementation of the `copy` function creates a snapshot in constant time (see below).

After task f forks task g. Task f updates its known tasks using function `add`, which consists of adding task g to its children set `ks.forks` using the set **`insert`** operation.

```
add :: KnownSet -> Task -> KnownSet
add ks f = {forks: insert f ks.forks, joins: ks.joins}
```

Before task f joins with task g. Before joining, task f tests if g is a member of its known tasks using function `contains`. The search for task g starts in children set `ks.forks` and continues depth-first through the join tree `ks.joins`, using function `isIn`. The navigation order is important: a depth-first order favors reaching tasks available via the most recent joins first, which matches the usual synchronization patterns.

```
contains :: KnownSet -> Task -> Bool
contains ks f = f `member` ks.forks || isIn ks.joins
  where isIn (Leaf s) = f `member` s
        isIn (Branch l r) = isIn l || isIn r
```

The complexity of searching a tree is $O(n)$ where n is the number of tasks in the tree. The tree might contain multiple paths to the same task, arising from concurrent joins. The search algorithm avoids revisiting nodes by maintaining a set of traversed nodes.

After task f joins with task g. Task f extends its known tasks using function `union`, which branches to the join-tree `joins` of task g (discussed below). This is a constant-time operation.

```
union :: KnownSet -> Join -> KnownSet
union ks j = {forks: ks.forks, joins: Branch j ks.joins}
```

Before terminating, each task converts its known tasks `ks` to a join-tree with function `convert`.

```
convert :: KnownSet -> Joins
convert ks = Branch (Leaf ks.forks) ks.joins
```

Concurrent hash-sets with constant time copying. We introduce a novel technique called *snapshot-sets*. There are two data structures to consider: a *timed set* that labels the members of a set with logical time, and a *snapshot* (an immutable (cheap) copy of a timed set). In a timed set, the time increases monotonically as elements are added. A snapshot consists of a reference to a timed set and the logical time of the last added element at the time of creation—in our code snippets function `copy` should yield a snapshot. Any member of a timed set whose time is within the snapshot's time is also a member of the snapshot.

In our implementation the set of children is a timed set. When forking a task the parent passes a snapshot of its children set to the forked task. A task and its descendants can access the same timed set concurrently: a parent can extend its child set while a descendant runs a membership test. Any task added to the timed set will not be visible in existing snapshots. When a membership test reads a newer value, the snapshot's time will be smaller than the new element's time, which makes the new contents invisible.

Implementing a timed set with a hash map. We designed a hash map that allows for one writer and multiple readers to run concurrently under the usage constraints of Gorn. The elements of the hash map are immutable pairs, consisting of a task name and an integer. Our hash map implementation uses open addressing and only supports two operations: put and lookup. Adding a member to a timed set corresponds a put in the hash map. Testing the membership of an element corresponds to a lookup in the hash map. An add operation (the writer) can run concurrently with one or more membership tests (the readers), because they operate on disjoint portions of the data structure.

Synchronization guarantees. While we have seen that there is no need for explicit synchronization, we might still wonder whether there needs to exist some form of stronger memory model guarantees. Our implementation of snapshot-sets does without compare-and-swap operations, atomics, or volatiles. When forking a task there is a happens-before relation between the parent and the child, which, according to Java's memory model [Manson et al. 2005], guarantees that the child can observe every element in the timed set *at the time of forking*. While any new elements are not guaranteed to be observed by the descendants, these do not affect the membership tests because the timestamps of the new elements are greater than the timestamp of the snapshot (hence skipped by the membership test).

The only two possible sources of inconsistency between writer and readers come from: (i) re-creating the backed array of the hash map, and (ii) re-creating a bucket. The members of the hash map are thread-safe because these are immutable. We address the writer-readers inconsistencies from (i) and (ii) with the copy-on-write pattern, therefore enforcing that concurrent reads always target a consistent backed array. For instance, the writer rehashing a map first creates a new backed-array, then the writer copies the map's contents to the new array, and, finally, the writer switches the backed-array reference to the new array. The Java memory model guarantees that references are written and read atomically.

6 EVALUATION

In this section we use Gorn to check a data-set of ~2.300 student's assignments, and measure the runtime overhead of the verification. We also measure runtime and memory overheads for five benchmarks. These benchmarks explore how Gorn scales in worst-case scenarios. The overhead of deadlock avoidance depends on the number of forks, joins, and the structure ("shape") of the computation graphs. Therefore, we favored benchmarks with distinct synchronization patterns, yielding computation graphs of different shapes. One of the selected benchmarks forks and joins close to 1 million tasks, as to explore the limits of the verification.

The KJ algorithm is validated against a test suite, which consists of 141 test cases specifically written for Gorn, checking the legal and disallowed joins of each task (scenarios include recursive divide-and-conquer; multiple forks followed by multiple joins; forks interleaved with joins). Additionally we run Gorn on HJ-LIB's test suite, which consists of 387 test cases, including 12 deadlocking programs.

6.1 Checking a Dataset of Student Submissions

Gorn verified a large corpus of parallel HJ programs written by undergraduate students in an introductory parallel programming course. These programs are collected from an auto-grading system, meaning that we have access to both final submissions and works-in-progress. The corpus contains ~26,000 student submissions from ~40 assignments by ~250 students. We restricted the dataset to submissions that make use of futures, totaling 2,277.

Our goals with this investigation are two-fold. First, we can think of few better stress tests for Gorn than applying it to a dataset of chaotic, partial, and often buggy student programs that create large numbers of tasks. Testing across such a diverse dataset increases our confidence in the stability of Gorn as a software artifact and in its lack of false positives. Second, we demonstrate that deadlocks are a real problem for students learning parallel programming for the first time, reinforcing the value of Gorn as a pedagogical tool.

In total, during our testing we found one final submission by a student for which Gorn identified and reported a deadlock. The assignment is to write a space- and time-efficient parallel implementation of Smith-Waterman's genome alignment algorithm, given a sample sequential version. This is a non-trivial task and requires a major re-architecting of the sequential algorithm to support processing of an alignment problem that does not fit in memory while still extracting enough parallelism for a 16-core systems. In general, top students in the class achieve 10–12× speedup over the reference sequential implementation. Testing requires ~160 seconds for the fastest submissions.

We examined the deadlocking submission in detail. The student's commit messages and report indicate that while she knew her program had an issue that was causing it to run longer than expected, she was unable to diagnose it in time for the assignment deadline and instead blamed the auto-grading system. The student's report incorrectly states that the program is data-race free. Gorn's diagnostic information would have greatly eased resolving this deadlock, and might have resulted in the student gaining full credit on the assignment. First, Gorn's early detection and abort

Table 2. List of benchmarks evaluated. #Asyncs is the dynamic number of asyncs performed. #Gets is the dynamic number of gets performed.

Benchmark	Input Size	#Asyncs	#Gets
Jacobi	8192×8192	15,872	37,696
Smith-Waterman	21,000	1,600	4,641
Crypt	(Size C) 50,000,000	16,384	16,384
Strassen	4096×4096	30,811	44,816
Series	(Size C) 1,000,000	999,999	999,999

behavior would have avoided the student having to wait for the deadlocking program to hit the autograder's 40 minute timeout, speeding up development iterations. In this particular submission, it takes 3 seconds to reach the deadlock. Second, with Gorn, diagnosing and fixing the deadlock is simplified, as the error message includes file location of the unknown join, and the tasks involved in the deadlock. We have checked 2,277 programs and found no example that was both rejected by KJ and deadlock free.

Example of the Deadlock. The code excerpt below exhibits the deadlock discovered by our tool. At the `i`-th iteration a task stores a future in `S[i]`; this future awaits futures `f1`, `f2`, and `f3` (line 87). The deadlock occurs when future `f1` awaits the future in `S[i]` which, in turn, awaits the termination of `f1`. Future `S[i]` knows `f1`, but future `f1` does *not* know `S[i]`, hence the **`get`** in line 82 raises an exception.

```
76 for (int i = 1; i <= yLength; i++) {
77   // ...
78   final Future<Integer> f1 = async{
79     return S[i - 1].get() + getScore(charMap(XChar), charMap(YChar)));
80   };
81   final Future<Integer> f2 = async{
82     return S[i].get() + getScore(charMap(XChar), 0));
83   };
84   final Future<Integer> f3 = async {
85     return S[i - 1].get() + getScore(0, charMap(YChar)));
86   };
87   S[i] = async { return Math.max(f1.get(), Math.max(f3.get(),
         f2.get())); };
88 }
```

The error message consists of a stack trace, from which we elide the internal calls of the parallel runtime. Recall that these errors can be caught by tasks.

```
1  gorn.DeadlockIdentifiedException: [Event[synch=ParallelFuture[id=65],
       phase=1], Event[synch=ParallelFuture[id=67], phase=1]]
2  [...]
3  at SparseParScoring.lambda$scoreSequences$13(SparseParScoring.java:82)
4  [...]
```

6.2 Verification Overhead

We compare the baseline (a program run without verification) against a program running with deadlock avoidance and: KJ disabled (KJ OFF), KJ enabled with vector clocks (KJ VC), and KJ enabled with snapshot sets (KJ SS). As explained in section 5.2, when KJ is enabled Armus is still maintaining the blocked status of each task, but cycle detection only runs in case of a data race. Also, as mentioned in the same section, the search algorithm for KJ SS avoids revisiting nodes by maintaining a set of traversed nodes. The results shown in this section were obtained without this optimization. However, we subsequently obtained execution times for KJ SS with and without this optimization, and saw no measurable difference between these two cases, thus indicating that maintaining the set of traversed nodes does not improve performance for these benchmarks. Our experiments are conducted on a 16-core Intel Ivybridge 2.6 GHz system with 48 GB memory, running Red Hat Enterprise Linux Server release 7.1, and Sun HotSpot JDK 1.7.

The benchmarks used in the evaluations are summarized in table 2. The columns list the benchmark name, the input size used for the evaluation, the number of tasks dynamically created, the number of joins performed by each of the programs. Crypt and Series were derived from the original versions in the Java Grande Forum benchmark suite [Smith et al. 2001]. Jacobi and Strassen were translated by the authors from OpenMP versions of those programs in the Kastors benchmark suite [Virouleau et al. 2014]. Futures are more general than deadlock-free group termination constructs like finish, and are used to express point-to-point synchronizations in all our benchmarks. Using finishes instead could lead to a loss of parallelism relative to the future-versions of these benchmarks.

Jacobi performs a 2 dimensional 5-point stencil computation on a 8192×8192 matrix, where each future task computes a 512×512 sub-matrix. This benchmark is chosen to demonstrate the performance impact of KJ on programs with complex point-to-point synchronization arising from the use of futures. These kind of computation graphs cannot be generated using structured task-parallel constructs such as async-finish or fork-sync. In the presence of such point-to-point synchronization, tasks have distinct non-null history information. *Synchronization pattern:* point-to-point synchronization between sibling tasks.

Smith-Waterman performs sequence alignment of two sequences of length 21,000. The alignment matrix computation is performed by 40×40 future tasks. With a different dependence pattern from Jacobi, this benchmark exhibits complex point-to-point synchronization and demonstrates the generality of computation graphs that can be generated using futures. Unlike Jacobi which makes multiple passes over the matrix, this benchmark performs its computation in a single pass through the alignment matrix. *Synchronization pattern:* point-to-point synchronization between sibling tasks.

Crypt Adapted from JGF [Smith et al. 2001], this benchmark performs IDEA encryption and decryption. Both encryption and decryption steps are parallelized using futures. The main program first performs encryption by forking a set of tasks and joins with the set of forked tasks. Next, it performs decryption by again forking a set of tasks and then joins with the forked tasks. This benchmark demonstrates the performance impact of having large history information. During the decryption step, the known set corresponding to the main task is huge due to the joins performed by it after the decryption step. *Synchronization pattern:* consecutively runs a series of forks followed by a series of joins.

Strassen performs multiplication of 2 matrices of 4096×4096 floating point numbers using a recursive divide and conquer approach with futures. The implementation uses a recursive cutoff of 128×128, i.e., sub-computations of this smaller size are performed sequentially. Recursive divide and conquer is a very common paradigm found in task-parallel programs and therefore it is important

Table 3. Comparison of overhead for deadlock avoidance, on 16 cores. The Baseline column lists absolute values of the unchecked execution: time in seconds and space in gigabytes. The next three columns list the ratio between verified and baseline. Armus is always enabled. In KJ OFF, KJ is disabled, so only Armus runs. In KJ VC, KJ is enabled using vector clocks. In KJ SS, KJ is enabled using snapshot-sets.

Benchmark	Baseline	KJ OFF	KJ VC	KJ SS
Jacobi	8.12 s	5.84×	1.10×	0.99×
	0.31 GB	1.08×	1.00×	1.00×
Smith-Waterman	4.70 s	1.80×	0.96×	0.96×
	3.61 GB	1.02×	1.00×	1.00×
Crypt	0.77 s	193.25×	10.15×	1.04×
	0.29 GB	1.48×	16.90×	1.08×
Strassen	3.13 s	28.49×	0.99×	1.07×
	5.48 GB	1.63×	1.01×	1.28×
Series	37.55 s	-	1.00×	1.06×
	1.00 GB	-	2.02×	2.34×

to study the performance impact of KJ on such programs. *Synchronization pattern:* a tree-shaped computation graph.

Series. Adapted from JGF and computes the first Fourier coefficients of the function $f(x) = (x+1)^2$ on the interval 0,2. The most time consuming component of the benchmark is the loop over the Fourier coefficients. Each iteration of the loop is independent of every other loop and the work may be distributed simply between processors. This benchmark is chosen to demonstrate the scalability of KJ since it creates 1 million tasks. *Synchronization pattern:* one task performs multiple forks followed by multiple joins.

Summary. Table 3, figure 6, and figure 7 compare the runtime and memory overhead of using the different deadlock avoidance algorithms. Table 3 serves as a bird's-eye view of the data, whereas the bar charts compare the two algorithms implementing Known Joins against the baseline. To reduce the impact of JIT compilation, garbage collection and other JVM services, we report the steady state mean execution time of 30 runs repeated in the same JVM instance for each data point, as recommended by Georges et al. [2007]. In short, there are three takeaways. First, the evaluation shows that runtime and memory overheads for deadlock avoidance are lower (better) with KJ enabled, as opposed to standard cycle detection (KJ OFF). Second, verification using snapshot-sets offers a better memory usage in average than using vector clocks. Third, vector clocks exhibit a reasonable performance, and benefit from being a well-understood data structure, specially when verifying programs with finish blocks.

Runtime overhead. When KJ is enabled, VC and SS have comparable overheads for all benchmarks except Crypt, where the overhead is much higher for KJ VC. Crypt runs two stages in sequence, first it performs encryption, then it performs decryption. At each stage a main task forks a group of child tasks and then awaits the completion of these children. If we refer back to table 1, we note that forking and merging are linear operations when implemented with vector clocks. This means that at the second stage, the vector clock of the main task has the size of the number of tasks forked when encrypting; such a vector clock is then copied to each child and merged when joining each

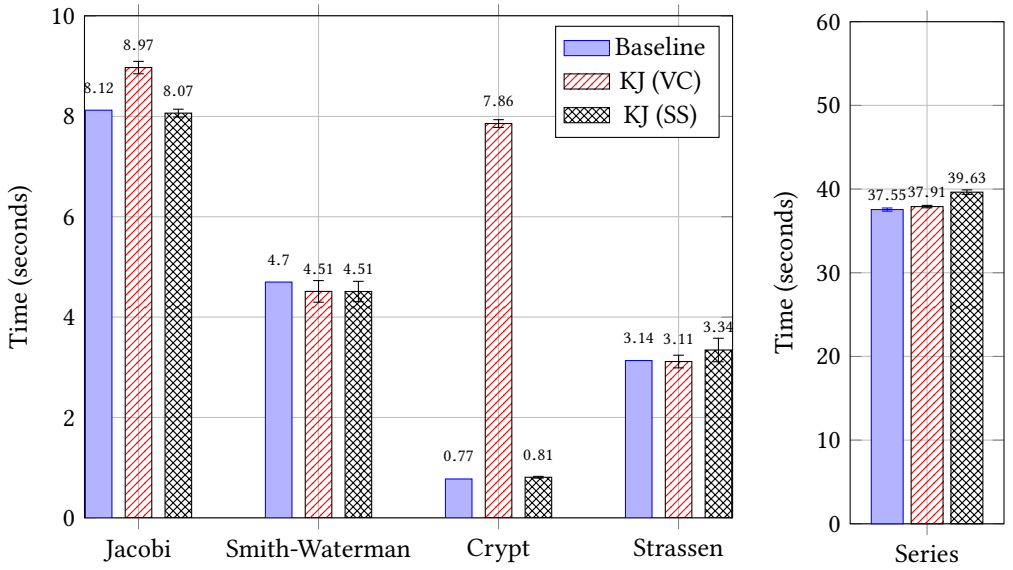

Fig. 6. Average execution times in milliseconds and 95% confidence interval of 30 runs for Baseline, KJ (VC) and KJ (SS) on 16 cores.

child, both in linear time. If we change the program and surround each stage with a finish block, then Gorn is able to reclaim the memory at the end of each stage, so enabling the overheads of VC match the ones of SS. The execution of the Series benchmark did not complete in 4 hours when performing deadlock avoidance without KJ; enabling KJ yields a runtime overhead of 1.06× for SS and a statistically negligible overhead for VC. This is because in standard deadlock avoidance, before blocking each task performs cycle detection that is quadratic in the number of tasks running concurrently, which, in the case of Series, can be up to one million tasks.

Memory overhead. To compute the memory usage, we created a daemon thread that periodically invokes the `totalMemory()` and `freeMemory()` methods from class `java.lang.Runtime`. The difference between the total memory available and the free memory approximates the memory usage of the application at a particular instant. We run the benchmarks 30 times on the same JVM instance. Average memory usage shows the arithmetic mean of the memory usage of the program, sampled every 100 milliseconds. We plot the memory usage between the baseline, KJ (SS), and KJ (VC) in figure 7.

The results indicate that enabling KJ reduces the memory overhead of deadlock avoidance. The benchmarks use exclusively futures to synchronize, so Gorn cannot take advantage of finish blocks to garbage-collect verification and better use memory. The outcome is that Gorn can verify these programs with a negligible memory overhead. For snapshot sets the memory overheads range from 1.04x to 2.34× with snapshot sets. For vector clocks, we have the case of Crypt where the memory overhead is 16.90×, which suggests the limitation of having a worst-case $O(n^2)$ space requirement for vector clocks, versus a $O(n)$ for snapshot sets. To illustrate how finishes can improve the memory usage we inserted a finish block around the encryption step; the result was KJ (VC) matching the performance of KJ (SS).

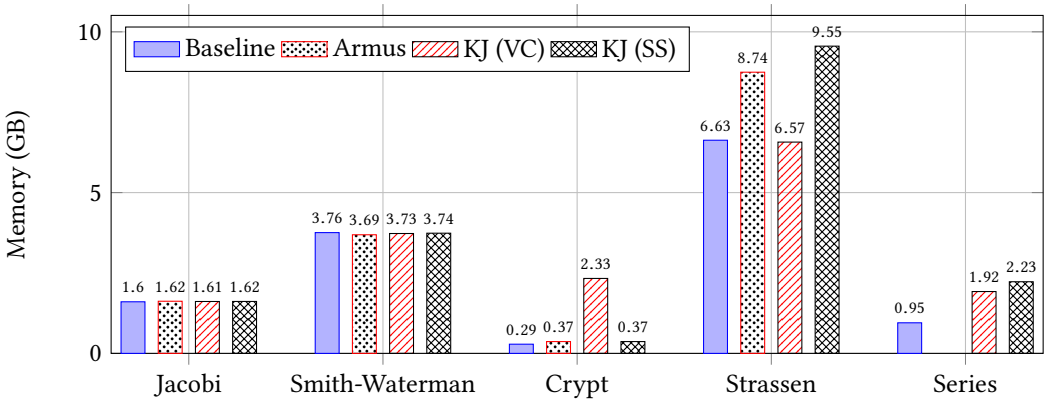

Fig. 7. Comparison of memory overhead for deadlock avoidance.

7 RELATED WORK

The deadlock avoidance problem. The problem of deadlock avoidance is a very well studied problem that dates as far back as 1960's, with Dijkstra [1965] Banker Algorithm. Minoura [1982] and Reveliotis et al. [1997b] cover the problem complexity in deadlock avoidance with complex synchronization patterns. The focus of our paper is on deadlock avoidance specialized for awaiting task termination.

In this paper we discuss two alternatives to mitigate deadlocks on awaiting task termination, one is ensuring the absence of data races (theorem 4.6), the is other by enforcing KJ. Welc et al. [2005] tackle the former problem by ensuring that the concurrent execution of a program with futures follows a valid serial execution, which implies the absence of data races. Similarly, Navabi et al. [2008] present an inter-procedural dataflow analysis for programs with futures that preserve sequential semantics with the help of a lightweight runtime. These two approaches yield stronger correctness guarantees by limiting the set of possible schedules.

Cogumbreiro et al. [2015] introduced a deadlock detection/avoidance algorithm for multiple synchronization mechanisms, including futures, by performing cycle detection on wait-for dependencies. Deadlock detection can be performed fully asynchronously, so the runtime overhead is usually negligible. Deadlock avoidance, however, must perform cycle detection every time a task blocks, which can cause severe slowdowns, as demonstrated in section 4.3.

Transitive closure. Instead of testing whether the wait-for dependencies are cyclic, we can test if a given blocked task can reach itself through the wait-for dependencies. The reachability problem can be solved by maintaining the transitive closure of the reachability relation, but the theoretical bounds are worst when compared to cycle detection. Computing the transitive closure from scratch can be solved with matrix multiplication [Munro 1971]; the best known algorithm solves this problem in $O(n^{2.376})$ [Coppersmith and Winograd 1990]. Alternatively, the transitive closure can be maintained dynamically [Demetrescu and Italiano 2005], but updating the graph takes $O(n^2)$ time. Furthermore, maintaining the transitive closure usually assumes a fixed set of vertices throughout the execution, and the problem is compounded since updates and tests run concurrently.

Data-race detection. There are three general approaches for detecting concurrency defects: *static analysis*, *dynamic analysis*, and a combination of the two. By admitting some restrictions on the

programming model, static analysis of certain classes of errors, such as deadlocks and atomicity violations, becomes more tractable [Boyapati et al. 2002b; Engler and Ashcraft 2003].

Dynamic analysis approaches identify races in an execution for a particular input. Causality-based methods [Banerjee et al. 2006; Flanagan and Freund 2009; Ha and Jun 2015] only identify data races that actually happened, and are usually implemented with vector clocks. Causality-based approaches suffer from a scalability problem, as the causal history can grow unbounded. For instance, in vector-clock-based techniques the logical time grows as a task produces more events, and the vector grows whenever a task forks another. Almeida et al. [2008] propose interval tree clocks as an alternative to vector clocks that can dynamically compact the causal history by reusing the same "task identifier" for multiple tasks. Techniques, such as using chain decomposition [Raychev et al. 2013], can also alleviate the history growth. Alternatively, summary-based methods [Savage et al. 1997] sacrifice preciseness to improve performance. For parallel programs that do not manipulate locks and restrict how tasks can join with other tasks, e.g., only parent-child joins, a single execution is sufficient to identify if any data race exist in any execution [Feng and Leiserson 1997; Mellor-Crummey 1991; Raman et al. 2012; Surendran and Sarkar 2016]. Surendran and Sarkar [2016] uses disjoint-sets to encode causality. Such data structure offers good space and time complexities, but assumes a global state. Maintaining a global state is impractical in concurrent dynamic analysis, as is the case of KJ.

8 CONCLUSIONS AND FUTURE WORK

In this paper we introduce a theory of futures and shared memory programming, and show that race conditions are the sole root cause for deadlocks. To the best of our knowledge our work is the first to build a mathematical argument on the root cause of deadlocks in this context. A novelty of our approach is on using causality to avoid deadlocks. The deadlock avoidance algorithm we introduce runs in $O(n)$ time and space, where n is the number of tasks forked. This algorithm is implemented in the Habanero-Java runtime as a tool called Gorn and exhibits a slowdown of around 1.06× for a program with 1 million tasks on a 16-core machine. Gorn verified a dataset of ~2,300 student homework solutions and found one deadlocked program. Furthermore, the evaluation of Gorn on student assignments shows the potential for such tools in parallel programming courses, including online courses that rely on autograding.

Opportunities for future work include both theoretical and algorithmic developments. We are interested in applying programming analysis methods that would allow us to tackle other results, such as determinism and serial equivalence. This paper introduces a novel approach of handling deadlock avoidance with causal history. It would also be interesting to try and extend this approach to other synchronization mechanisms. Promises seem a natural candidate, but pose a serious difficulty to dynamic verification, as it is not possible to know at runtime which task must fulfill the promise and therefore derive runtime dependencies as in this work.

ACKNOWLEDGMENTS

We would like to thank Alcides Fonseca and anonymous reviewers for valuable comments. This work was supported in part by FCT through the LASIGE Research Unit, ref. UID/CEC/00408/2013 and by Luso-American Development Foundation (Martins and Vasconcelos).

REFERENCES

Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. 2008. Interval Tree Clocks. In *OPODIS (LNCS)*, Vol. 5401. Springer, Article 18, 16 pages. https://doi.org/10.1007/978-3-540-92221-6_18

Utpal Banerjee, Brian Bliss, Zhiqiang Ma, and Paul Petersen. 2006. A theory of data race detection. In *PADTAD*. ACM, 69–78. https://doi.org/10.1145/1147403.1147416

Carlos Baquero and Nuno Preguiça. 2016. Why Logical Clocks Are Easy. *Commun. ACM* 59, 4 (2016), 43–47. https://doi.org/10.1145/2890782

Kenneth P. Birman and Thomas A. Joseph. 1987. Reliable Communication in the Presence of Failures. *Theoretical Computer Science* 5, 1 (1987), 47–76. https://doi.org/10.1145/7351.7478

Robert D. Blumofe and Charles E. Leiserson. 1998. Space-Efficient Scheduling of Multithreaded Computations. *Computing* 27, 1 (1998), 202–229. https://doi.org/10.1137/S0097539793259471

Gérard Boudol. 2009. A Deadlock-Free Semantics for Shared Memory Concurrency. In *ICTAC (LNCS)*, Vol. 5684. Springer, 140–154. https://doi.org/10.1007/978-3-642-03466-4_9

Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. 2002a. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA*. ACM, 211–230. https://doi.org/10.1145/582419.582440

Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. 2002b. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*. ACM, 211–230. https://doi.org/10.1145/582419.582440

Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-Java: The New Adventures of Old X10. In *PPPJ*. ACM, 51–61. https://doi.org/10.1145/2093157.2093165

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA*. ACM, 519–538. https://doi.org/10.1145/1094811.1094852

Tiago Cogumbreiro, Raymond Hu, Francisco Martins, and Nobuko Yoshida. 2015. Dynamic Deadlock Verification for General Barrier Synchronisation. In *PPoPP*. ACM, 150–160. https://doi.org/10.1145/2688500.2688519

Don Coppersmith and Shmuel Winograd. 1990. Matrix Multiplication via Arithmetic Progressions. *Symbolic Computation* 9, 3 (1990), 251–280. https://doi.org/10.1016/S0747-7171(08)80013-2

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). MIT.

Camil Demetrescu and Giuseppe F. Italiano. 2005. Trade-offs for Fully Dynamic Transitive Closure on DAGs: Breaking Through the $O(n^2)$ Barrier. *JACM* 52, 2 (2005), 147–156. https://doi.org/10.1145/1059513.1059514

Edsger W. Dijkstra. 1965. *Cooperating Sequential Processes.* Technical Report. Technical University of Eindhoven. EWD-123.

Anne Dinning and Edith Schonberg. 1991. Detecting Access Anomalies in Programs with Critical Sections. In *PADD*. ACM, 85–96. https://doi.org/10.1145/122759.122767

Dawson Engler and Ken Ashcraft. 2003. RacerX: effective, static detection of race conditions and deadlocks. In *SOSP*. ACM, 237–252. https://doi.org/10.1145/945445.945468

Mingdong Feng and Charles E. Leiserson. 1997. Efficient detection of determinacy races in Cilk programs. In *SPAA*. ACM, 1–11. https://doi.org/10.1145/258492.258493

Colin J. Fidge. 1988. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. In *ACSC*, Vol. 10. University of Queensland, 55–66.

Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: efficient and precise dynamic race detection. In *PLDI*. ACM, 121–133. https://doi.org/10.1145/1542476.1542490

Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *OOPSLA*. ACM, 57–76. https://doi.org/10.1145/1297027.1297033

Prodromos Gerakios, Nikolaos Papaspyrou, Konstantinos Sagonas, and Panagiotis Vekris. 2011. Dynamic Deadlock Avoidance in Systems Code Using Statically Inferred Effects. In *PLOS*. ACM, Article 5, 5 pages. https://doi.org/10.1145/2039239.2039247

Ok-Kyoon Ha and Yong-Kee Jun. 2015. An Efficient Algorithm for On-the-fly Data Race Detection Using an Epoch-based Technique. *Scientific Programming* 2015 (2015), 14. https://doi.org/10.1155/2015/205827

Robert H. Halstead, Jr. 1985. MULTILISP: A Language for Concurrent Symbolic Computation. *Transactions on Programming Languages and Systems* 7, 4 (1985), 501–538. https://doi.org/10.1145/4472.4478

Tobias Hilbrich, Bronis R. de Supinski, Martin Schulz, and Matthias S. Müller. 2009. A graph based approach for MPI deadlock detection. In *ICS*. ACM, 296–305. https://doi.org/10.1145/1542275.1542319

Tobias Hilbrich, Joachim Protze, Martin Schulz, Bronis R. de Supinski, and Matthias S. Müller. 2012. MPI runtime error detection with MUST: advances in deadlock detection. In *SC*. IEEE, Article 30, 11 pages.

Shams Imam and Vivek Sarkar. 2014. Habanero-Java Library: A Java 8 Framework for Multicore Programming. In *PPPJ*. ACM, 75–86. https://doi.org/10.1145/2647508.2647514

Shams Imam, Jisheng Zhao, and Vivek Sarkar. 2015. A Composable Deadlock-Free Approach to Object-Based Isolation. In *Euro-Par (LNCS)*, Vol. 9233. Springer, 426–437. https://doi.org/10.1007/978-3-662-48096-0_33

Sreekaanth S. Isloor and T. Anthony Marsland. 1980. The Deadlock Problem: An Overview. *Computer* 13, 9 (1980), 58–78. https://doi.org/10.1109/MC.1980.1653786

Edgar Knapp. 1987. Deadlock detection in distributed databases. *Computing Survey* 19, 4 (1987), 303–328. https://doi.org/10.1145/45075.46163

Bettina Krammer, Tobias Hilbrich, Valentin Himmler, Blasius Czink, Kiril Dichev, and Matthias S. Müller. 2008. MPI Correctness Checking with Marmot. In *PTW*. Springer, 61–78. https://doi.org/10.1007/978-3-540-68564-7_5

Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565. https://doi.org/10.1145/359545.359563

Peng Li, Kunal Agrawal, Jeremy Buhler, and Roger D. Chamberlain. 2010. Deadlock Avoidance for Streaming Computations with Filtering. In *SPAA*. ACM, 243–252. https://doi.org/10.1145/1810479.1810526

Glenn R. Luecke, Hua Chen, James Coyle, Jim Hoekstra, Marina Kraeva, and Yan Zou. 2003. MPI-CHECK: a tool for checking Fortran 90 MPI programs. *CCPE* 15, 2 (2003), 93–100. https://doi.org/10.1002/cpe.705

Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java Memory Model. In *POPL*. ACM, 378–391. https://doi.org/10.1145/1040305.1040336

Friedemann Mattern. 1989. Virtual time and global states in distributed systems. In *WDAG*. North-Holland/Elsevier, 215–226.

John Mellor-Crummey. 1991. On-the-fly detection of data races for programs with nested fork-join parallelism. In *SC*. ACM, 24–33. https://doi.org/10.1145/125826.125861

Toshimi Minoura. 1982. Deadlock Avoidance Revisited. *JACM* 29, 4 (1982), 1023–1048. https://doi.org/10.1145/322344.322351

Andrey Mokhov and Victor Khomenko. 2014. Algebra of Parameterised Graphs. *Transactions on Embedded Computing Systems* 13, 4s, Article 143 (2014), 22 pages. https://doi.org/10.1145/2627351

Ian Munro. 1971. Efficient Determination of the Transitive Closure of a Directed Graph. *Inform. Process. Lett.* 1, 2 (1971), 56–58. https://doi.org/10.1016/0020-0190(71)90006-8

V. Krishna Nandivada, Jun Shirako, Jisheng Zhao, and Vivek Sarkar. 2013. A Transformation Framework for Optimizing Task-Parallel Programs. *Transactions on Programming Languages and Systems* 35, 1, Article 3 (2013), 48 pages. https://doi.org/10.1145/2450136.2450138

Armand Navabi, Xiangyu Zhang, and Suresh Jagannathan. 2008. Quasi-static Scheduling for Safe Futures. In *PPoPP*. ACM, 23–32. https://doi.org/10.1145/1345206.1345212

Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2012. Efficient data race detection for async-finish parallelism. *Formal Methods in System Design* 41, 3 (2012), 321–347. https://doi.org/10.1007/s10703-012-0143-7

Veselin Raychev, Martin Vechev, and Manu Sridharan. 2013. Effective Race Detection for Event-driven Programs. In *OOPSLA*. ACM, 151–166. https://doi.org/10.1145/2509136.2509538

Spiridon A. Reveliotis, Mark A. Lawley, and Placid M. Ferreira. 1997a. Polynomial-complexity deadlock avoidance policies for sequential resource allocation systems. *Transactions on Automatic Control* 42, 10 (1997), 1344–1357. https://doi.org/10.1109/9.633824

Spiridon A. Reveliotis, Mark A. Lawley, and Placid M. Ferreira. 1997b. Polynomial-complexity deadlock avoidance policies for sequential resource allocation systems. *Transactions on Automatic Control* 42, 10 (1997), 1344–1357. https://doi.org/10.1109/9.633824

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *Transactions on Computer Systems* 15, 4 (1997), 391–411. https://doi.org/10.1145/265924.265927

Reinhard Schwarz and Friedemann Mattern. 1994. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. *Distributed Computing* 7, 3 (1994), 149–174. https://doi.org/10.1007/BF02277859

Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. 2008. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *ICS*. ACM, 277–288. https://doi.org/10.1145/1375527.1375568

Lorna A. Smith, J. Mark Bull, and Jan Obdrzálek. 2001. A Parallel Java Grande Benchmark Suite. In *SC*. ACM, 10. https://doi.org/10.1145/582034.582042

Rishi Surendran and Vivek Sarkar. 2016. Dynamic Determinacy Race Detection for Task Parallelism with Futures. In *RV (LNCS)*, Vol. 10012. Springer, 368–385. https://doi.org/10.1007/978-3-319-46982-9_23

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *CASCON*. IBM, Article 13, 160–170 pages.

Philippe Virouleau, Pierrick Brunet, François Broquedis, Nathalie Furmento, Samuel Thibault, Olivier Aumage, and Thierry Gautier. 2014. Evaluation of OpenMP Dependent Tasks with the KASTORS Benchmark Suite. In *IWOMP (LNCS)*, Vol. 8766. Springer, 16–29. https://doi.org/10.1007/978-3-319-11454-5_2

Anh Vo. 2011. *Scalable Formal Dynamic Verification of MPI Programs Through Distributed Causality Tracking*. Ph.D. Dissertation. University of Utah. Advisor(s) Gopalakrishnan, Ganesh. AAI3454168.

Adam Welc, Suresh Jagannathan, and Antony Hosking. 2005. Safe Futures for Java. In *OOPSLA*. ACM, 439–453. https://doi.org/10.1145/1094811.1094845

Amy Williams, William Thies, and Michael D. Ernst. 2005. Static Deadlock Detection for Java Libraries. In *ECOOP (LNCS)*, Vol. 3586. Springer, 602–629. https://doi.org/10.1007/11531142_26