Formalization of Habanero Phasers using Coq[☆]

Tiago Cogumbreiro, Jun Shirako, Vivek Sarkar

Rice University, 6100 Main St, Houston, TX 77005, USA

Abstract

Phasers pose an interesting synchronization mechanism that generalizes many collective synchronization patterns seen in parallel programming languages, including barriers, clocks, and point-to-point synchronization using latches or semaphores. This work characterizes scheduling constraints on phaser operations, by relating the execution state of two tasks that operate on the same phaser. We propose a formalization of Habanero phasers, May-Happen-In-Parallel, and Happens-Before relations for phaser operations, and show that these relations conform with the semantics. Our formalization and proofs are fully mechanized using the Coq proof assistant, and are available online.

Keywords: phasers, barriers, Coq, synchronization, formalization

1. Introduction

Phasers are an interesting synchronization mechanism that generalizes barriers with collective producer-consumer synchronization. A phaser can encode the synchronization mechanism of latches, futures, join barriers, cyclic barriers, as well as any *collective synchronization pattern* provided by CUDA, C♯, Java, MPI, and X10. Phasers [1] were first introduced in the Habanero Extreme Scale research project at Rice University, as an extension to X10 clocks [2], and implemented in Habanero-Java and Habanero-C. A restricted form of phasers was also introduced in the standard `java.util.concurrent.Phaser` library starting with Java 7. The phaser synchronization mechanism is relevant at the theoretical level because of its generality. Theoretical results that target phasers can easily translate across different languages and parallel runtimes [3].

This paper introduces the first formalization of: (1) the Habanero phaser semantics, and (2) the *Phase Ordering* relation proposed in [1], which is a relation used for causality analysis [4; 5]. The former is a crucial stepping stone for formal developments focused in many-to-many synchronization. The latter gives us an interpretation of a phaser as a logical clock. Causality analysis are fundamental in the verification of barrier synchronization errors [6], lock-based deadlock prediction [7; 8], and race-detection [9; 10].

Crafa *et al.* propose a Coq formalization of a subset of the X10 and define a causality relation in [11], but only consider fork-join synchronization, and omit dynamic barrier synchronization that we formalize. Feautrier uses an informal definition of the causality relation over clock operations [1] to optimize X10 programs by reducing synchronization [12]. Tomofumi *et al.* use a causality relation to check for data races in the polyhedral subset of clocked X10 programs [13]. Joshi *et al.* propose an informal causality relation for X10 clocks [14].

This paper establishes two main properties with respect to the phasers semantics we introduce. First, as required by causality analysis, we show that the Phase Ordering relation we define is a causality relation [4]. Second, since the Phase Ordering relation is defined on the state of a phaser P, we show that the Phase Ordering relation *conforms* with the execution semantics of phasers; that is, if a state P reduces to Q after zero or more steps, then Q cannot happen before P. Such technique of showing the correctness of a causality relation was first introduced in [15]. By

[☆]Published in Journal of Logical and Algebraic Methods in Programming, 14 March 2017. DOI:10.1016/j.jlamp.2017.02.006
Email addresses: cogumbreiro@rice.edu (Tiago Cogumbreiro), shirako@rice.edu (Jun Shirako), vsarkar@rice.edu (Vivek Sarkar)
[1]Phasers are an extension of clocks with registration modes.

targeting phasers, our formalization unifies collective producer-consumer synchronization [6] and barriers with dynamic membership [14] in a single theoretical framework. Additionally, we formalize and establish the correctness of our definitions with proofs verified by the Coq proof assistant, available online, as part of our **HJ-Coq formalization project**.[2]

The main contributions of this paper are:

1. the first formalization semantics of Habanero phasers and the Phase Ordering relation;
2. showing that the Phase Ordering is a causality relation (Theorem 1) that conforms with the reduction relation (Theorem 3);
3. the full Coq mechanization of the theory, along with examples.

This paper extends previous work [16] with the following two contributions:

§6 A discussion of our Coq formalization and decidability results for all definitions.

§7 A web-based interpreter for the theory in this paper, implemented with code translated from Coq.

In the next section, we overview the background of the three main topics covered in this paper: phasers, causality relations, and phase ordering. Next, in Section 3, we formally describe the phaser operations and its semantics. In Section 4, we introduce Phase Ordering, the MHP relation, and the HB relation. Next, in Section 5, we establish the main results with regards to the semantics of phasers. Section 6 discusses the design of our Coq formalization and establishes decidability results. Section 7 presents our executable semantics as a web application. We conclude in Section 8 and discuss future directions.

2. Background

Habanero Phasers. The phaser synchronization mechanism lets tasks observe a collective event, called *phase*, which is visible once every member of a group of tasks *signals* the phaser exactly once. We define *signalers* of the phaser as the group of tasks able to signal a phaser. The same phaser can be used to observe multiple phases, which are distinguishable by a natural number. A task can observe phase n once each signaler issues at least n signals. Phaser synchronization also features *dynamic membership*, that is, the group of signalers can grow and shrink dynamically: a signaler can add a member, which in turn inherits the signal count of the task adding it; a signaler can also revoke its membership at any time.

As an example of phaser synchronization, let us consider a group of three tasks, uniquely identified by x_1, x_2, and x_3, and let this group of tasks be the signalers of phaser P. Also, lets examine a point in time, with respect to phaser P, where task x_1 signaled 3 times, task x_2 signaled 4 times, and task x_3 signaled 10 times. Tasks can use P to observe any phase below or equal to phase 3, since the signalers collectively issued at least 3 signals. Conversely, at this point in time, any phase above 3 is *not* observable, *e.g.,* for phase 4 to be observed we are missing a signal from task x_1. Dynamic membership affects synchronization: if task x_1 adds a task x_4 as a signaler of phaser P, then for phase 4 to be observable we are missing a signal from task x_1 and a signal from task x_4; and, if, subsequently, tasks x_1 and x_4 revoke their membership, then phase 4 is observable.

Figure 1 depicts scheduling constrains imposed by sequential execution and phaser synchronization, where three tasks, x, y, and z, synchronize with the same phaser. In the context of this paper, scheduling constraints corresponds to causality. We annotate each node with the operation it is executing. Node 6 represents the event before writing to location l. Node 7 represents the event after writing to location l. Edge $(7, 2)$ represents that for the execution of the wait to happen, task x must observe the drop of task y. As for the happens-before-relation, since there is a path from node 7 to node 3, then the write to location l by task y at node 7 must happens before the read from the same location by task x at node 3. Event 12 and event 3 may happen in parallel, which means that task x and z read from the same location concurrently. By using the happens-before relation, we can conclude the execution develops into three stages: firstly, task y writes a value to location l; secondly, tasks x and z read from l and task z writes to m; thirdly, task x reads from m.

[2] `http://cogumbreiro.github.io/jlamp17/`

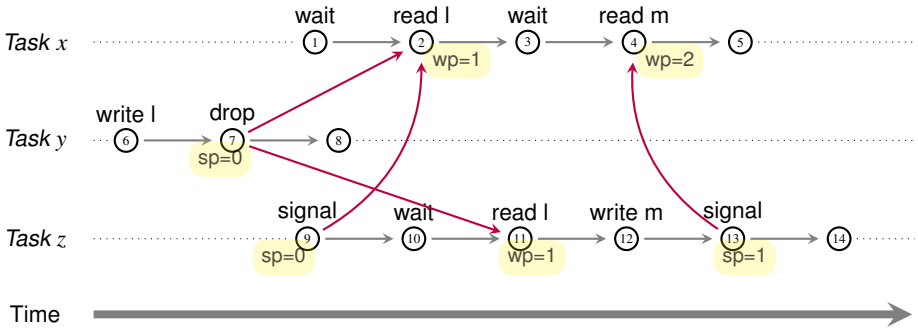


Figure 1: Event causality in a program with three tasks and one phaser.

Causality relations. How we represent the causality relation is paramount for its analysis. The field of distributed systems includes a vast body of work on representing causality with logical clocks with the objective of reducing space and time of causality-tracking. The idea behind formulating the causality-relation with logical clocks hinges on two concepts. First, rather than recording the set of events produced, each task counts the events it produced. Second, each task perceives the whole system by keeping track of the logical time of all tasks, this is also known as a *time stamp* of the system. With logical clocks the problem of checking if an event (time stamp) happens before another amounts to comparing counters (a point-wise operation on a vector) — which sits in contrast checking the reachability between two nodes, like in Figure 1. There are various formulations of logical clocks, each tailored to the characteristics of the concurrent system being modeled [17].

Let us revisit Figure 1 and direct our attention to phaser synchronization edges; notice how the nodes of these edges are annotated with wait and signal phases and how the signal phase of the source node always precedes the wait phase of the target node. For instance, consider edge $(7, 2)$, node 7 has a signal-phase of 0, which is smaller than wait-phase 1 of node 2. An important characteristic of Habanero phasers is that its semantics are defined according to the wait-phase and the signal-phase of each participant; these are task-local properties available to the programmer. At run-time, programs with Habanero phasers can use the task view of the phaser to reason about causality of phaser events.

Phase Ordering. This paper formalizes *Phase Ordering* [1], that defines a time stamp as two counters: how many times the task signaled a given phaser, and how many times the task waited on that phaser. The former is called the signal-phase, or `sp`, and the latter is called the wait-phase, or `wp`. We can observe that in Figure 1 that whenever there is a node such that the `sp` value is smaller than the `wp` value, then there is a path from the former node to the latter, *e.g.,* from node 7 to node 11.

3. Phaser semantics

Let us discuss the semantics of Habanero phasers by presenting Listing 1. This listing exhibits the causality relation of Figure 1, so we annotate each line with the nodes of the causality relation and overlay the synchronization edges. In the comments, we relate each API call with its respective phaser operation as defined in the formalization. As we have seen, from analyzing Figure 1, our program develops into three stages. First, task *y* writes to location `x`. Second, tasks *x* and *z* both read from location `x`, and task *x* writes to location `y`. Third, task *z* reads from location `y`. The listing makes explicit that: task *y* creates a phaser object `ph` to synchronize the memory accesses, at Line 1; task *y* spawns tasks *x* and task *z*, at Lines 2 and 8, respectively; and, task *y* registers both spawned tasks as members of phaser `ph` upon spawning.

Membership. Tasks can be registered with a phaser by creating a phaser with `newPhaser` and by being spawned with `asyncPhased`. In Line 1, task *y* is the only member of phaser `ph`. In Line 8, the members of the phaser are tasks *x*, *y*, and *z*. A task can only manipulate a phaser if it is registered with that phaser. At any time a task can revoke its membership by invoking phaser method `drop`, as in Line 15. This means that after dropping the phaser, task *y* cannot

Listing 1: HJ-Lib code listing three tasks that synchronize with a phaser.

```
1  ph = newPhaser();
2  asyncPhased(ph.inMode(WAIT_ONLY), () -> { // b:reg a WO
3    ph.doWait();                            // a:wait
4    println(x);
5    ph.doWait();                            // a:wait
6    println(y);
7  });                                       // a:drop
8  asyncPhased(ph.inMode(SIG_WAIT), () -> {  // b:reg c SW
9    ph.signal();                            // c:signal
10   ph.doWait();                            // c:wait
11   y = compute(x);
12   ph.signal();                            // c:signal
13 });                                       // c:drop
14 x = 1;
15 ph.drop();                                // c:drop
```

signal, wait, nor registered tasks on phaser `ph`. Before a phased task terminates, it implicitly drops from all phasers it is registered with. For instance, in Lines 7 and 13, tasks x and z invoke `ph.drop()` internally.

Registration modes. A feature that distinguishes Habanero phasers from other barrier-like synchronization mechanisms is that *signaling and waiting is optional*: declare which members do not influence synchronization, and which members need not wait for other members. A task can choose to manipulate a phaser according to two abilities: (i) the ability to observe phaser synchronization, *i.e.,* waiter, and (ii) the ability to influence synchronization, *i.e.,* signaler. Each member is registered according to a mode r among: `SIG_WAIT`, or `SW`, for tasks that must signal and wait, `WAIT_ONLY`, or `WO`, for tasks that wait but do not signal, and `SIG_ONLY`, or `SO`, for tasks that signal but do not wait. Function `newPhaser()` registers the calling task as `SIG_WAIT` mode, so that the task can then register with fewer capabilities. When spawning phased tasks, the programmer can set the mode of phaser `ph.inMode(r)`, such as in Lines 2 and 8. When registering a task the mode cannot have more capabilities than the calling task. For instance, a task registered with `WAIT_ONLY` cannot register a task in `SIG_WAIT`, nor on `SIG_ONLY` modes.

Waiting observes the signals from *every* signaler. Thus, tasks that wait and signal, mode `SW`, as in the example, must signal before waiting at every phase to prevent waiting for a signal the task did not produce. This synchronization pattern is known as a *barrier*. Members disregard wait-only (`WO`) tasks upon waiting; this subset of tasks cannot influence synchronization, only observe it. Phasers can encode latches, future-promises, and fork-join synchronization patterns using wait-only tasks. Finally, tasks that only signal, do not wait for others; this lets phasers encode producer-consumer synchronization.

In summary, the HJ-lib API for phasers is:

Membership Function `newPhaser()` creates a new phaser. The calling task becomes the sole member of the phaser. Function `asyncPhased(ph_1,...,ph_n, l)` spawns a task that is registered with phasers `ph_1`, ..., `ph_n`; the new task executes lambda expression `l`. Phaser method `drop()` revokes the membership of the issuing task.

Registration modes Upon registration the programmer can specify how a task operates the phaser, picking a mode from: `SIG_WAIT`, `SIG_ONLY`, `WAIT_ONLY`. Mode `SIG_WAIT` can issue waits and signals. Mode `SIG_ONLY` only signals (does not wait). Mode `WAIT_ONLY` only waits (does not signal). Upon registering a task with a phaser, the task being registered cannot have more capabilities than the calling task.

Signal The *non-blocking* phaser method `signal` increments the signal phase of the calling task. Any member task may thus independently advance up to an arbitrary phase.

Wait The phaser method `doWait` blocks the calling task until every signaler issued at least `wp` + 1 signals.

HJ Phaser formalization. We define the state of a phaser P to be a map from members $\mathcal{T}$ into *views* $\mathcal{V}$, which holds the signal count, the wait count, and registration mode of a member. Consider the usual operations on finite maps (which we use to encode phasers) with the given notation: predicate $P(x) = v$ ensures that the pair of key x and value v

$$\frac{P(x) = v \qquad \text{Signaler } v \qquad v.\texttt{mode} = \texttt{SW} \implies v.\texttt{wp} = v.\texttt{sp}}{P \xrightarrow{x:\texttt{signal}()} P[x \mapsto v.\texttt{sp} := v.\texttt{sp} + 1]}$$

$$\frac{P(x) = v \qquad \text{Phase}(P, v.\texttt{wp} + 1) \qquad \text{Waiter } v \qquad v.\texttt{mode} = \texttt{SW} \implies v.\texttt{wp} + 1 = v.\texttt{sp}}{P \xrightarrow{x:\texttt{wait}()} P[x \mapsto v.\texttt{wp} := v.\texttt{wp} + 1]}$$

$$\frac{y \notin P \qquad P(x) = v \qquad \text{Waiter } r \implies \text{Waiter } v \qquad \text{Signaler } r \implies \text{Signaler } v}{P \xrightarrow{x:\texttt{reg}(y,r)} P[y \mapsto v.\texttt{mode} := r]}$$

$$\frac{x \in P}{P \xrightarrow{x:\texttt{drop}()} P - x}$$

Figure 2: Operational semantics of phaser operations

is a member of map P, predicate $x \in P$ is short-hand for $\exists v\colon\ P(x) = v$, map $P[x \mapsto v]$ adds the pair x and v to map P (replacing the assigned view if $x \in P$), and map $P - x$ results from removing the pair associated with key x from map P. In the mechanization, we use Coq's standard library of finite maps `Coq.FSets.FMaps`.

A view v represents the task-local information that each member has over the phaser. The view consists of a triple: the first value n is a natural number that counts the number of times the given task issued a signal on the target phaser and can be accessed by $v.\texttt{sp}$; the second value m counts the number of waits and can be accessed by $v.\texttt{wp}$; the third value r is the registration mode of the given task and is accessed by $v.\texttt{mode}$.

$$v \ ::= \ \{\texttt{sp} := n, \texttt{wp} := m, \texttt{mode} := r\}$$

The field update operation $v.f := e$ yields a view that is the same as v except for field f that becomes e. For instance, $v.\texttt{sp} := 3$ yields a view, say w, where $w.\texttt{wp} = v.\texttt{wp}$, $w.\texttt{sp} = 3$, and $w.\texttt{mode} = v.\texttt{mode}$. To inquire the signaling and waiting abilities of a view we have the following predicates: Waiter $r \stackrel{\text{def}}{=} r \in \{\texttt{WO}, \texttt{SW}\}$, Signaler $r \stackrel{\text{def}}{=} r \in \{\texttt{SO}, \texttt{SW}\}$. And let the short-hand notation Signaler $v \stackrel{\text{def}}{=}$ Signaler $v.\texttt{mode}$ and Waiter $v \stackrel{\text{def}}{=}$ Waiter $v.\texttt{mode}$.

We define a small-step operational semantics for phaser operations in Figure 2. The reduction $\xrightarrow{x:o}$ is labeled by the member x issuing the operation, and by an operation o defined below.

$$o \ ::= \ \texttt{signal}() \mid \texttt{wait}() \mid \texttt{reg}(x, r) \mid \texttt{drop}()$$

Remark 1. *To model Java phasers and X10 clocks semantics refer to Figure 2 but limit the registration mode to signal-wait mode, that is* $r \ ::= \ \texttt{SW}$.

Operation `signal()` increments the signal phase. Only tasks registered as signalers can issue this operation. The pre-conditions in signal, $v.\texttt{wp} = v.\texttt{sp}$, and in wait, $v.\texttt{wp} + 1 = v.\texttt{sp}$, enforce tasks registered in signal-wait mode to interleave each signal with a wait.

Waiting is the crux of synchronization. A task x with view $P(x) = v$ awaits the subsequent wait phase, that is, $v.\texttt{wp} + 1$. Proposition Phase(P, n) holds once phase n can be observed; the definition ensures that all tasks that can signal have issued at least n signals.

$$\text{Phase}(P, n) \stackrel{\text{def}}{=} \forall x\colon\ \text{Signaler } P(x) \implies P(x).\texttt{sp} \geq n$$

Tasks register other tasks with with `asyncPhased`, which is captured by $\texttt{reg}(x, r)$. For instance, in Line 1 of Listing 1, instruction `asyncPhased(ph.inMode(SIG_WAIT),...)` becomes $\texttt{reg}(x_2, \texttt{SW})$ in this semantics if we are spawning task x_2. Habanero phasers limit task registration: only unregistered tasks can be added, thus $y \in P$, only registered

tasks x can add new members, $P(x) = v$, only waiters can register other waiters, hence Waiter $r \implies$ Waiter v, and only signalers can register other signalers, so Signaler $r \implies$ Signaler v.

To establish the results in the next section, let us establish the invariant of well-formedness. Additionally, let $P \to Q$ be defined as there exist x and o such that $P \xrightarrow{x:o} Q$.

Definition 1 (Well-formed view)**.** *Let a well-formed view be such that $v.\mathtt{wp} \leq v.\mathtt{sp}$ and if* Waiter *v then $v.\mathtt{sp} - v.\mathtt{wp} \leq 1$. Let $\mathcal{V}^{WF}$ be the set of all well-formed views.*

Lemma 1 (Reduction preserves well-formedness of views)**.** *Let P be such that if $P(x) = v$, then $v \in \mathcal{V}^{WF}$. If $P \to Q$, then Q is such that if $Q(x) = v$, then $v \in \mathcal{V}^{WF}$.*

Henceforth, we only consider views that are in $\mathcal{V}^{WF}$.

4. Phase Ordering

This section formalizes Phase Ordering, originally introduced in [1], to reason about whether two tasks should execute concurrently in terms of views $\mathcal{V}$ and states $\mathcal{P}$. Specifically, Phaser Ordering is a Happens-Before relation: if the number of signals issued by a task is smaller than the last phase observed by some other task, then the former Happened Before the latter. For instance, let v_1 be a view that task y has over the phaser in Figure 1 and v_2 be a view that task z has over the phaser in Figure 1, each from a distinct state of the same phaser `ph`, *i.e.,* there exists two states P and Q such that $P(y) = v_1$ and $Q(z) = v_2$. Now, let $v_1 \stackrel{\text{def}}{=} \{\mathtt{sp}:=0, \mathtt{wp}:=0, \mathtt{mode}:=\mathtt{SW}\}$ be the view of y when executing `write l` and $v_2 \stackrel{\text{def}}{=} \{\mathtt{sp} := 1, \mathtt{wp} := 1, \mathtt{mode} := \mathtt{SW}\}$ be the view z when executing `read l`. The registration mode tells us that view v_2 must observe and wait for the signals of v_1. View v_1 tells us that y did not produce any signal and v_2 tells us that z observed phase 1 (a collective signal). Thus, since y signaled fewer times than the phase observed by z, we can infer that v_1 must have happened before v_2. In order for view v_2 to observe a signal, the task controlling view v_1 must eventually signal to become $v_1.\mathtt{sp} = 1$.

Definition 2 (Happens-before (HB) relation)**.** *Let $v_1 \prec v_2$ read as v_1 must have happened before v_2, defined as the conjunction of:*

$$\text{Signaler } v_1 \qquad v_1.\mathtt{sp} < v_2.\mathtt{wp} \qquad \text{Waiter } v_2$$

We say that $P \prec Q$ if there exist two tasks x, y such that $P(x) \prec Q(y)$.

Example 1. *Suppose $P \xrightarrow{x:\mathtt{signal()}} Q \xrightarrow{x:\mathtt{wait()}} R$ and that $P(x).\mathtt{mode} = \mathtt{SW}$. We have that $P \prec R$.*

Proof. First, we simplify our goal, since we know that from `wait()` $R(x).\mathtt{wp} = Q(x).\mathtt{wp} + 1$ and because `signal()` does not alter the wait phase we have that $Q(x).\mathtt{wp} = P(x).\mathtt{wp}$. Hence, $P(x).\mathtt{sp} < R(x).\mathtt{wp} \equiv P(x).\mathtt{sp} < Q(x).\mathtt{wp} + 1 \equiv P(x).\mathtt{sp} < P(x).\mathtt{wp} + 1$. Now, by inverting reduction $\xrightarrow{x:\mathtt{signal()}}$, we get two cases: either $P(x).\mathtt{wp} = P(x).\mathtt{sp}$ and it trivially holds, otherwise we get a contradiction. □

Let us show that $\prec$ is a causality relation over views, a fundamental notion for many problems occurring in distributed computing [18]. By causality relation we mean a strict partial order: (i) *transitive:* if $v_1 \prec v_2$ and $v_2 \prec v_3$, then $v_1 \prec v_3$; (ii) *irreflexive:* for all v, we have that $v \not\prec v$; (iii) *asymmetric:* if $v_1 \prec v_2$, then $v_2 \not\prec v_1$.

Lemma 2. *$(\prec, \mathcal{V})$ is a causality relation.*

Finally, we define the usual notion of May-Happen-in-Parallel (or concurrency relation) for views and for phasers.

Definition 3 (May-Happen-in-Parallel relation)**.** *Let $v_1 \parallel v_2$ read as v_1 happens in parallel with v_2 and be defined as $v_1 \not\prec v_2$ and $v_2 \not\prec v_1$. Similarly, let $P_1 \parallel P_2$ be defined as $P_1 \not\prec P_2$ and $P_2 \not\prec P_1$.*

5. Results

Similarly to what happened with showing the causality of $\prec$ over views, when establishing the causality of $\prec$ over phasers we require an invariant that relates the various local views of a phaser. While in the context of views, we must ensure that the wait phase does not overtake the signal phase, in the context of phasers we must ensure that its views may happen in parallel with each other, *i.e.,* there must be no scheduling constraints within a phaser.

Definition 4 (Well-ordered phaser)**.** *Let a* well-ordered phaser *be such that* $P \| P$*. Let* $\mathcal{P}^{WO}$ *be the set of all well-ordered phasers.*

Reduction *cannot* introduce unsolvable scheduling constraints within a phaser.

Lemma 3 (Reduction preserves phaser well-orderedness)**.** *If* $P \in \mathcal{P}^{WO}$ *and* $P \to Q$*, then* $Q \in \mathcal{P}^{WO}$*.*

We are now ready to show that $\prec$ is a causality relation over phasers.

Theorem 1. $(\prec, \mathcal{P}^{WO})$ *is a causality relation.*

The execution of an HJ program must respect the scheduling restriction imposed by Phase Ordering. In the point of view of our formalization, the reduction relation captures the execution of a single phaser operation. Thus, Theorem 2 shows that the state after execution cannot happen before the state before execution, or, in other words, that the pre- and post-states of a phaser operation respect Phase Ordering.

Theorem 2. *If* $P \in \mathcal{P}^{WO}$ *and* $P \to Q$ *we have that* $Q \not\prec P$*.*

From $P \to Q$ we can also conclude $P \not\prec Q$. Thus, it follows that.

Lemma 4. *If* $P \in \mathcal{P}^{WO}$*,* $P \to Q$*, then* $P \parallel Q$*.*

Be aware, however, that MHP does *not* enjoy transitivity, otherwise HB would be an empty relation! Example 1 is an evidence of when $P \parallel Q$ and $Q \parallel R$ but $\neg(P \parallel R)$, as $P \prec R$. This also tells us that for any two states P and Q if we have $P \prec Q$, then state Q is the result of at least two phaser operations.

The final theorem establishes that the execution of an HJ program respects the scheduling restriction imposed by Phase Ordering. While Theorem 2 relates the pre- and post-states of executing a single operation, Theorem 3 generalizes this result to any possible execution trace, showing that Phase Ordering captures the execution order of instructions in programs that use phasers. Let $\to^*$ be defined as the reflexive transitive closure of $\to$.

Theorem 3 (Absence of synchronization errors)**.** $P \in \mathcal{P}^{WO}$*,* $P \to^* Q$*, then* $Q \not\prec P$*.*

Our results can be summarized into three groups. The first group consists of Lemmas 1 and 3; this serves as a steppingstone for our main results. The former lemma establishes an invariant on a relationship between wait and signal phases ($\mathcal{V}^{WF}$), while the latter lemma establishes an invariant on a relationship between any two views picked from a state ($\mathcal{P}^{WO}$). Since $\mathcal{V}^{WF}$ and $\mathcal{P}^{WO}$ are preserved by our semantics, any program that manipulates Habanero phasers can assume Lemmas 1 and 3 to hold. The second group consists of Lemma 2 and Theorem 1 and lets us relate views (and states) with the HB relation. The third group consists of Theorems 2 and 3 and it lets us conclude that the execution of a program using Habanero phasers respects the scheduling restriction imposed by Phase Ordering (HB) relation.

6. Coq Formalization

In this section, we introduce some details of our Coq formalization. We overview our definition of the operational semantics and establish decidability results for all definitions in the paper. Decidable properties can be automatically translated from Coq into JavaScript, which we use in our accompanying artifact.

6.1. Operational semantics

A phaser is defined as a finite map from task identifiers to task views; We use the finite map library (and properties) available from Coq's standard library `Coq.FSets.FMaps`.[3] The code below defines the operational semantics, where proposition `Reduces ph (x, o) ph'` represents $P \xrightarrow{x:o} P'$.

```
(* Auxiliary definitions: *)
Inductive op := SIGNAL | WAIT | DROP | REGISTER : registry -> op.
Record rule := {pre: tid -> phaser -> Prop; exec: tid -> phaser -> phaser}.
Definition select_rule o := (* ... *)
(* Operational semantics: *)
Inductive Reduces ph: (tid*op) -> phaser -> Prop :=
| ph_reduces:
  let r := select_rule o in
  ∀ x o, pre r x ph -> Reduces ph (x, o) (exec r x ph).
```

Using a more familiar notation, the above can be typeset as

$$\frac{r = \texttt{select\_rule}(o) \qquad \texttt{pre}(r, x, P)}{P \xrightarrow{x:o} \texttt{exec}(r, x, P)}$$

The reduction expects the rule `r` that is assigned to operation `o`, with expression `select_rule o`. Each rule `r` consists of a pre-condition and a rewrite function. The reduction itself is then defined as usual: if the pre-condition `pre r t ph` holds, then phaser `ph` reduces to expression `exec r t ph`.

Rule wait. The reduction rules of the semantics follow directly, each of which instantiates the type `rule`. Let us discuss the rule for operation wait, that consists of pre-condition `WaitPre` and rewrite function `wait`.

```
Definition rule_wait := {| pre := WaitPre; exec := wait |}.
```

The pre-condition of the wait rule (see Figure 2) is the conjunction of

$$P(x) = v \qquad \text{Phase}(P, v.\texttt{wp} + 1) \qquad \text{Waiter } v \qquad v.\texttt{mode} = \texttt{SW} \implies v.\texttt{wp} + 1 = v.\texttt{sp}$$

which can be defined as predicate `WaitPre t ph` listed below, where predicate `MapsTo t v ph` corresponds to $P(x) = v$.

```
Inductive WaitPre (t:tid) (ph:phaser) : Prop :=
| wait_pre_def:
  ∀ v, MapsTo t v ph -> Phase ph (wp v + 1) -> Waiter v -> (mode v = SW -> wp v + 1 = sp v) ->
  WaitPre t ph.
```

Finally, the rewrite function for wait, $P[x \mapsto v.\texttt{wp} := v.\texttt{wp} + 1]$, can be defined with the following code listing.

```
Definition wait (t:tid) (ph:phaser) : phaser :=
match find t ph with
| Some v ⇒ add t {|sp:=sp v; wp:=1 + wp v; mode:=mode v |} ph
| None ⇒ ph
end.
```

Expression `find t ph` is defined in the Coq's standard library of finite maps. A property of this function is that `find t ph = Some v` if, and only if, `MapsTo t v ph` holds. Thus, function wait increments the wait phase of task `t` if `MapsTo t v ph`, otherwise task `t` is not registered and function `wait` leaves phaser `ph` unaltered. Thus, whenever `WaitPre t ph` holds, then `wait t ph` increments the wait phase of task `t` in phaser `ph`.

[3] `https://coq.inria.fr/library/Coq.FSets.FMaps.html`

6.2. Decidability results

Background on `sumbools` *and* Σ*-types.* To reason about the decidability of a logical proposition we use Coq's `sumbool` library,[4] where type `{A}+{B}` represents the type of an expression that computes either a proof for `A`, or proof for `B`. For instance, type $\{P \prec P'\} + \{P \not\prec P'\}$ declares whether or not phaser P happened before P'. An expression that yields a `sumbool` is restricted in that it cannot manipulate proof objects—*e.g.,* cannot destruct a disjunction—while choosing which branch to prove from `A` or `B`. Furthermore, as a result of the extraction process, when translating a sumbool `{A}+{B}`, proof objects `A` and `B` are erased, so the translated type of a `sumbool` is isomorphic to the type of booleans.[5]

A Σ-type [19], also called dependent sum type, notation `{x:T | A}`, the expression yields an `x` of type `T` such that `A` holds, where variable `x` is bound in `A`. Code extraction erases proof object `A` from the type and retains the witness, so type `{x:T | A}` is extracted as type `T`. For instance, type $\{P' \mid P \xrightarrow{x:o} P'\}$ declares a phaser P' that results from a reduction step.

Operational semantics. The decidable version of the operational reduction benefits from the fact that the rewrite function of each rule (`exec` is decidable by definition as it yields a `phaser`; similarly, function `select_rule` is also decidable by definition as it returns a `rule`. Thus, proving the decidability of the operational semantics amounts to showing the decidability of the pre-condition of each rule. With the goal of decidability in mind, we revisit the definition of a rule to include the decidable pre-condition.

```
Record rule := mk_rule {
  pre: tid -> phaser -> Prop;
  pre_dec t ph : { pre t ph } + { ¬ pre t ph };
  exec: tid -> phaser -> phaser}.
```

Function `eval` captures the executable semantics of predicate $P \xrightarrow{x:o} P'$, by returning `Some` P' if, and only if, the predicate holds and `None` otherwise.

```
Definition eval (t:tid) (o:op) ph : option phaser :=
if pre_dec (select_rule o) t ph
then Some (exec (select_rule o) t ph)
else None.
```

Rule wait. The pre-condition of rule wait is the most interesting due to the difficulty of showing that predicate Phase(P, n) is decidable. Definition Phase$(P, n) \stackrel{\text{def}}{=} \forall x$: Signaler $P(x) \implies P(x).\text{sp} \geq n$ can be written in Coq as

```
Definition Phase ph n := ∀ t v, MapsTo t v ph -> Signaler (mode v) -> sp v ≥ n.
```

where term Signaler $P(x)$ becomes the implication $P(x) = v \implies$ Signaler v, which in Coq is formulated as `MapsTo t v ph -> Signaler (mode v)`.

Our decidability proof of `Phase ph n` is divided into two parts. First, we declare a function `signalers` that retrieves the signal phase of every task that is a signaler in phaser `ph`. Function `List.omap f l` is a higher-order function that applies `f` to each element `e` of `l` and constructs a list of the result of function `f e = Some y` in the same order. Function `List.omap` filters out any element `e` for which `f e = None`. Function `elements`, defined in `Coq.FSets.FMaps`, converts the phaser `ph` into a list of pairs of a task and its view.

```
Definition signalers ph : list (tid * nat) :=
let handle_entry (p:tid * taskview) :=
  let (t, v) := p in
  if signaler (mode v) then Some (t, sp v)
  else None
in
List.omap handle_entry (elements ph).
```

We specify function `signalers ph` with the following lemma.

[4] `https://coq.inria.fr/library/Coq.Bool.Sumbool.html`

[5] Since Coq lets us control the extraction process, in our project a sumbool is translated into an OCaml boolean.

Lemma 5 (Correctness)**.** *We have* $(x, n) \in$ `signalers` P *if, and only if,* $P(x).wp = n$.

The decidable function of predicate Phase(P, n) is function `phase` which computes `true` if, and only if, Phase(P, n) holds. Function `phase` returns true if, and only if, each signal phase m in `signalers ph` we have that $m \geq n$.

```
Definition phase ph n : boolean :=
let f (p : tid * nat) := if ge_dec (snd p) n then true else false in
List.forallb f (signalers ph).
```

Main results. There are two main results. First, that the operational semantics is decidable. Second, that the happens-before relation is decidable.

Lemma 6 (Correctness of the executable semantics)**.** *We have* `eval` (x, o) $P =$ `Some` Q *if, and only if,* $P \xrightarrow{x:o} P'$. *We have* `eval` (x, o) $P =$ `None` *if, and only if,* $\forall P' \colon P \not\xrightarrow{x:o} P'$.

The next result follows trivially, where the decidability algorithm is function `eval`. The type signature deserves some discussion. Type `sum A B` holds two constructors, one holds a value of type `A`, the other a value of type `B`. In this case, each of which is a Σ-type. The first type, $\{P' \mid P \xrightarrow{x:o} P'\}$, pairs a phaser P' with a proof of reduction. The second type, $\{u \colon \texttt{unit} \mid \forall P' \colon P \not\xrightarrow{x:o} P'\}$, pairs a unit value with a proof that there is no possible reduction for task x and action o. The unit value only serves as a witness type to associate the proof object with.

Theorem 4 (Reduction is decidable)**.** *For all x, o, and P, we have that* `sum` $\{P' \mid P \xrightarrow{x:o} P'\}$ $\{u \colon$ `unit` $\mid \forall P' \colon P \not\xrightarrow{x:o} P'\}$.

Theorem 5 (Decidable Happens-Before)**.** *Well-formed views, well-ordered phasers are decidable and so is the happens-before relation.*

- *For all v and v′, we have that* $\{v \in \mathcal{V}^{WF}\} + \{v \notin \mathcal{V}^{WF}\}$ *and also* $\{v \prec v'\} + \{v \not\prec v'\}$.
- *For all phasers P and P′ we have* $\{P \in \mathcal{P}^{WO}\} + \{P \notin \mathcal{P}^{WO}\}$ *and also* $\{P \prec P'\} + \{P \not\prec P'\}$.

7. Formalization in the browser

Having an executable implementation of the reduction relation helps users understand and exercise the phasers semantics. In this section, we present, as a companion artifact, a web page (link below) that lets users exercise the definitions included in this paper.

`http://cogumbreiro.github.io/jlamp17/`

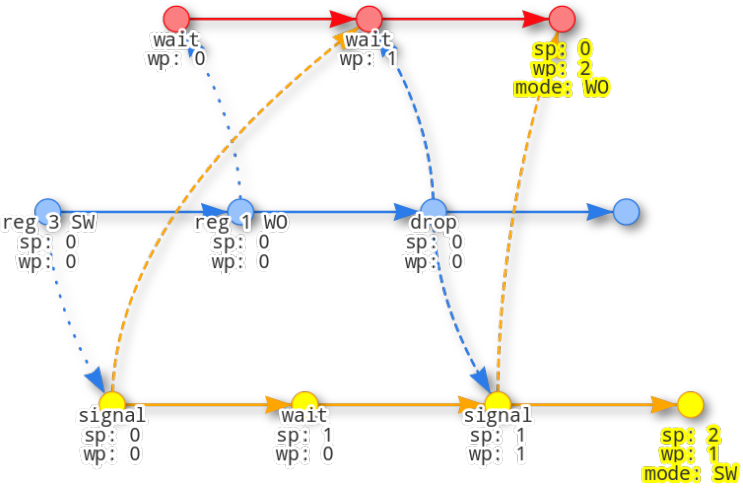
The artifact includes a JavaScript library that is translated automatically from our Coq formalization. The translation process is two parted. Coq can translate any decidable definition as an OCaml function. `js_of_ocaml`[6] can translate OCaml code into JavaScript. Thus, we use Coq to translate Definitions 1 to 4 and Figure 2 as an OCaml library and then use `js_of_ocaml` to expose this OCaml library as a JavaScript library. The latter step requires some programming to expose OCaml functions as JavaScript, as code generated from `js_of_ocaml` cannot be invoked from regular JavaScript as is; this programming consists of data serialization code.

The web page includes a prompt to evaluate JavaScript expressions, so that the user can invoke our library. Additionally, we include a graph-visualization widget that renders the happens-before relation on a sequence of operations. The graph-generation code is also implemented in Coq, but we leave a theoretical discussion of this feature as future work.

[6] `http://ocsigen.org/js_of_ocaml/`

```
[[1, wait()]
 [3, signal()],
 [1, wait()],
 [3, wait()],
 [3, signal()],
 [2, drop()],
 [2, reg(1, WO)],
 [2, reg(3, SW)]]
```

(a) Operation trace.



(b) Happens-before visualization.

Figure 3: Depicting the running example in our JavaScript executable semantics.

JavaScript library. The web page documents the available functions along with examples of their usage. We briefly discuss the reduction relation and the happens-before relation. The prompt evaluates the user input and prints the result of evaluation. Any code after >>> is a user input. The subsequent line displays the result of evaluating the user expression.

In the following example we declare a phaser with two tasks — task identifiers are non-negative integers encoded as strings. The code below assigns a phaser to variable `p1` and then displays the state of `p1` back to the user.

```
>>> p1 = {"1": {sp:0,wp:0,mode:SW}, "2":{sp:0,wp:0,mode:SW}}
{"1":{"sp":0,"wp":0,"mode":"SW"},"2":{"sp":0,"wp":0,"mode":"SW"}}
```

Function `hj.run(e,ph)` reduces state `ph` and an event `e` (which represents a task-operation pair); failing to reduce throws an exception. An event is encoded as a two-element array `[t,o]` where `t` is a non-negative integer and `o` is an operation. For instance, in the next code listing, task 1 signals phaser `ph1` and returns the new state.

```
>>> hj.run([1,signal()], p1)
{"1":{"wp":0,"sp":1,"mode":"SW"},"2":{"wp":0,"sp":0,"mode":"SW"}}
```

Function `hj.hb(ph1,ph2)` returns `true` if, and only if, `ph1` happens before `ph2`. As we know, from Lemma 4, the input of `hj.run` cannot happen before its output, hence the return value of the following expression.

```
>>> hj.hb(p1, hj.run([1,signal()], p1))
false
```

Graph visualization. The syntax of the input is a JavaScript array that consists of events. The array of events is ordered from the most recent at position 0 to the least recent (*i.e.,* first operation performed). Figure 3a lists the operation trace and Figure 3b shows the graph rendering. Each event corresponds to a node in the graph; the solid and dotted edges can be mapped to the reduction relation over a pair of states. All nodes of the same task id share the same color. The dashed arrows depict the happens-before relation among states. The nodes highlighted in yellow represent the task views of each task, constituting the phaser state. For instance, task 2, in blue, does not have a task view associated to a node, since it is deregistered from the phaser.

8. Conclusion

In this paper we propose the first formalization of Habanero phaser semantics and of Phase Ordering, from which we derive the May-Happen-In-Parallel (MHP) and Happens-Before (HB) relations for phaser operations, as part of an

ongoing effort to formalize the Habanero programming model. Our definitions and proofs are mechanized using the Coq proof assistant, consisting of 2800 lines of code and 140 lemmas.

We discuss the design of our Coq formalization as well as implement the reduction relation as a web application, compiled automatically from our Coq code as a JavaScript library. Our implementation of the reduction rules are proved to be correct with respect to the semantics we defined. The code responsible for constructing the graph used in our web application consists of 1900 lines of Coq code, plus 200 lines of OCaml code.

Our next step is to verify data-race errors in parallel programs that feature collective producer-consumer synchronization patterns. Since race detectors are based on analyzing a causality graph of memory accesses, we need to define the phaser semantics in terms of a causality graph and show that the Phase Ordering property captures the causality relation.

References

[1] J. Shirako, D. M. Peixotto, V. Sarkar, W. N. Scherer, Phasers: a unified deadlock-free construct for collective and point-to-point synchronization, in: ICS'08, ACM, 2008, pp. 277–288. doi:10.1145/1375527.1375568.

[2] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, V. Sarkar, X10: an object-oriented approach to non-uniform cluster computing, in: OOPSLA'05, ACM, 2005, pp. 519–538. doi:10.1145/1103845.1094852.

[3] T. Cogumbreiro, R. Hu, F. Martins, N. Yoshida, Dynamic deadlock verification for general barrier synchronisation, in: PPoPP'15, ACM, 2015, pp. 150–160. doi:10.1145/2688500.2688519.

[4] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Communications of the ACM 21 (7) (1978) 558–565. doi:10.1145/359545.359563.

[5] E. Duesterwald, M. L. Soffa, Concurrency analysis in the presence of procedures using a data-flow framework, in: TAV'91, ACM, 1991, pp. 36–48. doi:10.1145/120807.120811.

[6] R. Sharma, M. Bauer, A. Aiken, Verification of producer-consumer synchronization in gpu programs, in: PLDI'15, ACM, 2015, pp. 88–98. doi:10.1145/2737924.2737962.

[7] Y. Cai, S. Wu, W. K. Chan, ConLock: A constraint-based approach to dynamic checking on deadlocks in multithreaded programs, in: ICSE'14, ACM, 2014, pp. 491–502. doi:10.1145/2568225.2568312.

[8] T. Elmas, S. Qadeer, S. Tasiran, Goldilocks: Efficiently computing the happens-before relation using locksets, in: FATES'06/RV'06, Springer, 2006, pp. 193–208. doi:10.1007/11940197_13.

[9] P. Maiya, A. Kanade, R. Majumdar, Race detection for android applications, in: PLDI '14, ACM, 2014, pp. 316–325. doi:10.1145/2594291.2594311.

[10] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, C. Flanagan, Sound predictive race detection in polynomial time, in: POPL'12, ACM, 2012, pp. 387–400. doi:10.1145/2103656.2103702.

[11] S. Crafa, D. Cunningham, V. Saraswat, A. Shinnar, O. Tardieu, Semantics of (resilient) x10, in: R. Jones (Ed.), ECOOP'14, Vol. 8586 of LNCS, Springer, 2014, pp. 670–696. doi:10.1007/978-3-662-44202-9_27.

[12] P. Feautrier, E. Violard, A. Ketterlin, Improving the performance of X10 programs by Clock removal, in: CC'14, Vol. 8409 of LNCS, Springer, 2014, pp. 113–132. doi:10.1007/978-3-642-54807-9_7.

[13] T. Yuki, P. Feautrier, S. V. Rajopadhye, V. Saraswat, Checking race freedom of clocked X10 programs, CoRR abs/1311.4305. URL http://arxiv.org/abs/1311.4305

[14] S. Joshi, R. K. Shyamasundar, S. K. Aggarwal, A new method of MHP analysis for languages with dynamic barriers, in: IPDPSW'12, IEEE, 2012, pp. 519–528. doi:10.1109/IPDPSW.2012.70.

[15] P. S. Almeida, C. Baquero, V. Fonte, Version stamps-decentralized version vectors, in: Proceedings 22nd International Conference on Distributed Computing Systems, 2002, pp. 544–551. doi:10.1109/ICDCS.2002.1022304.

[16] T. Cogumbreiro, J. Shirako, V. Sarkar, Formalization of phase ordering, in: PLACES'16, 2016, pp. 13–24. doi:10.4204/EPTCS.211.2.

[17] C. Baquero, N. Preguiça, Why logical clocks are easy, Communications of the ACM 59 (4) (2016) 43–47. doi:10.1145/2890782.

[18] R. Schwarz, F. Mattern, Detecting causal relationships in distributed computations: In search of the holy grail, Distributed Computing 7 (3) (1994) 149–174. doi:10.1007/BF02277859.

[19] P. Martin-Lf, G. Sambin, Intuitionistic type theory, Studies in proof theory, Bibliopolis, 1984.