Coordinating phased activities while maintaining progress

Tiago Cogumbreiro, Francisco Martins, and Vasco Thudichum Vasconcelos

LaSIGE, Faculty of Sciences, University of Lisbon

Abstract. In order to develop reliable applications for parallel machines, programming languages and systems need to provide for flexible parallel programming coordination techniques. Barriers, clocks and phasers constitute promising synchronisation mechanisms, but they exhibit intricate semantics and allow writing programs that can easily deadlock. We present an operational semantics and a type system for a fork/join programming model equipped with a flexible variant of phasers. Our proposal allows for a precise control over the maximum number of synchronisation steps each task can be ahead of others. A type system ensures that programs do not deadlock, even when they use multiple phasers.

1 Introduction

The key to develop scalable parallel applications lies in using coordination mechanisms at the "right" level of abstraction [8]. Rather than re-inventing the wheel with *ad hoc* solutions [20], programmers should resort to off-the-shelf coordination mechanisms present in programming languages and systems. Barriers, in their multiple forms [1,5,6,7,9,11,14] constitute one such coordination mechanism. A barrier allows multiple tasks to synchronise at a single point, in such a way that: a) before synchronisation *no task* has crossed the barrier, and b) after synchronisation *all tasks* have crossed the barrier.

Programs that use a single barrier to coordinate all their tasks do not deadlock. If using a single barrier may reveal itself quite limited in practice, groups of tasks that use multiple barriers may easily deadlock. To address this issue, the X10 programming language [7] proposes clocks, a deadlock-free coordination mechanism. Clocks later inspired primitives in other languages, such as Java [13] (as of version 7), Habanero Java [17] (HJ), and an extension to OpenMP [18].

For some applications, the semantics of traditional barriers is overly inflexible. With this mind, Shirako *et al.* introduced *phasers* [17], a primitive that allows some tasks to cross the barrier before synchronisation, thus relaxing condition a) above. Phasers allow for asymmetric parallel programs, including multiple producer/single consumer applications where, at each iteration, the consumer waits for the producers to cooperate in assembling an item. In a different direction, Albrecht *et al.* proposed a *partial barrier* construct [3] that only requires some tasks to arrive at the barrier, thus relaxing condition b). This form of barriers

can be used in applications that synchronise on a subset of all tasks, allowing, *e.g.*, computation to progress quickly even when in presence of slow tasks.

Reasoning about such enriched constructs is usually far from trivial: the semantics is intricate and most languages lack a precise specification (including Java barriers and HJ phasers). In this work we propose a calculus for a fork/join programming model that unifies the various forms of barriers, including X10 clocks [7], HJ phasers (both bounded [16] and unbounded [17]), and Java barriers [13].

Our proposal not only subsumes those of X10, HJ and Java, but further increases the flexibility of the coordination mechanism. We allow tasks to be ahead of others up to a *bounded* number of synchronisation phases, and yet guarantee that well-typed programs are deadlock free. In contrast, HJ allows tasks to be ahead of others by an arbitrary number of phases, which is of limited interest, since fast tasks may eventually exhaust computational resources, such as buffer space.

To summarise, our contributions are:

- a flexible barrier construct that allows for a precise control over the maximum number of phases each task can be ahead of others;
- an operational semantics for a fork/join programming model that captures barrier-like coordination patterns found in X10, HJ, and Java;
- a type system that ensures progress, hence the absence of deadlocks, in addition to the usual type preservation.

The paper is organised as follows. The next section addresses related work. Section 3 presents the syntax and the operational semantics of our language. Thereafter, we introduce the type system and the main results of our work. Section 5 concludes the paper while putting forward lines of future research.

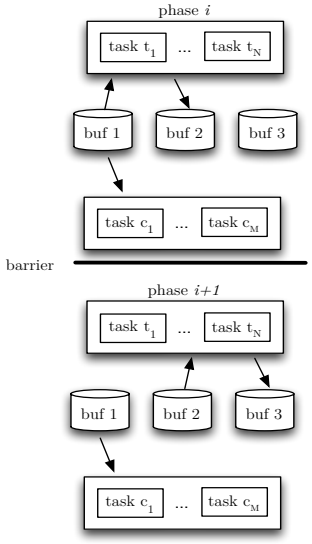
2 Related work via an example

Figure 1a sketches a parallel breadth-first search algorithm to find an exit in a labyrinth. The algorithm uses two groups of tasks: one traverses the labyrinth (modelled by a graph) and another inspects the visited nodes for an exit. The sketch was originally implemented in OpenMP by Süß and Leopold [19]. The algorithm proceeds iteratively, handling at each step (or phase) all nodes at the same depth. If a node is an exit, the algorithm terminates; otherwise it computes the node's neighbours and places them in a buffer to be processed in a later phase. The tasks synchronise at the end of each phase.

Figure 1b depicts two execution states for phases i and $i+1$. At phase i, tasks $t_1, \ldots, t_N$ read nodes from buf 1 (that stores the nodes of depth level i), compute their descendants, and then in buf 2 (via function traverseNodes()). Tasks $c_1, \ldots, c_M$ read nodes from buf 1 and look for an exit node (function checkNodesForAnExit()). Both groups of tasks synchronise (at phaser c) and advance to phase $i+1$. Notice that at phase $i+1$ the tasks $c_1, \ldots, c_M$ continue to process buf 1, while the traversal group $t_1, \cdots, t_N$ is handling nodes at depth

```
1  finish (
2    let t = newPhaser 0 in
3    let c = newPhaser 0 in (
4      for (int j=0; j<N; j++)
5        async {c:K, t:0} (
6          while(!exitFound) (
7            traverseNodes();
8            arrive c; arrive t; awaitAll);
9          drop c; drop t);
10     drop t;
11     for (int j=0; j<M; j++)
12       async {c:0} (
13         while(!exitFound) (
14           arrive c; awaitAll;
15           checkNodesForAnExit());
16         drop c)
17     drop c)
18 ) // join task created in lines 5,12
```

(a) Algorithm



(b) Execution diagram (K=2)

Fig. 1: A parallel breadth-first search algorithm.

level $i + 1$ (buf 2). This situation is possible due to the bound K assigned to phaser c upon launching the traversal tasks (line 5). Bound K denotes the number of phases the traversal tasks may be ahead of the checker tasks. The number of buffers is equal to K + 1.

The synchronisation of tasks is challenging and involves phasers c and t. Phaser c is used to synchronise traversal tasks with checker tasks, while the traversal tasks further synchronise among themselves at phaser t. The bound c:K in line 5 reads as "phaser b has bound K in the spawned task." On the other hand, bound t:0 (also in line 5) means that the traversal activities must all be in the same phase. This synchronisation scheme ensures that the traversal tasks must process all nodes at the same depth and only after that advance to the next level. The traversal tasks set the pace for the checker tasks with phaser c. In their turn, the checker tasks (lines 12–16) use bound c:0, thus enforcing that they simultaneously process the same depth level and do not overtake the tasks in the traversal group.

Barriers, clocks, and phasers are insufficient for this sort of coordination. Barriers and clocks are too inflexible, since no task can cross the barrier before the others arrive. HJ phasers are too loose, since when not used as barriers, tasks run unconstrained and may overflow buffers. Phasers beams [16] are a step towards this sort of control, but its semantics is only informally described and they do not guarantee deadlock-freedom. Our proposal permits different tasks to specify the maximum number of phases they can be ahead of the slowest.

In contrast, phaser beams define such a number on a per-phaser basis. To unify regular phasers and phaser beams, we also supply an operation that skips waiting on a phase.

Saraswat and Jagadeesan presented a calculus for the X10 language that includes barriers, fork/join, conditional atomic blocks, and hierarchical shared memory [15]. The authors define a small-step operational semantics and claim that X10 programs without conditional atomic blocks do not deadlock, yet no deadlock-freedom theorem is formally proved. Lee and Palsberg present a calculus, called FX10, with two constructs from X10: fork/join and atomic blocks [10]. FX10 is suited for inter-procedural analysis through type inference and includes a formal proof of a fragment of the deadlock theorem stated by Saraswat and Jagadeesan. The type system used to identify may-happen-parallelism is further explored in [2]. Other formal studies on fork/join semantics include [1,4]. Our previous work defines an operational semantics and a type system for a calculus with a fork/join programming model and clocks [12], but does not include a deadlock-freedom theorem.

X10 clocks and HJ phasers are language-based approaches, whereas the Java barrier (also called phaser) is a library-based approach. Features that appear simultaneous in our calculus, in X10, and in HJ are: controlled barrier registration to avoid non-determinism, advancement on multiple barriers, and enforced barrier deregistration before activity termination to avoid barrier-related deadlocks. Features that appear in our calculus and in HJ alone include phaser visibility restricted to finish scopes to avoid deadlocks between phaser and finish; enforced deregistration before terminating a finish to avoid deadlocks between barriers. HJ and X10 automatically deregister from barriers when activities terminate. Instead we require explicit operations on phasers, in order to obtain a clearer operational semantics. Our type system provides enough information to guide a compiler into automatically inserting such operations, if desired. Finally, the ability to specify the maximum number of phases a task may be ahead of the slowest is unique to our language.

3 Syntax and operational semantics

Our language, inspired by X10 and HJ, uses activities to organise independent computations and features two coordination mechanisms to control concurrency: phaser and finish. For the sake of simplicity, the language focuses on task coordination, providing very little in the way of describing complex computations.

The syntax of the language is defined in Figure 2. It relies on a base set of variables, ranged over by x, and a set of natural numbers, ranged over by m and n. Bounds B map phaser names to the phaser bounds. We use the standard abbreviation $t; e$ to denote the expression $\textsf{let}\ x = t\ \textsf{in}\ e$ when variable x does not occur in expression e. A term t is transformed into an expression via $\textsf{let}\ x = t\ \textsf{in}\ x$. We describe the language constructs along with the presentation of the operational semantics.

e	$::=$	*Expressions*	t	$::=$	*Terms*
	v	value		e	expression
	$\mid$ **let** $x = t$ **in** e	local declaration		$\mid$ **newPhaser** v	new phaser, bounded by v
v	$::=$	*Values*		$\mid$ **drop** v	deregister from phaser v
	x	variable		$\mid$ **arrive** v	arrive at phaser v
	$\mid n$	natural number		$\mid$ **awaitAll**	await on registered phasers
	$\mid ()$	unit		$\mid$ **skipAll**	skip phase on all phasers
B	$::=$	*Bounds*		$\mid$ **async** B e	fork an activity
	$\emptyset$	empty		$\mid$ **finish** e	wait for termination
	$\mid B \uplus \{v \colon v\}$	bound			

Fig. 2: The syntax of expressions.

S	$::= \langle Q; A \rangle$	*States*	A	$::= \emptyset \mid A \uplus \{l \colon a\}$	*Activity maps*
Q	$::= \emptyset \mid Q \uplus \{h \colon P\}$	*Phaser maps*	a	$::= \langle B; e \rangle \mid \langle S; B; e \rangle$	*Activities*
P	$::= \emptyset \mid P \uplus \{l \colon r\}$	*Phasers*	v	$::= \ldots \mid h$	*Values*
r	$::= \langle n; s \rangle$	*Local views*	s	$::=$ **un** $\mid$ **ar**	*Arrival status*

Fig. 3: The syntax states.

Figure 3 introduces the syntax of a machine state. The run-time system relies on two additional disjoint sets, $\mathcal{H}$ and $\mathcal{L}$. Set $\mathcal{H}$ contains *phaser names*, ranged over by h. *Activity names*, l, are taken from set $\mathcal{L}$. A state S of a computation comprises a shared *phaser map* Q and an *activity map* A. Each phaser map Q stores the available phasers, mapping phaser names to phasers. A phaser maps activity names to *local views*. A local view consists of the phase n the activity is in and an arrival status s set to **ar** when the activity arrives at the phaser. An activity map A maps activity names l to activities a. There are two kinds of activities: regular activities $\langle B; e \rangle$ consist of the bound for each phaser the activity is registered with, and an expression e; finish activities $\langle S; B; e \rangle$ additionally include a state S comprising the activities spawned within a **finish** instruction. Given any map, say X, we write $\operatorname{dom} X$ for the domain of X and $\operatorname{range} X$ for the co-domain of X. In a map $X \uplus \{x \colon u\}$ we assume x does not occur in $\operatorname{dom} X$.

Figure 4 introduces a small step reduction relation on states, $S_1 \to S_2$, capturing the non-deterministic choice of which activity to evaluate next. Auxiliary functions and predicates are in Figure 5. The phaser creation instruction, **newPhaser** n, evaluates to a fresh phaser name h; it also registers under h the current activity l with a local view composed of bound n and phase 0. All other phaser-related operations evaluate to $()$. Activities deregister from a phaser h via an expression **drop** h, thus removing the local view from the phaser. Instruc-

$$\frac{h \notin \operatorname{dom} Q}{\begin{array}{l}\langle Q \;;\; A \uplus \{l : \langle B; \mathsf{let}\ x = \mathsf{newPhaser}\ n\ \mathsf{in}\ e\rangle\}\rangle \\ \to \langle Q \uplus \{h : \{l : \langle 0; \mathsf{un}\rangle\}\} \;;\; A \uplus \{l : \langle B \uplus \{h : n\}; \mathsf{let}\ x = h\ \mathsf{in}\ e\rangle\}\rangle\end{array}} \text{(R-PHASER)}$$

$$\frac{}{\begin{array}{l}\langle Q \uplus \{h : (P \uplus \{l : \_\})\} \;;\; A \uplus \{l : \langle B \uplus \{h : \_\}; \mathsf{let}\ x = \mathsf{drop}\ h\ \mathsf{in}\ e\rangle\}\rangle \\ \to \langle Q \uplus \{h : P\} \;;\; A \uplus \{l : \langle B; \mathsf{let}\ x = ()\ \mathsf{in}\ e\rangle\}\rangle\end{array}} \text{(R-DROP)}$$

$$\frac{}{\begin{array}{l}\langle Q \uplus \{h : (P \uplus \{l : \langle n; \mathsf{un}\rangle)\}\} \;;\; A \uplus \{l : \langle B; \mathsf{let}\ x = \mathsf{arrive}\ h\ \mathsf{in}\ e\rangle\}\rangle \\ \to \langle Q \uplus \{h : (P \uplus \{l : \langle n; \mathsf{ar}\rangle)\}\} \;;\; A \uplus \{l : \langle B; \mathsf{let}\ x = ()\ \mathsf{in}\ e\rangle\}\rangle\end{array}} \text{(R-ARRIVE)}$$

$$\frac{\operatorname{unblocked}(Q, l, B)}{\begin{array}{l}\langle Q \;;\; A \uplus \{l : \langle B; \mathsf{let}\ x = \mathsf{awaitAll}\ \mathsf{in}\ e\rangle\}\rangle \\ \to \langle \operatorname{commit}(l, Q) \;;\; A \uplus \{l : \langle B; \mathsf{let}\ x = ()\ \mathsf{in}\ e\rangle\}\rangle\end{array}} \text{(R-AWAIT)}$$

$$\frac{}{\begin{array}{l}\langle Q \;;\; A \uplus \{l : \langle B; \mathsf{let}\ x = \mathsf{skipAll}\ \mathsf{in}\ e\rangle\}\rangle \\ \to \langle \operatorname{commit}(l, Q) \;;\; A \uplus \{l : \langle B; \mathsf{let}\ x = ()\ \mathsf{in}\ e\rangle\}\rangle\end{array}} \text{(R-SKIP)}$$

$$\frac{l_2 \notin \operatorname{dom} A \cup \operatorname{dom} B_1 \cup \operatorname{dom} B_2}{\begin{array}{l}\langle Q \;;\; A \uplus \{l_1 : \langle B_1; \mathsf{let}\ x = \mathsf{async}\ B_2\ e_2\ \mathsf{in}\ e_1\rangle\}\rangle \\ \to \langle \operatorname{copy}(l_1, l_2, \operatorname{dom} B_2, Q) \;;\; A \uplus \{l_1 : \langle B_1; \mathsf{let}\ x = ()\ \mathsf{in}\ e_1\rangle\} \uplus \{l_2 : \langle B_2; e_2\rangle\}\rangle\end{array}} \text{(R-ASYNC)}$$

$$\frac{l_2 \notin \operatorname{dom} A \cup \operatorname{dom} B}{\begin{array}{l}\langle Q; A \uplus \{l_1 : \langle B; \mathsf{let}\ x = \mathsf{finish}\ e_1\ \mathsf{in}\ e_2\rangle\}\rangle \\ \to \langle Q; A \uplus \{l_1 : \langle \langle \emptyset; \{l_2 : \langle \emptyset; e_1\rangle\}\rangle; B; \mathsf{let}\ x = ()\ \mathsf{in}\ e_2\rangle\}\rangle\end{array}} \text{(R-FINISH)}$$

$$\frac{S_1 \to S_2}{\langle Q; A \uplus \{l : \langle S_1; B; e\rangle\}\rangle \to \langle Q; A \uplus \{l : \langle S_2; B; e\rangle\}\rangle} \text{(R-ACTIVITY)}$$

$$\frac{\operatorname{halted}(S)}{\langle Q; A \uplus \{l : \langle S; B; e\rangle\}\rangle \to \langle Q; A \uplus \{l : \langle B; e\rangle\}\rangle} \text{(R-JOIN)}$$

$$\langle Q; A \uplus \{l : \langle B; \mathsf{let}\ x = v\ \mathsf{in}\ e\rangle\}\rangle \to \langle Q; A \uplus \{l : \langle B; e[v/x]\rangle\}\rangle \quad \text{(R-LET)}$$

$$\frac{}{\begin{array}{l}\langle Q; A \uplus \{l : \langle B; \mathsf{let}\ x = (\mathsf{let}\ y = e_1\ \mathsf{in}\ t)\ \mathsf{in}\ e_2\rangle\}\rangle \\ \to \langle Q; A \uplus \{l : \langle B; \mathsf{let}\ y = e_1\ \mathsf{in}\ (\mathsf{let}\ x = t\ \mathsf{in}\ e_2)\rangle\}\rangle\end{array}} \text{(R-UNFOLD)}$$

Fig. 4: Small step semantics for states, $S \to S$.

tion **arrive** h marks the local view of activity l as arrived, **ar**. An activity can only **arrive** once per phase.

Instruction **awaitAll** waits until activity l becomes *unblocked*. The current phase of a phaser is the natural number corresponding to the smallest local view among all activities registered in the phaser (Figure 5). An activity l is unblocked when it has arrived at all phasers *and* each bound allows progress (i.e., the bound is larger than the difference between the current phase and the phaser's phase). For each phaser activity l is registered with, rule R-AWAIT advances the phase and sets the arrival status back to **un**, via function $\operatorname{commit}(l, Q)$. HJ and X10 implicitly arrive at all non-arrived barriers before advancing.

Expression **skipAll** simply advances the phase and does not wait for other activities. This operation can be used to let an activity use a phaser repeatedly without waiting for others. Spawning a new activity with rule R-ASYNC

Phase function for local views, $\mathrm{phase}(r) = n$:

$$\mathrm{phase}(\langle n; \mathbf{un} \rangle) = n \qquad \mathrm{phase}(\langle n; \mathbf{ar} \rangle) = n + 1$$

Phase partial function for phasers, $\mathrm{phase}(P) = n$:

$$\mathrm{phase}(P) = \min\{\mathrm{phase}(r) \mid r \in \mathrm{range}\, P\}$$

Unblocked predicate, $\mathrm{unblocked}(Q, l, B)$: $\qquad \mathrm{unblocked}(Q, l, \emptyset)$

$$\frac{\mathrm{unblocked}(Q, l, B) \qquad Q(h) = P \qquad P(l) = \langle n; \mathbf{ar} \rangle \qquad m > n - \mathrm{phase}(P)}{\mathrm{unblocked}(Q, l, B \uplus \{h \colon m\})}$$

Phase commit partial function, $\mathrm{commit}(l, Q) = Q$:

$$\mathrm{commit}(l, Q \uplus \{h \colon P \uplus \{l \colon \langle n; \mathbf{ar} \rangle\}\}) = \mathrm{commit}(l, Q) \uplus \{h \colon P \uplus \{l \colon \langle n + 1; \mathbf{un} \rangle\}\}$$

$$\frac{l \notin \mathrm{dom}\, P}{\mathrm{commit}(l, Q \uplus \{h \colon P\}) = \mathrm{commit}(l, Q) \uplus \{h \colon P\}} \qquad \mathrm{commit}(l, \emptyset) = \emptyset$$

Local view copy partial function, $\mathrm{copy}(l_1, l_2, H, Q) = Q$:

$$\frac{h \in H \qquad P(l_1) = r}{\mathrm{copy}(l_1, l_2, H, Q \uplus \{h \colon P\}) = \mathrm{copy}(l_1, l_2, H, Q) \uplus \{h \colon P \uplus \{l_2 \colon r\}\}}$$

$$\frac{h \notin H}{\mathrm{copy}(l_1, l_2, H, Q \uplus \{h \colon P\}) = \mathrm{copy}(l_1, l_2, H, Q) \uplus \{h \colon P\}} \qquad \mathrm{copy}(l_1, l_2, H, \emptyset) = \emptyset$$

Halted state predicate, $\mathrm{halted}(S)$: $\qquad \mathrm{halted}(\mathbf{\langle\!\langle} Q; \{l_1 \colon \langle \emptyset; v_1 \rangle, \ldots, l_n \colon \langle \emptyset; v_n \rangle\} \mathbf{\rangle\!\rangle})$

Fig. 5: Phaser-related functions and predicates.

evaluates expression e concurrently, by augmenting the activity map with a new activity $\langle B_2; e \rangle$. For each phase in bounds B_2, we copy the local views from the spawning activity l_1 to the spawned activity l_2, as captured by function $\mathrm{copy}(l_1, l_2, H, Q)$, where H is a set of phaser names, defined in Figure 5. Spawned activities inherit the arrival statuses, for otherwise, depending on the order of reduction of the spawning and spawned activities, the spawned activity may or may not participate in the synchronisation of the current phase, inducing an undesirable non-determinism in the phaser semantics [12].

Rule R-FINISH evaluates expressions of the form **let** $x =$ **finish** e_1 **in** e_2 by suspending expression **let** $x = ()$ **in** e_2 and by evaluating e_1 in a newly created state. The finish activity (a triple as in Figure 3) holds a state (comprising an empty phaser map and an activity $\langle \emptyset; e_1 \rangle$ that is not registered on any phaser), the current bounds B, and the suspended expression **let** $x = ()$ **in** e_2. Such an activity will then reduce, via rule R-ACTIVITY, until halted (a predicate introduced in Figure 5). Then, the suspended activity resumes execution by means of rule R-JOIN.

The evaluation of let-expressions is standard. Rule R-LET replaces variable x by value v in continuation e. Nested let bindings are unfold with rule R-UNFOLD.

We complete this section with a pair of examples leading to deadlocks, thus motivating the need for the type system in the next section. An activity that, before terminating, does not deregister from every phaser it is registered with will cause every activity that synchronises with that phaser to deadlock. Consider a program composed of two activities. Activity l1 creates a phaser x (line 2) and in the subsequent line spawns an activity l2 also registered with x; the latter activity does nothing (not even deregisters from phaser x). Activity l1 synchronises and deadlocks at line 5, forever waiting for activity l2 to **arrive**.

```
1  // activity l1
2  let x = newPhaser 0 in
3  async {x:0} (); // forgets drop x
4  arrive x;
5  awaitAll // l1 deadlocks here
```

Listing 1.1: An activity that forgets to deregister from a phaser.

The deadlocked state:

$$\langle\{h\colon \{l_1\colon \langle 0; \mathbf{ar}\rangle, l_2\colon \langle 0; \mathbf{un}\rangle\}\};\ \{l_1\colon \langle\{h\colon 0\}; \mathbf{let}\ w = \mathbf{awaitAll}\ \mathbf{in}\ w\rangle,\ l_2\colon \langle\{h\colon 0\}; ()\rangle\},\}\rangle$$

An activity that forgets to **arrive** before awaiting other activities deadlocks itself and the remaining participants in the synchronisation. In the next program, activity l1 creates a phaser x and spawns activity l2 that simply arrives at x (line 4) and awaits for l1 (line 5). Activity l1 forgets to arrive at x but still awaits (line 6) and therefore deadlocks along with activity l2 (in line 5).

```
1  // activity l1
2  let x = newPhaser 0 in
3  async {x:0} ( // activity l2
4    arrive x;
5    awaitAll); // l2 deadlocks here
6  awaitAll // l1 deadlocks here
```

Listing 1.2: An activity that forgets to arrive at a phaser.

The deadlocked state:

$$\langle\{h\colon \{l_1\colon \langle 0; \mathbf{un}\rangle, l_2\colon \langle 0; \mathbf{ar}\rangle\}\};\ \{l_1\colon \langle\{h\colon 0\}; \mathbf{let}\ w = \mathbf{awaitAll}\ \mathbf{in}\ w\rangle,\ l_2\colon \langle\{h\colon 0\}; \mathbf{let}\ z = \mathbf{awaitAll}\ \mathbf{in}\ z\rangle\}\rangle$$

4 Type system and results

This section introduces our type system and its main results, namely type preservation and progress for typable states.

We rely on a set of type variables, ranged over by α. The syntax of types is defined by the grammar in Figure 6, and include those for the unit constant, **unit**, for the natural numbers, **nat**, and for phasers, α. We assign a different (singleton) type α to each phaser, in order to track how phasers are used throughout the program. The type system for our programming language is also defined in Figure 6. *Typings* Γ are maps from variables and phaser names to types. *Arrival maps* Φ map type variables (singleton phaser types) into arrival status. The re-

The syntax of types:

$$\tau \ ::= \ \mathsf{unit} \ \mid \ \mathsf{nat} \ \mid \ \alpha$$

Well-formed types, $\Phi \vdash \tau$:

$$\Phi \vdash \mathsf{unit} \qquad \Phi \vdash \mathsf{nat} \qquad \Phi, \alpha : s \vdash \alpha \qquad \text{(T-WF-U, T-WF-N, T-WF-P)}$$

Typing rules for bounds, $\Gamma; \Phi \vdash B : \Phi$:

$$\frac{\Gamma; \Phi_1 \vdash B : \Phi_2 \quad \Gamma; \Phi_1 \vdash v_1 : \alpha \quad \Gamma; \Phi_1 \vdash v_2 : \mathsf{nat}}{\Gamma; \Phi_1 \vdash (B \uplus \{v_1 : v_2\}) : (\Phi_2 \uplus \{\alpha : \Phi_1(\alpha)\})} \qquad \Gamma; \Phi \vdash \emptyset : \emptyset$$

(T-BOUND-CONS, T-BOUND-NIL)

Typing rules for values, $\Gamma; \Phi \vdash v : \tau$:

$$\Gamma; \Phi \vdash () : \mathsf{unit} \qquad \Gamma; \Phi \vdash n : \mathsf{nat} \qquad \frac{\Gamma(v) = \tau \qquad \Phi \vdash \tau}{\Gamma; \Phi \vdash v : \tau} \quad \text{(T-UNIT, T-NAT, T-VAL)}$$

Typing rules for terms and for expressions, $\Gamma; \Phi \vdash t : (\tau, \Phi)$ and $\Gamma; \Phi \vdash e : (\tau, \Phi)$:

$$\frac{\Gamma; \Phi \vdash v : \mathsf{nat} \qquad \alpha \notin \operatorname{dom} \Phi}{\Gamma; \Phi \vdash (\mathsf{newPhaser}\ v) : (\alpha, \Phi \uplus \{\alpha : \mathsf{un}\})} \qquad \text{(T-PHASER)}$$

$$\frac{\Gamma; \Phi \uplus \{\alpha : s\} \vdash v : \alpha}{\Gamma; \Phi \uplus \{\alpha : s\} \vdash (\mathsf{drop}\ v) : (\mathsf{unit}, \Phi)} \qquad \frac{\Gamma; \Phi \uplus \{\alpha : \mathsf{un}\} \vdash v : \alpha}{\Gamma; \Phi \uplus \{\alpha : \mathsf{un}\} \vdash (\mathsf{arrive}\ v) : (\mathsf{unit}, \Phi \uplus \{\alpha : \mathsf{ar}\})}$$

(T-DROP, T-ARRIVE)

$$\Gamma; \{\alpha_1 : \mathsf{ar}, \ldots, \alpha_n : \mathsf{ar}\} \vdash \mathsf{awaitAll} : (\mathsf{unit}, \{\alpha_1 : \mathsf{un}, \ldots, \alpha_n : \mathsf{un}\}) \qquad \text{(T-AWAIT)}$$

$$\Gamma; \{\alpha_1 : \mathsf{ar}, \ldots, \alpha_n : \mathsf{ar}\} \vdash \mathsf{skipAll} : (\mathsf{unit}, \{\alpha_1 : \mathsf{un}, \ldots, \alpha_n : \mathsf{un}\}) \qquad \text{(T-SKIP)}$$

$$\frac{\Gamma; \Phi_1 \vdash B : \Phi_2 \qquad \Gamma; \Phi_2 \vdash e : (\_, \emptyset)}{\Gamma; \Phi_1 \vdash (\mathsf{async}\ B\ e) : (\mathsf{unit}, \Phi_1)} \qquad \frac{\Gamma; \emptyset \vdash e : (\tau, \emptyset)}{\Gamma; \Phi \vdash (\mathsf{finish}\ e) : (\mathsf{unit}, \Phi)}$$

(T-ASYNC, T-FINISH)

$$\frac{\Gamma; \Phi \vdash v : \tau}{\Gamma; \Phi \vdash v : (\tau, \Phi)} \qquad \frac{\Gamma; \Phi_1 \vdash e_1 : (\tau_1, \Phi_2) \qquad \Gamma \uplus \{x : \tau_1\}; \Phi_2 \vdash e_2 : (\tau_2, \Phi_3)}{\Gamma; \Phi_1 \vdash (\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) : (\tau_2, \Phi_3)}$$

(T-VALUE, T-LET)

Fig. 6: The syntax of types and the typing rules.

lation for well-formed types $\Phi \vdash \tau$ ensures that activities only make use of the phasers they are registered with.

The typing rules for bounds $\Gamma; \Phi_1 \vdash B : \Phi_2$ assign an arrival map Φ_2 to bounds B, under a context consisting of a typing Γ and an arrival map Φ_1. Rule T-BOUND-CONS ensures that B maps phasers into natural numbers, and also that it holds distinct phasers. The fact that phasers are associated to singleton types enables us to track aliasing in bounds.

The typing rules for values $\Gamma; \Phi \vdash v : \tau$ are straightforward. Rule T-VAL asserts that the value, either a variable or a phaser name, must be in typing Γ and that its type well formed.

For expressions we define a *type and effect* system $\Gamma; \Phi_1 \vdash e : (\tau, \Phi_2)$ stating that expression e is of type τ and effect Φ_2. The effects are important to track

the changes in the arrival map, forced by the evaluation of an expression. The type of a **newPhaser** term is a new singleton type α that is also introduced in the effect. All remaining terms are of type **unit**. The effect a **drop** v term is the incoming arrival map from which α was removed, so that value v cannot be further used (*cf.* rule T-VAL). Rule T-ARRIVE ensures that activities can **arrive** at phaser α only once, by requiring that the phaser's arrival status is **un** and by changing it into **ar**.

Terms **awaitAll** and **skipAll** mark the end of a phase: the rules check that all phasers have arrived and then reset the phasers to **un**. For example, in line 6 of Listing 1.2, activity l1 does not **arrive** at x, so we must type the term **awaitAll** under a typing $\{x\colon \alpha\}$ and an arrival map $\{\alpha\colon \mathbf{un}\}$ which does not succeed according to rule T-AWAIT.

In rule T-ASYNC, the spawned activity e is checked against an arrival map Φ_2 for the phasers in B. Furthermore, e must deregister from all its phasers before terminating (hence the empty effect for e). The effect of the **finish** itself is the incoming phaser map, so that the spawning activity inherits the arrival status of the phasers. For example, we reject the program in Listing 1.1, since the spawned activity does not drop all its phasers before terminating. In fact, the empty task in line 3 must be typed under context $\{\mathsf{x}\colon \alpha\}$ and phaser map $\{\alpha\colon \mathbf{un}\}$, but the arrival map is not empty for the unit term ().

Rule T-FINISH also forces e to deregister from all the phasers it has created, and therefore **finish** e has no effect on the arrival map Φ. In order to avoid deadlocks, we prevent e from accessing any existing phaser, thus eliminating (nested) dependencies between phasers and **finish**. The typing rule for **let** is standard.

The typing rules for states are introduced in Figure 7. We rely on a *set of activity names* Λ, a *phase difference* map Δ mapping pairs of activity names to integer values (not necessarily natural numbers), and a *phase difference tree* map Σ mapping activity names to pairs composed of a phase difference map (for the root activity) and a phase difference tree map (for the children activities, if any). A state $\langle\!\langle Q; A \rangle\!\rangle$ can be seen as a set of activity trees, the trees in A. Regular activities $\langle B; e\rangle$ are leaf nodes, whereas finish activities $\langle S; B; e\rangle$ are internal nodes whose children are the activities in S. When type checking a state, the topology of the phase difference tree Σ matches that of the activity tree.

We use $\Delta \vdash P$ to check that the difference of phases between activities l_i and l_j are recorded in $\Delta(l_i, l_j)$, for any pair l_i, l_j registered with P. Judgement $\Gamma \vdash_l Q\colon \Phi$ collects the arrival statuses of every phaser activity l is registered with. Judgement $\Delta; \Lambda \vdash Q\colon \Gamma$ checks the phase difference and the registered activity names for each phaser in Q, while building a context Γ for Q.

We have two rules for activities. For regular activities, rule T-ACT ensures that the bounds B and the arrival map Φ mentions the same phasers, while ensuring that expression e deregisters from all phasers when before terminating (the effect of typing the expression is the empty arrival map). For finish activities, rule T-F-ACT ensures that both the state S and the finish continuation $\langle B; e\rangle$ are

Phase difference for phasers, $\Delta \vdash P$:

$$\frac{\Delta(l_i, l_j) = n_i - n_j \qquad \forall 1 \le i, j \le k}{\Delta \vdash \{l_1 \colon \langle n_1; \_ \rangle, \ldots l_k \colon \langle n_k; \_ \rangle\}} \quad \text{(T-DIF)}$$

Arrival map of a phaser map, $\Gamma \vdash_l Q \colon \Phi$:

$$\frac{\Gamma \vdash_l Q \colon \Phi \qquad \Gamma(h) = \alpha \qquad P(l) = \langle \_; s \rangle}{\Gamma \vdash_l (Q \uplus \{h \colon P\}) \colon (\Phi \uplus \{\alpha \colon s\})} \qquad \frac{\Gamma \vdash_l Q \colon \Phi \qquad l \notin \operatorname{dom} P}{\Gamma \vdash_l (Q \uplus \{\_ \colon P\}) \colon \Phi} \qquad \Gamma \vdash_l \emptyset \colon \emptyset$$

(T-AR-CONS, T-AR-SKIP, T-AR-NIL)

Typing context of a phaser map, $\Delta; \Lambda \vdash Q \colon \Gamma$:

$$\frac{\alpha \notin \operatorname{range} \Gamma \qquad \operatorname{dom} P \subseteq \Lambda \qquad \Delta \vdash P \qquad \Delta; \Lambda \vdash Q \colon \Gamma}{\Delta; \Lambda \vdash (Q \uplus \{h \colon P\}) \colon (\Gamma \uplus \{h \colon \alpha\})} \qquad \Delta; \Lambda \vdash \emptyset \colon \emptyset$$

(T-PM-CONS, T-PM-NIL)

Typing rules for activities, $\Delta; \Sigma; \Gamma; \Phi \vdash a$:

$$\frac{\Gamma; \Phi \vdash B \colon \Phi \qquad \Gamma; \Phi \vdash e \colon (\_, \emptyset)}{\emptyset; \emptyset; \Gamma; \Phi \vdash \langle B; e \rangle} \qquad \frac{\Delta; \Sigma \vdash S \qquad \emptyset; \emptyset; \Gamma; \Phi \vdash \langle B; e \rangle}{\Delta; \Sigma; \Gamma; \Phi \vdash \langle S; B; e \rangle}$$

(T-ACT, T-F-ACT)

Typing rules for activity maps, $\Sigma; \Gamma \vdash_Q A$:

$$\frac{\Gamma \vdash_l Q \colon \Phi \qquad \Delta_1; \Sigma_1; \Gamma; \Phi \vdash a \qquad \Sigma; \Gamma \vdash_Q A}{\Sigma \uplus \{l \colon \langle \Delta_1; \Sigma_1 \rangle\}; \Gamma \vdash_Q A \uplus \{l \colon a\}} \qquad \emptyset; \Gamma \vdash_Q \emptyset$$

(T-AM-CONS, T-AM-NIL)

Typing rule for states, $\Delta; \Sigma \vdash S$:

$$\frac{\Delta; \operatorname{dom} A \vdash Q \colon \Gamma \qquad \Sigma; \Gamma \vdash_Q A}{\Delta; \Sigma \vdash \langle Q; A \rangle} \quad \text{(T-STATE)}$$

Fig. 7: Typing rules for states

well typed. To type check a state $\Delta; \Sigma \vdash \langle Q; A \rangle$, rule T-STATE uses the activity names in A and the phase difference Δ to type check the phaser map Q; it also checks that the activity map A is well typed according to the phase difference tree Σ.

We complete this section by presenting the main results of the paper.

Lemma 1. *If $\Sigma \uplus \{l \colon \langle \Delta_1; \Sigma_1 \rangle\}; \Gamma \vdash_Q A \uplus \{l \colon a\}$, then there exists Φ such that $\Gamma \vdash_l Q \colon \Phi$, $\Delta_1; \Sigma_1; \Gamma; \Phi \vdash a$, and $\Sigma; \Gamma \vdash_Q A$.*

Theorem 1 (Subject reduction). *If $\Delta_1; \Sigma_1 \vdash S_1$ and $S_1 \to S_2$, then there exists Δ_2 and Σ_2 such that $\Delta_2; \Sigma_2 \vdash S_2$.*

Proof (Sketch). The proof follows by induction on the derivation of the reduction step. In each case we have to exhibit Δ_2 and Σ_2. For R-PHASER we make use of a weakening lemma. Cases R-DROP, R-ARRIVE, R-AWAIT, R-SKIP, and R-ASYNC are similar. We prove that changes made in the phaser map after reduction have no effect on any activity besides the one under reduction. When the derivation

of the reduction step ends with rule R-ACTIVITY, we know that $\Sigma_1 = \Sigma \uplus \{l\colon \langle \Delta_1'; \Sigma_1' \rangle\}$; by induction it follows that $\Delta_2'; \Sigma_2' \vdash S_2$. We take $\Delta_2 = \Delta_1$ and $\Sigma_2 = \Sigma \uplus \{l\colon \langle \Delta_2'; \Sigma_2' \rangle\}$. We apply Lemma 1 to the hypothesis to, and the induction hypothesis to complete the proof. Case R-JOIN, and R-UNFOLD are similar to R-ACTIVITY. For R-FINISH, we know that $\Sigma_1 = \Sigma \uplus \{l_1\colon \langle \emptyset; \emptyset \rangle\}$; we take $\Delta_2 = \Delta_1$ and $\Sigma_2 = \Sigma \uplus \{l_1\colon \langle \emptyset; \{l_2\colon \langle \emptyset; \emptyset \rangle\} \rangle\}$. We use a strengthening lemma: $\Gamma; \emptyset \vdash e\colon (\tau, \emptyset)$ and Γ only contains phaser names in its domain implies $\emptyset; \emptyset \vdash e\colon (\tau, \emptyset)$. Strengthening is applied to the newly created state. For R-LET we need a substitution lemma, strengthening and weakening.

For progress, we start by extracting a total order for the activities in a typable state.

Lemma 2. *If* $\Delta; \Sigma \vdash \langle Q; \_ \rangle$ *then the relation* $\{(l_1, l_2) \mid P \in \operatorname{range} Q, P(l_1) = \langle n_1; \_ \rangle, P(l_2) = \langle n_2; \_ \rangle, n_1 \leq n_2\}$ *is a total order.*

Theorem 2 (Progress). *If* $\Delta; \Sigma \vdash S_1$ *then* S_1 *is either* $\operatorname{halted}(S_1)$ *or there is a state* S_2 *such that* $S_1 \to S_2$.

Proof (Sketch). Activities can block for a number of reasons, easily deduced from the reduction rules in Figure 4. The cases for **drop**, **arrive**, **skipAll**, **async** and **let** are easily dismissed by a simple analysis of the typing derivation rules. For example, when the reduction step ends with rule R-ASYNC, we must show that $\Delta; \Sigma \vdash \langle Q; A \uplus \{l_1\colon \langle B_1; \textsf{let } x = \textsf{async } B_2\ e_2 \textsf{ in } e_1 \rangle\} \rangle$ and $l_2 \notin \operatorname{dom}(A, B_1, B_2)$ implies that $\operatorname{copy}(l_1, l_2, \operatorname{dom} B_2, Q)$ is defined. We proceed by induction on the structure of Q. The interesting case is when Q is $Q' \uplus \{h\colon P\}$, where we must show that $l_1 \in \operatorname{dom} P$. From $\Delta; \Sigma \vdash S$ we know that $\Gamma \vdash_{l_1} Q\colon \Phi$. By showing that $\Gamma \vdash_l Q\colon \Phi$ implies $\operatorname{dom} Q = \operatorname{dom} \Gamma$, we obtain $h \in \operatorname{dom} \Gamma$. Then we show that $h \in \operatorname{dom} \Gamma$ and $\Gamma \vdash_l Q\colon \Phi$ implies $l \in \operatorname{dom} Q(h)$.

Otherwise suppose that all activities in S_1 are blocked at **awaitAll**. Lemma 2 ensures that there is a total order on activity names. Since the order is total, there is one activity name that is smaller than all others; let it be l. From $\Delta; \Sigma \vdash \langle Q; A \uplus \{l\colon \langle B; \textsf{let } x = \textsf{awaitAll in } e_1 \rangle\} \rangle$, we know that $P(l) = \langle m; \textsf{ar} \rangle$, for some m. We also know that $\operatorname{phase}(P) = \min\{n + 1 \mid \langle n; \textsf{ar} \rangle \in \operatorname{range} P\} = 1 + \min\{n \mid \langle n; \textsf{ar} \rangle \in \operatorname{range} P\} = 1 + m$. Hence l is unblocked and we have attained a contradiction.

In absence of infinite computations, it follows from the above theorem that all typable states eventually reach a halted state.

5 Conclusion and further work

We presented a calculus and a type system for a fork/join programming model with a flexible phaser mechanism. We favour explicit operations, yielding phaser operators which are simpler than those of Habanero Java [17]. Our proposal unifies the semantics of clocks [7], regular phasers [17], and phaser beams [16],

but goes further by allowing tasks to be ahead of others by a bounded number of phases.

HJ permits writing applications where the same task is registered with a phaser as a regular barrier and with another phaser that disregards all forms of synchronisation. For such cases, our **skipAll** operation is not enough. We can however introduce *unbounded local views* for tasks registered at phasers with an infinite bound (tasks that do not want to synchronise, only to influence others). An expression **advance** v, available only for activities registered with an infinite bound, would advance a single phaser. We believe that such an extension can be easily accommodated in our system.

To focus on the intricacies of synchronisation, we kept our language very simple. Extensions required for real world programming include provision for unbounded computations (in the form of recursion or loops) and for a mutable store (in the form of imperative variables). Loops can be introduced in our calculus while causing little interference with the model and results, as long as they preserve an invariant on the registered phasers at every iteration. In order to build circular buffers of phasers, HJ applications may create phasers within a loop. A possible workaround to accommodate such a feature in our language is to introduce a primitive that allocates an array of phasers.

Acknowledgements. This work was partially supported by project PTDC/EIA-CCO/122547/2010. The first author would like to thank Vivek Sarkar for welcoming him at the Habanero group at Rice University, during the year of 2012. We are grateful to Jun Shirako and anonymous referees for their feedback on this paper and to Vivek Sarkar for discussions related to phasers.

References

1. S. Aditya, J. E. Stoy, and Arvind. Semantics of barriers in a non-strict, implicitly-parallel language. In *Proceedings of FPCA'95*, pages 204–215. ACM, 1995.
2. S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel analysis of X10 programs. In *Proceedings of PPoPP'10*, pages 183–193. ACM, 2007.
3. J. Albrecht, C. Tuttle, A. C. Snoeren, and A. Vahdat. Loose synchronization for large-scale networked systems. In *Proceedings of ATEC'06*, pages 28–28. USENIX Association, 2006.
4. Arvind, J.-W. Maessen, R. S. Nikhil, and J. E. Stoy. λ_s: an implicitly parallel λ-calculus with letrec, synchronization and side-effects. *Electronic Notes Theoretical Computer Science*, 16(3):265–290, 1998.
5. F. R. Barnes, P. H. Welch, and A. T. Sampson. Barrier Synchronisation for occam-pi. In *Proceedings of PDPTA'05*, pages 173–179. CSREA Press, 2005.
6. V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: the new adventures of old X10. In *Proceedings of PPPJ'11*, pages 51–61. ACM, 2011.
7. P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of OOPSLA'05*, pages 519–538. ACM, 2005.
8. C. Cole and R. Williams. Photoshop scalability: Keeping it simple. *Queue*, 8:20–28, 2010.

9. L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *Computing in Science and Engineering*, 5(1):46–55, 1998.
10. J. K. Lee and J. Palsberg. Featherweight X10: a core calculus for async-finish parallelism. In *Proceedings of PPoPP'10*, pages 25–36. ACM, 2010.
11. D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *Proceeding of OOPSLA'09*, pages 227–242. ACM, 2009.
12. F. Martins, V. T. Vasconcelos, and T. Cogumbreiro. Types for X10 Clocks. In *Proceedings of PLACES'10*, volume 69 of *EPTCS*, pages 111–129, 2011.
13. Oracle. Java Specification Request JSR-166, 2002.
14. J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, 2007.
15. V. Saraswat and R. Jagadeesan. Concurrent clustered programming. In *Proceedings of CONCUR'05*, volume 3653 of *LNCS*, pages 353–367. Springer, 2005.
16. J. Shirako, D. Peixotto, D. Sbirlea, and V. Sarkar. Phaser beams: Integrating stream parallelism with task parallelism. In *X10 Workshop*, 2011.
17. J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *Proceedings of ICS'08*, pages 277–288. ACM, 2008.
18. J. Shirako, K. Sharma, and V. Sarkar. Unifying barrier and point-to-point synchronization in OpenMP with phasers. In *Proceeding of IWOMP'11*, volume 6665 of *LNCS*, pages 122–137. Springer, 2011.
19. M. Süß and C. Leopold. Implementing irregular parallel algorithms with OpenMP. In *Proceedings of Euro-Par'06*, pages 635–644. Springer, 2006.
20. W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In *Proceedings of OSDI'10*, pages 1–8. USENIX Association, 2010.