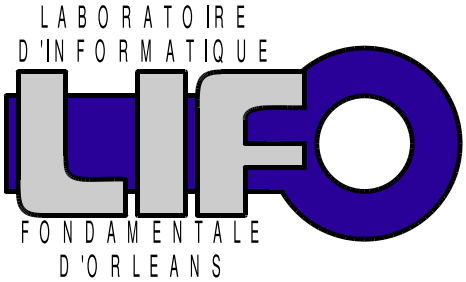
Textual Alignment in SPMD Programs

Frederic Dabrowski



4 rue Léonard de Vinci
BP 6759
F-45067 Orléans Cedex 2
FRANCE
http://www.univ-orleans.fr/lifo

Rapport de Recherche

Textual Alignment in SPMD Programs

Frédéric Dabrowski
LIFO, Université d'Orléans

Rapport n^o **RR-2017-07**

Textual Alignment in SPMD Programs

Frédéric Dabrowski

LIFO, Université d'Orléans

Abstract

We provide a formal definition of textual alignment of collective operations in a simple imperative language extended with minimal SPMD constructs. The definition relies on a instrumented operational semantics in which we record suitable informations about the execution flow. We prove that this property entails proper the absence of deadlocks and we claim that it provides an intuitive programming model.

1 Introduction

In the *single program, multiple data* (SPMD) programming model[2, 7], a collection of parallel processes executes the same program on different data. In contrast to models targeting *single instruction, multiple data* (SIMD)[8] architectures, where all processors execute the same instructions at the same pace, the SPMD model allows replicated processes to follow distinct flows of control. This model thus provides greater flexibility and can execute on *multiple instruction,multiple data* (MIMD) [8] architectures. A SPMD programming language can propose several interprocess cooperation means, of which the best known are direct remote memory access (DRMA) and message passing. Among them, *collective operations* play an important role. Broadcast, reduction and barriers are only a few examples of such operations. They offer a simple synchronization scheme : all processes of a group must suspend their execution, by performing the same collective operation, and only after that they can go further. However, behind the apparent simplicity of this model, the ability to execute distinct instruction streams without restriction exposes the programmers to the risk of deadlock. In this context, deadlocks related to collective operations may only occur when at least two processes execute distinct instruction streams.

To preclude this type of error, one can introduce a separation at the programming language level between global and parallel flows of control. The former produces a single instruction stream which every process follows. The latter produces multiple instruction streams free of collective operations[20, 19, 10]. It is noteworthy that this distinction had been already present in the early definition of the SPMD model, quoting [5] : 'the participating processes follow a different parallel flow of control, but all the processes follow the same global flow of control'. However, SPMD programs are most often written in general programming languages using libraries and, as a result, the two flows become mixed up (e.g. MPI[9], implementations of the BSP model[12, 24]). This simple observation highlights the need for tools capable of reconstructing the global control flow in SPMD programs.

Standard practices show that collective operations are most often *textually aligned*, which means that all processes synchronize on the same textual instruction. In other words, the use of collective operations is confined to the global control flow. Not only does this model simplify the programming of parallel programs, but it is also a prerequisite for some program analysis[16, 4, 3]. Despite its importance, and to the best of our knowledge, textual alignment in SPMD programs has not been the subject of an in-depth semantic study. Aiken and Gay introduced the concept of structural correctness for unrestricted programs. In structurally correct programs, parallel instruction streams produce the same sequence of collective operations. A type system is designed to determinate which branches can produce distinct behaviors and, in this case, compare the

numbers of operations[1, 11]. This work has been used for the design of the Titanium language [23, 6]. A later proposal introduced textually aligned barriers in Titanium by revisiting structural correctness. This proposal was finally replaced by a dynamic approach [17] after it has been observed that it was flawed [15, 17]. Recent work also considers dynamic approaches to the problem of textual alignment of collectives [18]. Barrier checking for arbitrary programs is also studied in [25]. More recently Jakobsson and al.[13] consider a static analysis based on the same principles than [1, 11] but dedicated to textually aligned programs. In this work, good programs are defined by means of replicated synchronization. This property is defined on top of the static analysis and thus depends on its precision. Intuitively, this analysis reject programs that perform collective operations under branches that may depend on process context. In some senses, it is an attempt to formalize the introduction of textual alignement in Titanium. Replicated synchronization is claimed to entail textual alignment, however, although the assumption sounds acceptable for any reasonable definitinon of textual alignement, the paper lacks a formal definition of the latter property.

This document is a first step toward a detailed semantic study of textual alignment. First, we introduce an operational semantics for a toy imperative language with minimal SPMD-style support for collective operations (Sections 2 and 3). Second, we rely on this semantics to define deadlocks (Section 3). Finally, we show how to instrument the semantics to provide a formal definition of textual alignment and prove that this property entails the absence of deadlock. These results lay down the basis for future static analyses ensuring the absence of deadlock as well as other kinds of analysis that depend on this assumption.

2 Language Definition

We consider a toy imperative language [22] extended by the addition of a global barrier instruction. Each process can obtain its id as well as the total number of processes. For the sake of simplicity, we leave aside communications.

2.1 Syntax

The syntax of the language is given below, where $\mathbb{X}$ stands for the set of program variables.

$$
\begin{array}{llllll}
\textit{expr} & \ni & e & ::= & x \mid n \mid e + e \mid e - e \mid e \times e \mid \texttt{nprocs} \mid \texttt{pid} & x \in \mathbb{X},\ n \in \mathbb{N} \\
\textit{bexpr} & \ni & b & ::= & \texttt{true} \mid \texttt{false} \mid e < e \mid e = e \mid b \ \textbf{or}\ b \mid b \ \textbf{and}\ b \mid !b & \\
\textit{stmt} & \ni & s & ::= & \texttt{skip} \mid x := e \mid s; s \mid \texttt{if}\,(b)\,\texttt{then}\ s\ \texttt{else}\ s\ \texttt{end} \mid \texttt{while}\,(b)\,\texttt{do}\ s\ \texttt{done} \mid \texttt{sync} & x \in \mathbb{X}
\end{array}
$$

The execution of a program consists in the parallel run of `p` copies of a given statement, `p` > 0 stands for the number of processes. The expression `nprocs` returns the constant `p`. The expression `pid` returns the identifier, ranging over $\{0, 1, \ldots, \texttt{p} - 1\}$, of the calling process. The execution proceeds in successive steps. At each step, all processes execute their current statement until they terminate normally or suspend their execution. A process suspends its becution by execution the `sync` instruction (the global barrier). The calling process is suspended untill all process excute this instruction. Other syntactic constructs have the usual meaning.

2.2 Semantics of Expressions

Arithmetic and boolean expressions respectively denote values ranging in $\mathbb{N}$ and $\mathbb{B} = \{\texttt{tt}, \texttt{ff}\}$. The evaluation of an expression depends on a store, σ which is a mapping from variables to natural numbers. A partial definition of their semantics is given in Figure 1 to illustrate how their meaning depend both on the number of processors and on the processor where evaluation takes place. Missing cases can be inferred in the obvious way.

$$[\![b]\!]^{\mathtt{p}}_i : (\mathbb{X} \to \mathbb{N}) \to \mathbb{B} \qquad i \in \{0, \ldots, \mathtt{p}-1\}$$

$$[\![e_1 < e_2]\!]^{\mathtt{p}}_i = \lambda\sigma. \begin{cases} \mathtt{tt} & \text{if } [\![e_1]\!]^{\mathtt{p}}_i\ \sigma < [\![e_2]\!]^{\mathtt{p}}_i\ \sigma \\ \mathtt{ff} & \text{oth.} \end{cases}$$

$$[\![e]\!]^{\mathtt{p}}_i : (\mathbb{X} \to \mathbb{N}) \to \mathbb{N} \qquad i \in \{0, \ldots, \mathtt{p}-1\}$$

$$\begin{array}{lcl} [\![\mathtt{nprocs}]\!]^{\mathtt{p}}_i & = & \lambda\sigma.\mathtt{p} \\ [\![\mathtt{pid}]\!]^{\mathtt{p}}_i & = & \lambda\sigma.i \\ [\![x]\!]^{\mathtt{p}}_i & = & \lambda\sigma.\sigma\ x \end{array}$$

Figure 1: Semantics of expressions

2.3 Dynamic Semantics

The semantics is given in the form of a two-level transition system, parameterized by the number of processes $\mathtt{p} > 0$. At the lower level, big step transitions[14] denote maximal sequential computations over local states, one for each process. The second level combines these computations into small step transitions [21]. The latter denote parallel computation steps over global states and is iteratively applied after each barrier. Local states and global states are defined below. We note $A^{\mathtt{p}}$ for the set of $\mathtt{p}$-tuples of elements of type A.

$$\begin{array}{lcll} \sigma & \in & store = \mathbb{X} \to \mathbb{N} & \text{(store)} \\ \gamma & \in & (stmt * store) \cup store & \text{(local state)} \\ \Gamma & \in & (stmt * store)^{\mathtt{p}} \cup store^{\mathtt{p}} & \text{(global state)} \end{array}$$

A store σ maps program variables to values ranging over natural numbers. We note $\sigma[x \mapsto v]$ the store mapping x to v and y to $\sigma(y)$ for all $y \neq x$. The sequential computations are defined using rules in the form $\mathtt{p}, i \vdash (s, \sigma) \to_\alpha \gamma$, where i is the id of the process at which the execution takes place and $\alpha \in \{\kappa, \iota\}$. We say that the computation is *suspended* if $\alpha = \iota$ and we say that it is *terminated* if $\alpha = \kappa$. Semantic rules are given in (Figure. 2) and their intuitive meaning is given below.

- The `skip` instruction terminates the execution, leaves the state unchanged and does not produce any continuation (rule $(skip)$).
- The $x := e$ instruction terminates the execution, updates the store by associating to x the value of e and does not produce any continuation (rule $(assign)$).
- The `sync` instruction suspends the execution, leaves the state unchanged and produces the continuation `skip` (rule $(sync)$).
- A sequence $s_1; s_2$ first executes s_1. If the execution of s_1 terminates then s_2 is executed (rule (seq_1)). If the execution of s_1 is suspended and produces the continuation s, the execution of the sequence is suspended and produces the continuation $s; s_2$ (rule (seq_2)).
- As usual, conditionals behave as the selected branch (rules (if_1) and (if_2)).
- Loops behave as their unfolding on each iteration (rule (wh_1)). On exit, the loop terminates the execution and does not produce any continuation (rule (wh_2)).

Remark 1. *Note that a statement produces a continuation if and only if its execution is suspended. The symbols α and κ are only there to improve readability.*

Example 1. *Consider the statement* `if` $(pid == 0)$ `then sync;` $x := 1$ `else skip end` *executed at*

$$\mathtt{p}, i \vdash \mathtt{skip}, \sigma \to_\kappa \sigma \quad (skip) \qquad \frac{[\![e]\!]^{\mathtt{p}}_i(\sigma) = v \qquad \sigma' = \sigma[x \mapsto v]}{\mathtt{p}, i \vdash x := e, \sigma \to_\kappa \sigma'} \quad (assign)$$

$$\mathtt{p}, i \vdash \mathtt{sync}, \sigma \to_\iota \mathtt{skip}, \sigma \quad (sync)$$

$$\frac{\mathtt{p}, i \vdash s_1, \sigma \to_\kappa \sigma' \qquad \mathtt{p}, i \vdash s_2, \sigma' \to_\alpha \gamma}{\mathtt{p}, i \vdash s_1; s_2, \sigma \to_\alpha \gamma} \quad (seq_1)$$

$$\frac{\mathtt{p}, i \vdash s_1, \sigma \to_\iota s_1', \sigma'}{\mathtt{p}, i \vdash s_1; s_2, \sigma \to_\iota s_1'; s_2, \sigma'} \quad (seq_2)$$

$$\frac{[\![b]\!]_i(\sigma) = \mathtt{tt} \qquad \mathtt{p}, i \vdash s_1, \sigma \to_\alpha \gamma}{\mathtt{p}, i \vdash \mathtt{if}\,(b)\,\mathtt{then}\ s_1\ \mathtt{else}\ s_2\ \mathtt{end}, \sigma \to_\alpha \gamma} \quad (if_1)$$

$$\frac{[\![b]\!]_i(\sigma) = \mathtt{ff} \qquad \mathtt{p}, i \vdash s_2, \sigma \to_\alpha \gamma}{\mathtt{p}, i \vdash \mathtt{if}\,(b)\,\mathtt{then}\ s_1\ \mathtt{else}\ s_2\ \mathtt{end}, \sigma \to_\alpha \gamma} \quad (if_2)$$

$$\frac{[\![b]\!]_i(\sigma) = \mathtt{tt} \qquad \mathtt{p}, i \vdash s; \mathtt{while}\,(b)\,\mathtt{do}\ s\ \mathtt{done}, \sigma \to_\alpha \gamma}{\mathtt{p}, i \vdash \mathtt{while}\,(b)\,\mathtt{do}\ s\ \mathtt{done}, \sigma \to_\alpha \gamma} \quad (wh_1)$$

$$\frac{[\![b]\!]_i(\sigma) = \mathtt{ff}}{\mathtt{p}, i \vdash \mathtt{while}\,(b)\,\mathtt{do}\ s\ \mathtt{done}, \sigma \to_\kappa \sigma} \quad (wh_2)$$

$$\frac{\forall i \in [0, \mathtt{p}-1].\ \mathtt{p}, i \vdash \pi_i(\Gamma) \to_\iota \pi_i(\Gamma')}{\mathtt{p} \vdash \Gamma \to_\iota \Gamma'} \quad (int) \qquad \frac{\forall i \in [0, \mathtt{p}-1].\ \mathtt{p}, i \vdash \pi_i(\Gamma) \to_\kappa \pi_i(\Gamma')}{\mathtt{p} \vdash \Gamma \to_\kappa \Gamma'} \quad (stop)$$

$$\frac{\Gamma \in (stmt * store)^{\mathtt{p}} \cup store^{\mathtt{p}}}{reachable_{\mathtt{p}}(\Gamma, \Gamma)} \qquad \frac{reachable_{\mathtt{p}}(\Gamma, \Gamma'') \qquad \mathtt{p} \vdash \Gamma'' \to_\alpha \Gamma'}{reachable_{\mathtt{p}}(\Gamma, \Gamma')}$$

Figure 2: Operational Semantics

processes 0 and 1.

$$\frac{2, 0 \vdash (\mathtt{sync}; x := 1, \sigma) \to_\iota (\mathtt{skip}; x := 1, \sigma)}{2, 0 \vdash (\mathtt{if}\,(pid == 0)\,\mathtt{then\ sync}; x := 1\ \mathtt{else\ skip\ end}, \sigma) \to_\iota (\mathtt{skip}; x := 1, \sigma)}$$

$$\frac{2, 1 \vdash (\mathtt{skip}, \sigma) \to_\iota (\sigma)}{2, 1 \vdash (\mathtt{if}\,(pid == 0)\,\mathtt{then\ sync}; x := 1\ \mathtt{else\ skip\ end}, \sigma) \to_\kappa \sigma}$$

Parallel computations are defined using rules in the form $\mathtt{p} \vdash ((s_0, \sigma_0), \ldots, (s_{\mathtt{p}-1}, \sigma_{\mathtt{p}-1})) \to \Gamma$. Rules are given in (Figure. 2). If all processes terminate then the global computation terminates (rule $(term)$). If all processes are suspended then the global computation is suspended (rule (int)). The last rules deafine reachability.

Definition 1. *An execution of the statement s for $\mathtt{p}$ processes with input $\Sigma \in (\mathbb{X} \to \mathbb{N})^{\mathtt{p}}$ is a maximal sequence $\Gamma_0, \Gamma_1, \ldots$ such that*

$$\Gamma_0 = (replicate_{\mathtt{p}}(s, \Sigma)) \ \textit{and}\ \forall i \geq 0. \exists \alpha. \Gamma_i \to_\alpha \Gamma_{i+1}$$

where $replicate_{\mathtt{p}}(s, (\sigma_0, \ldots, \sigma_{\mathtt{p}-1})) = ((s, \sigma_0), \ldots, (s, \sigma_{\mathtt{p}-1}))$.

Example 2. *Consider the statement* $s \triangleq$ `if` $(pid == 0)$ `then` $x := 1;$`sync else sync;`$x := 1$ `end` *and a possible execution for two processes with input* $(\lambda x.0, \lambda x.0)$.

$$\begin{array}{lll} ((s,[x \mapsto 0]),(s,[x \mapsto 0])) & \rightarrow_\iota & ((\texttt{skip},[x \mapsto 1]),(\texttt{skip};x := 1,[x \mapsto 0])) \\ & \rightarrow_\kappa & ([x \mapsto 1],[x \mapsto 1]) \end{array}$$

Definition 2. *Given* $\mathtt{p} > 0$ *and a global state* $\Gamma \in (stmt * store)^{\mathtt{P}}$*, a* deadlock *occurs in* Γ *iff* $deadlock_{\mathtt{p}}(\Gamma)$ *holds. A statement* s *is deadlock free if for all* $\mathtt{p} > 0$ *we have* $dlf_{\mathtt{p}}(s)$ *holds.*

$$deadlock_{\mathtt{p}}(\Gamma) = \exists i, j, \gamma_i, \gamma_j.((\mathtt{p}, j \vdash \pi_i(\Gamma) \rightarrow_\kappa \gamma_i) \wedge (\mathtt{p}, j \vdash \pi_j(\Gamma) \rightarrow_\iota \gamma_j))$$

$$dlf_{\mathtt{p}}(s) = \forall \Sigma, \Gamma. reachable_{\mathtt{p}}(replicate_{\mathtt{p}}(s, \Sigma), \Gamma) \Rightarrow \neg deadlock_{\mathtt{p}}(\Gamma)$$

Example 3. *The statement of Example 2 is deadlock free. On the opposite* $s =$ `if (pid ==` 0`) then sync else skip end` *is not as for* $\mathtt{p} = 2$ *and any* σ *we have*

$$2, 0 \vdash s \rightarrow_\iota (\texttt{skip}, \sigma) \qquad 2, 1 \vdash s \rightarrow_\kappa \sigma$$

3 Textual Alignment Property

Intuitively, a program is said to be textually aligned if, whenever a call to a `sync` instruction occurs, all processes jointly execute the same textual instance of the `sync` instruction. To provide a rigourous definition of theses notions we introduce an appropriate notion of execution path. It is defined in such a way that equality of paths leading to `sync` instructions defines equality over instances of instructions of the same kind. Before we go further, we discuss the four examples given in Figure 3. Although some cases are undoubtedly textually aligned, others require the use

---

`if (pid > 0) then sync else sync end` (1)

`if (pid < nprocs) then sync else sync end` (2)

$x := 0;$ `while` $(x <$ `nprocs`$)$ `do` $\{$`if` $(x =$ `pid`$)$ `then sync else skip end`$;\ x := x + 1\}$ `done` (3)

$x := 0;$ `while` $(x < 3)$ `do` $\{$`if` $(x = 2)$ `then sync else skip end`$;\ x := x + 1\}$ `done` (4)

process 0 $(x := 0; s_1, [x \mapsto \_]) \rightarrow_\iota (\texttt{skip}; x := x + 1; s_1, [x \mapsto 0]) \rightarrow_\kappa [x \mapsto 2]$
process 1 $(x := 0; s_1, [x \mapsto \_]) \rightarrow_\iota (\texttt{skip}; x := x + 1; s_1, [x \mapsto 1]) \rightarrow_\kappa [x \mapsto 2]$
where $s_1 =$ $x := 0;$ `while` $(x <$ `nprocs`$)$ `do` $\{$`if` $(x =$ `pid`$)$ `then sync else skip end`$;\ x := x + 1\}$ `done` (5)

process 0 $(x := 0; s_2, [x \mapsto \_]) \rightarrow_\iota (\texttt{skip}; x := x + 1; s_2, [x \mapsto 2]) \rightarrow_\kappa [x \mapsto 2]$
process 1 $(x := 0; s_2, [x \mapsto \_]) \rightarrow_\iota (\texttt{skip}; x := x + 1; s_2, [x \mapsto 2]) \rightarrow_\kappa [x \mapsto 2]$ (6)
where $s_2 =$ $x := 0;$ `while` $(x < 3)$ `do` $\{$`if` $(x = 2)$ `then sync else skip end`$;\ x := x + 1\}$ `done`

Figure 3: Examples

of a precise definition in order to be classified. The program 3.(1) is clearly not textually aligned as for $\mathtt{p} = 2$, processes execute sync instructions that occur at different program points. Obviously, program points equality is the least that might be expected for the definition of instances equality. On the opposite, it is obvious that the program 3.(2) should be considered texually aligned. In 3.(5) we provide an execution of the program 3.(3- where processes are separated to enhance readability. Although the behaviors of the two processes are similar, the two `sync` instructions are executed at different iterations of the loop. Our definition will classify this program as not textually aligned. On the opposite 3.(4 is textually aligned. n execution is given in 3.(6) Intuitively, two instances of a sync instruction are the same if they are reach through the same path in the program unfolding. Paths are formally defined in Definition 3.

Definition 3. *A path δ is a pair (n, w) where w is a, possibly empty, list of elements of $\{l, r\} \times \mathbb{N}$. The empty list is noted ε and $(l, k) \cdot w$ builds a new list by adding the element (l, k) to the top of w*

In Figure 4 we reformulate our semantics to incorporate path calculation. Each time a process terminates or is suspended, the path corresponding to the current computation step is exposed. It is immediate that the semantics of the previous section can be recovered by removing annotations. Intuitively, a path records the number of conditionals crossed so far, at the current nesting level, as well as the history of choices for the current nesting level (l for left branch and r for right branch). Unsurprisingly, loop iterations are treated as nested conditionals.

- The path associated with the execution of a `skip` instruction is $(0, \varepsilon)$ as no conditional statement is executed. The same holds for assignment and the `sync` instruction.
- If the execution of s_1 terminates then the number of conditional in the execution of $s_1; s_2$ is the total number of conditionals in s_1 and s_2 and the history of choices for the current nesting is that of s_2. If the execution of s_1 is interrupted then the path is the one generated by the execution of s_1.
- When executing a conditional, it is the only conditional of the current nesting leve. The path is that of the execution of the taken branch combined with the correct tag (l or r) and the current counter (rules (if_1) and (if_2)).
- The path generated by a loop is recursively defined as the path generated by its unfolding.

Parallel executions (rules (int) and $(stop)$) and reachable states are defined as in the previous section

Definition 4. *Barriers of a program s are textually aligned if $aligned_{\mathtt{p}}(s)$ holds for all $\mathtt{p} > 0$, where*

$$\begin{aligned} aligned_{\mathtt{p}}(s) \quad = \quad & \forall \Sigma, \Gamma. reachable_{\mathtt{p}}(replicate_{\mathtt{p}}(s, \Sigma), \Gamma) \Rightarrow \\ & \forall i, j < \mathtt{p}. (\mathtt{p}, i \vdash \pi_i(\Gamma) \xrightarrow{\delta_i} \gamma_i \wedge \mathtt{p}, j \vdash \pi_j(\Gamma) \xrightarrow{\delta_j} \gamma_j) \Rightarrow \delta_i = \delta_j \end{aligned}$$

In Figure 5, annotated executions (7), (8), (9) and (10) respectively illustrate the definition for examples (1) (2) (3) and (4) of Figure 3. The four examples are classified as expected.

Before to prove that textual alignment prevents deadlocks we introduce a few intermediary results. Lemma 1 shows some simple properties of execution paths. Property (1) states that whenever two executions of the same statement terminate, they cross the same number of top level conditionals. Property (2) states that whenever two executions of the same statement have different status, the terminated process has crossed at least as many top level conditionals as the suspended process. Obviously, if one execution is suspended and the other is not then the synchronization occurs under a conditional and the other execution cannot terminate before the conditional is closed. Property (3) states that whenever an execution of a statement is suspended at the top level, any other execution of the same statement is suspended. Property (4) states that the history of choices remains empty as long as no conditional is crossed.

Lemma 1. *The following properties hold.*

1. *if* $\mathtt{p}, \mathtt{i}_1 \vdash (s, \sigma_1) \xrightarrow{(n_1, w_1)}_{\kappa} \Gamma_1$ *and* $\mathtt{p}, \mathtt{i}_1 \vdash (s, \sigma_2) \xrightarrow{(n_2, w_2)}_{\kappa} \Gamma_2$ *then* $n_2 = n_1$.
2. *if* $\mathtt{p}, \mathtt{i}_1 \vdash (s, \sigma_1) \xrightarrow{(n_1, w_1)}_{\kappa} \Gamma_1$ *and* $\mathtt{p}, \mathtt{i}_1 \vdash (s, \sigma_2) \xrightarrow{(n_2, w_2)}_{\iota} \Gamma_2$ *then* $n_2 \leq n_1$.
3. $\neg(\mathtt{p}, \mathtt{i}_1 \vdash (s, \sigma_1) \xrightarrow{(m, w)}_{\kappa} \Gamma_1 \wedge \mathtt{p}, \mathtt{i}_1 \vdash (s, \sigma_2) \xrightarrow{(n, \varepsilon)}_{\iota} \Gamma_2)$
4. *if* $\mathtt{p}, \mathtt{i} \vdash (s, \sigma) \xrightarrow{(0, w)}_{\iota} \Gamma$ *then* $w = \varepsilon$.

$$\mathtt{p}, i \vdash \mathtt{skip}, \sigma \xrightarrow{(0,\varepsilon)}_{\kappa} \sigma \quad (skip) \qquad \frac{[\![e]\!]^{\mathtt{p}}_i = v \qquad \sigma' = \sigma[x \mapsto v]}{\mathtt{p}, i \vdash x := e, \sigma \xrightarrow{(0,\varepsilon)}_{\kappa} \sigma} \quad (assign)$$

$$\mathtt{p}, i \vdash \mathtt{sync}, \sigma \xrightarrow{(0,\varepsilon)}_{\iota} \mathtt{skip}, \sigma \quad (sync)$$

$$\frac{\mathtt{p}, i \vdash s_1, \sigma \xrightarrow{(k_1,w_1)}_{\kappa} \sigma' \qquad \mathtt{p}, i \vdash s_2, \sigma' \xrightarrow{(k_2,w_2)}_{\alpha} \gamma}{\mathtt{p}, i \vdash s_1; s_2, \sigma \xrightarrow{(k_1+k_2,w_2)}_{\alpha} \gamma} \quad (seq_1)$$

$$\frac{\mathtt{p}, i \vdash s_1, \sigma \xrightarrow{(k,w)}_{\iota} (s_1', \sigma')}{\mathtt{p}, i \vdash s_1; s_2, \sigma \xrightarrow{(k,w)}_{\iota} (s_1'; s_2, \sigma')} \quad (seq_2)$$

$$\frac{[\![b]\!]^{\mathtt{p}}_i(\sigma) = \mathtt{tt} \qquad \mathtt{p}, i \vdash s_1, \sigma \xrightarrow{(k,w)}_{\alpha} \gamma}{\mathtt{p}, i \vdash \mathtt{if}\,(b)\,\mathtt{then}\ s_1\ \mathtt{else}\ s_2\ \mathtt{end}, \sigma \xrightarrow{(1,(l,k)\cdot w)}_{\alpha} \gamma} \quad (if_1)$$

$$\frac{[\![b]\!]^{\mathtt{p}}_i(\sigma) = \mathtt{ff} \qquad \mathtt{p}, i \vdash s_2, \sigma \xrightarrow{(k,w)}_{\alpha} \gamma}{\mathtt{p}, i \vdash \mathtt{if}\,(b)\,\mathtt{then}\ s_1\ \mathtt{else}\ s_2\ \mathtt{end}, \sigma \xrightarrow{(1,(r,k)\cdot w)}_{\alpha} \gamma} \quad (if_2)$$

$$\frac{[\![b]\!]^{\mathtt{p}}_i(\sigma) = \mathtt{tt} \qquad \mathtt{p}, i \vdash s; \mathtt{while}\,(b)\,\mathtt{do}\ s\ \mathtt{done}, \sigma \xrightarrow{(k,w)}_{\alpha} \gamma}{\mathtt{p}, i \vdash \mathtt{while}\,(b)\,\mathtt{do}\ s\ \mathtt{done}, \sigma \xrightarrow{(1,(l,k)\cdot w)}_{\alpha} \gamma} \quad (wh_1)$$

$$\frac{[\![b]\!]^{\mathtt{p}}_i(\sigma) = \mathtt{ff}}{\mathtt{p}, i \vdash \mathtt{while}\,(b)\,\mathtt{do}\ s\ \mathtt{done}, \sigma \xrightarrow{(0,\varepsilon)}_{\alpha} \sigma} \quad (wh_2)$$

$$\frac{\forall i \in [0, \mathtt{p}-1].\mathtt{p}, i \vdash \pi_i(\Gamma) \xrightarrow{\delta_i}_{\iota} \pi_i(\Gamma'')}{\mathtt{p} \vdash \Gamma \to_{\iota} \Gamma'} \quad (int) \qquad \frac{\forall i \in [0, \mathtt{p}-1].\mathtt{p}, i \vdash \pi_i(T) \xrightarrow{\delta_i}_{\kappa} \pi_i(\Gamma')}{\mathtt{p} \vdash \Gamma \to_{\kappa} \Gamma'} \quad (stop)$$

$$\frac{\Gamma \in (stmt * store)^{\mathtt{p}} \cup store^{\mathtt{p}}}{reachable_{\mathtt{p}}(\Gamma, \Gamma)} \qquad \frac{reachable_{\mathtt{p}}(\Gamma, \Gamma'') \qquad \mathtt{p} \vdash \Gamma'' \to_{\alpha} \Gamma'}{reachable_{\mathtt{p}}(\Gamma, \Gamma')}$$

Figure 4: Instrumented Semantics

$$\mathtt{p}, 0 \vdash \mathtt{if}\,(pid > 0)\,\mathtt{then\ sync\ else\ sync\ end} \xrightarrow{(1,[(r,0)])}_\iota (\mathtt{skip}, \sigma) \xrightarrow{(0,\varepsilon)}_\kappa \sigma \tag{7}$$
$$\mathtt{p}, 1 \vdash \mathtt{if}\,(pid > 0)\,\mathtt{then\ sync\ else\ sync\ end} \xrightarrow{(1,[(l,0)])}_\iota (\mathtt{skip}, \sigma) \xrightarrow{(0,\varepsilon)}_\kappa \sigma$$

$$\mathtt{p}, 0 \vdash \mathtt{if}\,(pid < nprocs)\,\mathtt{then\ sync\ else\ sync\ end} \xrightarrow{(1,[(l,0)])}_\iota (\mathtt{skip}, \sigma) \xrightarrow{(0,\varepsilon)}_\kappa \sigma \tag{8}$$
$$\mathtt{p}, 1 \vdash \mathtt{if}\,(pid < nprocs)\,\mathtt{then\ sync\ else\ sync\ end} \xrightarrow{(1,[(l,0)])}_\iota (\mathtt{skip}, \sigma) \xrightarrow{(0,\varepsilon)}_\kappa \sigma$$

$$\mathtt{p}, 0 \vdash (x := 0; s_1, [x \mapsto \_]) \xrightarrow{(1,[(l,1),(l,0)])}_\iota (\mathtt{skip}; x := x + 1; s_1, [x \mapsto 0]) \xrightarrow{(1,[(l,1)])}_\kappa [x \mapsto 1]$$
$$\mathtt{p}, 1 \vdash (x := 0; s_1, [x \mapsto \_]) \xrightarrow{(1,[(l,2),(l,1),(l,0)])}_\iota (\mathtt{skip}; x := x + 1; s_1, [x \mapsto 1]) \xrightarrow{(0,\varepsilon)}_\kappa [x \mapsto 2]$$
where $s_1 = x := 0;\ \mathtt{while}\,(x < \mathtt{nprocs})\,\mathtt{do}\ \{\mathtt{if}\,(x = \mathtt{pid})\,\mathtt{then\ sync\ else\ skip\ end};\ x := x + 1\}\ \mathtt{done}$ (9)

$$\mathtt{p}, 0 \vdash (x := 0; s_2, [x \mapsto \_]) \xrightarrow{(1,[(l,2),(l,2)(l,1)(l,0)])}_\iota (\mathtt{skip}; x := x + 1; s_2, [x \mapsto 2]) \xrightarrow{(0,\varepsilon])}_\kappa [x \mapsto 3]$$
$$\mathtt{p}, 1 \vdash (x := 0; s_2, [x \mapsto \_]) \xrightarrow{(1,[(l,2),(l,2)(l,1)(l,0)])}_\iota (\mathtt{skip}; x := x + 1; s_2, [x \mapsto 2]) \xrightarrow{(0,\varepsilon)}_\kappa [x \mapsto 3]$$
where $s_2 = x := 0;\ \mathtt{while}\,(x < 3)\,\mathtt{do}\ \{\mathtt{if}\,(x = 2)\,\mathtt{then\ sync\ else\ skip\ end};\ x := x + 1\}\ \mathtt{done}$ (10)

Figure 5: Examples revisited

Proof. By structural induction on s. The sequence case in the proof of (2) uses (1) and simple arithmetic. All other cases are either trivial or immediate by induction hypothesis. □

Proposition 1 states that if two executions of the same statement follow the same path and are suspended they produce the same continuation. Proposition 2 states that any two executions of the same statement following the same path end with the same status. Lemma generalize equality of continuations to all reachable states.

Proposition 1. *If* $\mathtt{p}, \mathtt{i}_1 \vdash (s, \sigma_1) \xrightarrow{\delta}_\iota (s_1, \sigma_1')$ *and* $\mathtt{p}, \mathtt{i}_2 \vdash (s, \sigma_2) \xrightarrow{\delta}_\iota (s_2, \sigma_2')$ *then* $s_1 = s_2$

Proof. By induction on the first derivation tree. Suppose that $s = s_1; s_2$ and $\delta = (n, w)$. We consider two cases. First suppose that we have (a) and (b) where $n_1 + n_2 = n$. If (c) and (d), where $n_3 + n_4 = n$, then by (a), (c) and (1) we have $n_3 = n_1$ and then $n_2 = n_4$. We conclude by applying the induction hypothesis to (b) and (d). If (e) where $n_3 = n$. By (a), (e) and (2) we have $n_1 + n_2 \leq n_1$ and then $n_2 = 0$ and $n_3 = n_1$. By (b) and (4) we have $w = \varepsilon$. By (a), (e) and (3) we have a contradicton.

$$(a)\quad \mathtt{p}, i_1 \vdash (s_1, \sigma_1) \xrightarrow{(n_1,w_1)}_\kappa \sigma_1' \qquad (b)\quad \mathtt{p}, i_1 \vdash (s_2, \sigma_1') \xrightarrow{(n_2,w)}_\iota \gamma_1$$
$$(c)\quad \mathtt{p}, i_2 \vdash (s_1, \sigma_2) \xrightarrow{(n_3,w_3)}_\kappa \sigma_2' \qquad (d)\quad \mathtt{p}, i_2 \vdash (s_2, \sigma_2') \xrightarrow{(n_4,w)}_\iota \gamma_2$$
$$(e)\quad \mathtt{p}, i_2 \vdash (s_1, \sigma_2) \xrightarrow{(n_3,w)}_\iota \gamma_2$$

Second, suppose that we have (f) where $n_1 = n$. If (g) and (h) then following the same reasonment as in the previous case we have a contradiction. If (i) the conclusion is immediate by induction hypothesis.

$$(f)\quad \mathtt{p}, i_1 \vdash (s_1, \sigma_1) \xrightarrow{(n,w)}_\iota \sigma_1'$$
$$(g)\quad \mathtt{p}, i_2 \vdash (s_1, \sigma_2) \xrightarrow{(n_3,w_3)}_\kappa \sigma_2' \qquad (h)\quad \mathtt{p}, i_2 \vdash (s_2, \sigma_2') \xrightarrow{(n_4,w)}_\iota \gamma_2$$
$$(i)\quad \mathtt{p}, i_2 \vdash (s_1, \sigma_2) \xrightarrow{(n,w)}_\iota \gamma_2$$

Other cases are either trivial or immediate by induction hypothesis. □

Lemma 2. *Let* s *be a textually aligned program. Suppose that for* $\mathtt{p} > 0$*, an input* Σ *and a global state* $\Gamma \in (stmt \times store)^{\mathtt{p}}$ *we have* $reachable_{\mathtt{p}}(replicate_{\mathtt{p}}(s, \Sigma), \Gamma)$*. For all* $i, j < \mathtt{p}$ *there exists* s'*,* σ_i *and* σ_j *such that* $\pi_i(\Gamma) = (s', \sigma_i)$ *and* $\pi_j(\Gamma) = (s', \sigma_j)$*.*

Proof. The proof is by induction on the proof tree of $\mathit{reachable}_{\mathtt{p}}(\mathit{replicate}_{\mathtt{p}}(s,\Sigma),\Gamma)$. For the base case, the result is immediate for by definition of $\mathit{replicate}_{\mathtt{p}}$. For the induction step, suppose that (1) $\mathit{reachable}_{\mathtt{p}}(\mathit{replicate}_{\mathtt{p}}(s,\Sigma),\Gamma')$ and (2) $\mathtt{p} \vdash \Gamma' \rightarrow_{\alpha} \Gamma$ where $\Gamma \in (\mathit{stmt} \times \mathit{store})^{\mathtt{P}}$. By induction hypothesis, there exists s_0, σ_i, σ_j such that $\pi_i(\Gamma') = (s_0,\sigma_i)$ and $\pi_j(\Gamma') = (s_0,\sigma_j)$. By (2) we know that there exists s_1, s_2, σ_1 and σ_2 such that $\mathtt{p}, i_1 \vdash (s_0,\sigma_i) \xrightarrow{\delta_1}_{\iota} (s_1,\sigma_1)$ and $\mathtt{p}, i_2 \vdash (s_0,\sigma_j) \xrightarrow{\delta_2}_{\iota} (s_2,\sigma_2)$. By definition of $\mathit{aligned}_{\mathtt{p}}$ it comes $\delta_1 = \delta_2$ and then $s_1 = s_2$ by Proposition 1. □

Proposition 2. *If* $\mathtt{p}, \mathtt{i}_1 \vdash (s,\sigma_1) \xrightarrow{\delta}_{\alpha_1} \Gamma_1$ *and* $\mathtt{p}, \mathtt{i}_2 \vdash (s,\sigma_2) \xrightarrow{\delta}_{\alpha_2} \Gamma_2$ *then* $\alpha_1 = \alpha_2$

Proof. Similar to the proof of 1. □

Theorem 1. *For all statement s, if s is aligned then s is deadlock free.*

Proof. Suppose that there exists a reachable state where a deadlock occurs. By definition of $\mathit{aligned}_{\mathtt{p}}$ all local computations follow the same path. Then, by Lemma 2 all components contain the same statement. We conclude with Proposition 2 which contradicts the existence of a deadlock. □

Lemma 2 is usefull to prove our main result but we believe it also provides a nice argument in favor of the chosen definition of paths. Indeed textually aligned barriers should always lead to the same continuation. We expect the fact that every computation step starts with the same statement at every process should simplify program analysis.

4 Conclusion

In this document, we have proposed an operational semantics for a minimal SPMD language with collective operations. We have also defined when deadlocks occur in this semantics and proposed a formal definition of textual alignment. Finally we have prove that the latter property guarantees the absence of deadlocks. Future work include deeper semantic investigation as well as the design of a tool based on abstract interpretation to check for textual alignment. We conjecture that adding procedures to the language can be achieved easily by recording appropriate informations about frames in paths. No difficulties are expected to add pointers. We also plan to study the implication of textual alignment on the design of other static analysis. In particular may-happen in parallel analysis should benefit from this assumption. Another line of study would be to extend the notion of textual alignement to arbitrary control points and to study the possible implications in the field of compiler optimisation by providing a better understanding of the notion of global path.

References

[1] Alexander Aiken and David Gay. Barrier inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 342–354, New York, NY, USA, 1998. ACM.

[2] Larbey F. Auguin M. Opsila: an advanced simd for numerical analysis and signal processing. In *Microcomputers: developments in industry, business, and education, Ninth EUROMICRO Symposium on Microprocessing and Microprogramming*, pages 311–318, Madrid, 1983.

[3] Prasanth Chatarasi, Jun Shirako, Martin Kong, and Vivek Sarkar. *An Extended Polyhedral Model for SPMD Programs and Its Use in Static Data Race Detection*, pages 106–120. Springer International Publishing, Cham, 2017.

[4] C. Chen, W. Huo, L. Li, X. Feng, and K. Xing. Can we make it faster? efficient may-happen-in-parallel analysis revisited. In *2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 59–64, Dec 2012.

[5] Frederica Darema. *SPMD Computational Model*, pages 1933–1943. Springer US, Boston, MA, 2011.

[6] Hilfinger P. N. (editor), Dan Bonachea, David Gay, Susan Graham, Ben Liblit, Geoff Pike, and Katherine Yelick. Titanium Language Reference Manual, Version 1.16.8. Technical Report UCB//CSD-04-1163x, Computer Science, UC Berkeley, 2004.

[7] Darema F. Spmd model: past, present and future, recent advances in parallel virtual machine and message passing interface. In *Proceedings of the 8th European PVM/MPI Users' Group Meeting, Lecture Notes in Computer Science*, Santorini/Thera, Greece, 2001.

[8] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, September 1972.

[9] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.1.

[10] F. Gava and F. Loulergue. A static analysis for bulk synchronous parallel ml to avoid parallel nesting. *Future Generation Computer Systems*, 21(5):665 – 671, 2005. Parallel computing technologies.

[11] D. Gay. *Barrier Inference*. PhD thesis, University of California, Berkeley, 1998.

[12] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. Bsplib: The bsp programming library. *Parallel Comput.*, 24(14):1947–1980, December 1998.

[13] Arvid Jakobsson, Frédéric Dabrowski, Wadoud Bousdira, Frédéric Loulergue, and Gaetan Hains. Replicated synchronization for imperative {BSP} programs. *Procedia Computer Science*, 108:535 – 544, 2017. International Conference on Computational Science, {ICCS} 2017, 12-14 June 2017, Zurich, Switzerland.

[14] G. Kahn. *Natural semantics*, pages 22–39. Springer Berlin Heidelberg, Berlin, Heidelberg, 1987.

[15] A. Kamil. Problems with the titanium type system for alignment of collectives. unpublished note, 2006.

[16] Amir Kamil and Katherine Yelick. Concurrency analysis for parallel programs with textually aligned barriers. In *Proceedings of the 18th International Conference on Languages and Compilers for Parallel Computing*, LCPC'05, pages 185–199, Berlin, Heidelberg, 2006. Springer-Verlag.

[17] Amir Kamil and Katherine Yelick. *Enforcing Textual Alignment of Collectives Using Dynamic Checks*, pages 368–382. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[18] Andreas Knüpfer, Tobias Hilbrich, Joachim Protze, and Joseph Schuchart. *Dynamic Analysis to Support Program Development with the Textually Aligned Property for OpenSHMEM Collectives*, pages 105–118. Springer International Publishing, Cham, 2015.

[19] Frédéric Loulergue, Frédéric Gava, and David Billiet. Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction. volume 3515 of *LNCS*, pages 1046–1054. Springer, 2005.

[20] Frédéric Loulergue and Gaétan Hains. *Functional parallel programming with explicit processes: Beyond SPMD*, pages 530–537. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[21] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[22] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, Cambridge, MA, USA, 1993.

[23] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: a high-performance java dialect. *Concurrency: Practice and Experience*, 10(11-13):825–836, 1998.

[24] A. N. Yzelman, R. H. Bisseling, D. Roose, and K. Meerbergen. Multicorebsp for c: A high-performance library for shared-memory parallel programming. *Int. J. Parallel Program.*, 42(4):619–642, August 2014.

[25] Yuan Zhang and Evelyn Duesterwald. Barrier matching for programs with textually unaligned barriers. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 194–204, New York, NY, USA, 2007. ACM.