Transforming non textually aligned SPMD programs into textually aligned SPMD programs by using rewriting rules

Wadoud Bousdira

Transforming non textually aligned SPMD programs into textually aligned SPMD programs by using rewriting rules

Wadoud Bousdira
LIFO, University of Orléans, France
wadoud.bousdira@univ-orleans.fr

***Abstract*—The problem of analyzing parallel programs that access shared memory and use barrier synchronization is known to be hard to study. For a special case of those programs with minimal SPMD (Single Program Multiple Data) constructs, a formal definition of textually aligned barriers with an operational semantics has been proposed in previous work. Then, the textual alignement of the synchronization barriers that is defined prevents deadlocks. However, the textual alignement property is not verified by all SPMD programs. We propose a set of transformation rules using rewriting techniques which allows to turn a non-textually aligned program to be textually aligned. So, we can benefit of a simple static analysis for deadlock detection. We show that the rewrite rules form a terminating confluent system and we prove that the transformation rules preserve the semantics of the programs.**

Keywords : SPMD programs; synchronization; textual alignement; rewriting rules

I. Introduction

Since its introduction, static source code analysis has had a mixed reputation with development teams due to long analysis times, excessive noise or an unacceptable rate of false-positive results. Excessive false-positive results are the main reason why many source code analysis products quickly become shelfware after a few uses. Despite early shortcomings, the promise of static analysis remained of interest to developers because the technology offers the ability to find bugs before software is run, improving code quality and dramatically accelerating the availability of new applications.

With applications assuming more critical functions for business and industry, the consequence of defects in the field now mandates that software meet specific quality standards prior to release. Applying static analysis to software, the automated review of code prior to run-time with the intention of identifying defects, was an obvious solution to this fundamental challenge of ensuring code quality.

In this paper, we focus on a static analysis applied to scalable parallellism in a so called BSP model. Many parallel programs are written in SPMD (Single Program Multiple Data) style, i.e. by running the same sequential program on all processes. Typically, SPMD-style programs have a barrier synchronization primitive that can be used to partition the program into a sequence of parallel phases. When a thread reaches a barrier statement it cannot proceed until all other threads have arrived at the barrier statement. Barriers are textually aligned if all threads must reach the same textual barrier statement before they can proceed. A barrier synchronization error occurs if a thread bypasses a barrier, leaving the remaining threads stalled.

Usually, SPMD programs are written using a standard communication library (e.g. MPI [16], implementations of the BSP model [7, 17, 28]). However, in this model, there are no constraints on the placement of barrier statements in the program. Barrier statements may be textually unaligned making it more difficult for programmers to understand the synchronization structure of the program and, thus, easier to write programs with synchronization errors. Textually unaligned barriers also hinder concurrency analysis [1, 14, 21, 27] because understanding which barrier statements form a common synchronization point is a prerequisite to analyzing the ordering constraints imposed by the program. Some concurrency analyses therefore require barriers to be named or textually aligned[3].

An extensive amount of work on static analysis of SPMD programs has been done. In the synchronization point of view, the most relevant previous work on verifying barrier synchronization is proposed by Aiken and Gay [4]. The authors propose a system for SPMD programs that verifies their synchronization pattern, based on structural correctness and single-valued expressions. This analysis has been extended inter-procedurally and with barrier matching [29]. In [3], the authors exploit that barriers in Titanium (a Java-dialect) are statically guaranteed to be textually aligned. They construct a May-Happen-in-Parallel relation and use it for data-race detection. Jeremiassen and Eggers developed a static analysis for barrier synchronization for SPMD programs with non-textual barriers and used the information to reduce false sharing on cache-coherent machines [25]. Their analysis does not take advantage of barrier alignment or single-valued expressions, so it is not as precise as in [8].

However, in these papers devoted to a correct synchronisation of SPMD programs, there is no definition of any semantics underlying the corresponding programs. About semantics, a number of formal semantics for functional and imperative BSP languages has been proposed. Loulergue formalizes BSML [20], Tesson and al. formalize BSPlib [26] and Gava and al. formalize Paderborn's BSPlib [15]. As far as

we know, the only contribution that proposes a denotational semantics which gives meaning exactly to programs with textual alignment is in [8]. The author proposes a formal definition of textual alignment for a small language with minimal SPMD support, based on an operational semantics. The textual alignment property entails the absence of deadlocks and provides an intuitive programming model that makes it easier to perform program analysis and program optimization. Our work is based on this semantics that is explicitly designed to model the features of BSPLIB most relevant to synchronization. Our contribution in this paper concerns non-textually aligned SPMD programs. We propose a transformation strategy based on rewriting technics to modify these programs to make them textually aligned. Of course, the transformation that we propose does not guarantee to turn all non-textually aligned SPMD programs to textually aligned ones. In fact, only if the constructions of the program can be captured by the transformation pattern, then we can obtain an equivalent textually aligned program. As far as we know, there is no work about transforming a non-textually aligned SPMD program by rewriting techniques to make it textually aligned. In these last decades, the evolution of rewriting concepts is opening new fields of application, and we show in this paper that parallel programming is one of them.

The paper is organized as follows : Section 2 defines the syntax of the imperative language we use to write SPMD programs. Section 3 is devoted to the textual alignment property of SPMD programs and gives a precise intuition how it can prevent deadlocks. Then for non-textually aligned programs, we present a pre-processing based on constant propagation analysis. In Section 4, the second step of transformation is introduced by using simplification axioms and rewriting technics. Then we conlude in Section 5.

II. Language

We consider an extension of a basic imperative language supporting collective operations in the SPMD style. For the sake of simplicity, we focus on global synchronization barriers and we omit communications. The syntax of the language is given below, where $\mathbb{X}$ stands for the set of program variables and $\mathbb{N}$ is the set of integers. The following grammars define respectively the set `expr` of arithmetic expressions, the set `bexpr` of boolean expressions and the set `stmt` of program statements :

$$
\begin{array}{rcl}
e & ::= & x \mid n \mid e + e \mid e - e \mid e * e \mid \texttt{nprocs} \mid \texttt{pid} \\
b & ::= & \texttt{true} \mid \texttt{false} \mid e < e \mid e = e \mid b \texttt{ or } b \mid b \texttt{ and } b \\
 & \mid & ! \, b \mid b = b \\
s & ::= & \texttt{skip} \mid x := e \mid \texttt{if } (b) \texttt{ then } s \texttt{ else } s \\
 & \mid & \texttt{while } (b) \texttt{ do } s \mid \texttt{sync}
\end{array}
$$

Where $x \in \mathbb{X}$ and $n \in \mathbb{N}$.

A program is a single statement that is executed in a parallel way by p processors. Each processor executes a copy of the program. $p > 0$ indicates the number of processors. The specific element `nprocs` in the language evaluates to p. The expression `pid` represents the identifier of the current process, ranging over $\mathbb{P} = \{0, 1, \ldots, p-1\}$. The instruction `sync` requests a global synchronization barrier. As a collective operation, it needs to be performed by all processes. For example, the statement `if` $(x < y)$ `then sync else skip` is correct only if executed in a context where the condition evaluates to the same value at every process. When a process executes a `sync` instruction, it interrupts, waiting all other processes reach the synchronization barrier. If all processes interrupt, the synchronization succeeds and communications between processes occur exchanging data (we do not detail this step). Then each process continues with the next instruction. If all processes terminate normally then the program terminates. Finally, a deadlock situation occurs if some processes interrupt while other terminate normally.

In the SPMD programming model, collective operations rely on a global synchronization scheme, that is all processes must execute the same operation before they can proceed further. Therefore, a deadlock is detected because at least two distinct processes execute distinct instruction streams. To ensure that there is no deadlock situation, a distinction between separation global and local flow of control is made in the language. On the one hand, the former produces a single instruction stream among processes. On the other hand, the latter can produce distinct instruction streams but they must be free of collective operations. However, SPMD programs are usually written using a standard communication library (e.g. mpi[7], implementations of the bsp model[9, 17],...) where the two flows are merged. This raises the need for program analysis tracking the global flow of control of SPMD programs in order to validate the use of collective operations.

In [8], an operational semantics is given in term of transitions between global states which are tuples of local states. A local state is either a non-terminal state composed of a statement and a substitution $\sigma : \mathbb{X} \to \mathbb{N}$ or a terminal state only composed of σ. A global state Γ is a tuple of p local states or Err. Err denotes an error state, considered as a terminal state. The author defines a small step semantics and a big step semantics. The operational semantics records suitable information about the execution flow, and then distinguish three cases 1) if all processors terminate normally then the global computation terminates normally ; 2) if all processors interrupt then the global computation interrupts ; 3) if some processes terminate with different status, then an error occurs.

III. Textual alignment property

Standard practices show that collective operations are most frequently textually aligned [18], which means that all processes synchronize on the same textual instruction. The operational semantics of SPMD programs is formalized by transitions between global states which are tuples of local states. A local state γ describes the behavior of local computations. Local rules state that executing s at location i over p with local memory σ produce the new state σ. A local rule is either a non-terminal state $(s, \sigma) \in \texttt{stmt} \times (\mathbb{X} \to \mathbb{N})$, or a terminal state $\sigma \in \mathbb{X} \to \mathbb{N}$. Local rules are of the form $p,\ i \vdash (s, \sigma) \to_\alpha \gamma$ where $\alpha \in \{\kappa, \iota\}$. The process interrupts

if $\alpha = \kappa$ which occurs if and only if the step terminates with a `sync` instruction. In this case, γ is a pair (s, σ) where s is the continuation for the next step. Otherwise, if $\alpha = \iota$, the process terminates. In this case, γ is a terminal state σ. A global state Γ is an element of $(\texttt{stmt} \times (\mathbb{X} \to \mathbb{N}))^p \cup (\mathbb{X} \to \mathbb{N})^p \cup \{Err\}$ where p is the number of processes, Err denotes an error state. It corresponds to a situation in which some processes terminate with different status. A global state cannot comprise both terminal and non-terminal states. For example, to give an intuition of the operational semantics, we show one of the two local rules of the sequence (seq_1) and the global rule $(stop)$ which illustrates the termination of the program:

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

$$(stop) \quad \frac{\forall i \in \mathbb{P}\ p, i \vdash \pi_i(T) \to_\kappa \pi_i(\theta)}{p \vdash T \to \theta}$$

Definition 1: A program P is textually aligned if for every call to a collective operation, all processes execute the same textual instance.

Textual instances of instructions are represented by paths associated with the execution. One can refer to [8] for more details.

(i) `if (pid > 0) then sync else sync`

(ii) `if (pid < nprocs) then sync else sync`

(iii)
```
x := 0; while (x < nprocs) do
    (if (x = pid) then sync else skip;
    x := x + 1)
```

(iv) `if (pid %2 = 0) then sync else skip`

(v)
```
x := 0 ; if (x > 0) then skip
 else
  if (pid < nprocs) then
    if (pid%2 = 0) then sync
    else sync
  else skip;
```

Figure 1: Examples of SPMD programs

For instance, `sync` instructions are not textually aligned in (i) nor (iii) nor (iv) nor (v) while they are in (ii). In (i) the process 0 executes an instance of the `sync` instruction different from the one performed by all other processes. In (iii), at iteration i, only the process i executes a `sync` instruction, then each process executes a different instance of `sync`. In (iv), some processes execute the `sync` instruction while others do not, according to the `pid` identifier. (v) is not textually aligned because of the condition $(pid\%2 = 0)$ which leads processes do not execute the same instance of `sync`. (v) illustrates why a correct synchronization can be subtil. Indeed, all processes execute `else` branch of the first conditional instruction provided the value of the variable x (which is replicated i.e. each process has a variable x local to the process) is the same at this point of the program.

Identifying textual instances of instructions during the execution is necessary to guarantee the textual alignement property. For this, an annotated version of the operational semantics is used to record the path associated with an execution of a processor in [8] and the author proves that programs verifying the textual alignment property are well-synchronized. An immediate consequence is that textual alignment of collective operations prevents deadlocks.

For every program for which the analysis fails to state that it is well-synchronized, we propose a transformation routine to modify the code of the program in order to make it textually aligned. As we will see later, the transformation focuses on conditional instructions of the program which are the crucial points that we observe if all processes execute the same branch of the instruction. First this routine needs a prior labelling treatment `Labelling`.

1) `Labelling`: we detect all the conditional instructions of the program that contain at least a `sync` instruction. Then a unique label, (different form all the others) $l \in \mathbb{L}$ is attached to each of the conditions. Note that labels are natural numbers and that $\mathbb{L} \subset \mathbb{N}$ is a finite set. For instance, for (i) we get `if` $[\texttt{(pid} > 0)]^0$ `then sync else sync`. For (v) we obtain the following tagged code:

```
x := 0 ; if [(x > 0)]^0 then skip
 else
  if [(pid < nprocs)]^1 then
    if [(pid%2 = 0)]^2 then sync
    else sync
  else skip;
```

Figure 2: `Labelling` task

2) `CP Analysis` : this second part consists in applying the Constant Propagation Analysis over all the program, and then focusing on labelled conditions and using some specific axioms such that `pid` $<$ `nprocs` $=$ `true`, and `pid` $<$ `0` $=$ `false` etc... to symbolically evaluate the conditional expressions.
The Constant Propagation Analysis is a classical static analysis which maintains information about what constant, if any, a variable has at each point. It supposes that each variable of the program takes a value in a domain $\mathcal{D}$ extended by two values $\top$ and $\bot$. $\top$ is used to indicate that a variable is not a constant, $\bot$ means that we do not yet know anything and all other elements indicate that the value is that particular constant. The analysis defines a lattice on $\mathcal{D} \cup \{\top, \bot\}$ and the analysis process contains the following stages: 1) construct a control flow graph (CFG) of the program, 2) associate transfer functions with the edges of the CFG. The transfer function of an edge reflects the semantics as its source node. 3) at each node (program point), we maintain the values of the program's variables at this point. We initialize those to $\bot$. 4) iterate until the values of the variables stabilize. One can refer to [23] for more details about Constant Propagation analysis.
Once the analysis is finished, given the set of labels $\mathbb{L}$ of `Labelling`, we define an abstract interpretation $\mathcal{A} : \mathbb{L} \to \{\mathit{TT}, \mathit{FF}, \top\} = \mathit{Bool}^\sharp$ such that $\mathcal{A}(l) = \mathit{TT}$

(respectively *FF*) if in the condition b labelled by l, there are only variables associated to constant values and if the condition is evaluated to tt by replacing variables by the values, where tt is the boolean value of the constant `true` (respectively to *ff*, where *ff* is the boolean value of the constant `false`). In all other cases, $\mathcal{A}(l) = \top$. After this, for the conditions labelled by $\top$, we try to use specific axioms such that $\texttt{pid} < \texttt{nprocs} = \texttt{true}$, and $\texttt{pid} < 0 =$ `false` etc... to modify the value to *TT* or *FF* or to keep it to $\top$.
Thereafter, we introduce three notations to represent the conditional symbol `if` of the language, and in every labelled condition, we replace the symbol `if` by $if_{\mathcal{A}(l)}$. Note that a boolean condition b can be evaluated to tt or *ff* whether it is pid-dependent or not. For example, `CP Analysis` returns the code of Figure 3 for (v):

```
x := 0 ; if_FF [(x > 0)]^0 then skip
 else
  if_TT [(pid < nprocs)]^1 then
    if_⊤ [(pid%2 = 0)]^2 then sync
    else sync
  else skip;
```

Figure 3: Result of `CP Analysis`

Proposition 1: Let P be an SPMD program and let P' be the program obtained from P by performing `Labelling` and `PC Analysis`; then P and P' are semantically equivalent, that is, both programs have the same execution trace.

Proof: `Labelling` does not modify the semantics of P since it just detects the `if` statements in P that contain a `sync` instruction and attaches a different unique label to each different condition. Furthermore, since the Constant Propagation analysis preserves the semantics of the program, the translation does not modify the semantics of the program. ■

IV. Rewriting rules

In this section, we propose a set of axioms which express the equality between portions of code that contain conditional instructions. We aim to define a method that simplifies a conditional instruction to another one using this set of axioms. These transformations of an SPMD program which contains non-textually aligned synchronization instructions aim to obtain an equivalent SPMD program with textually aligned instructions. The set of transformation axioms is given in term of equalities. Then, to have a deterministic transformation method, instead of using equalities, we will use rather rewrite rules after running the Knuth-Bendix (KB) completion on the set of transformation axioms. Running the KB completion, we propose a semantically equivalent set of terminating confluent rewrite rules, ensuring in this way that if we rewrite an SPMD program by this set of rules, we get an SPMD program that is semantically equivalent. One can refer to [18] for a survey of equational and rewriting theories.

Given two (denumerable) sets $\mathcal{X}$ of variable symbols and $\mathcal{F} = \cup_{n \geqslant 0} \mathcal{F}_n$ of function symbols, the set of first order-terms $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is the smallest set containing $\mathcal{X}$ such that $f(t_1, \ldots, t_n)$ is in $\mathcal{T}(\mathcal{F}, \mathcal{X})$ whenever $f \in \mathcal{F}_n$ and $t_i \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ for $i \in [1 \ldots n]$. Each symbol f in $\mathcal{F}$ has an arity which is the index of the set $\mathcal{F}_n$ it belongs to. A position within a term t is represented as a sequence ω of positive integers describing the path from the root of t to the root of the subterm at that position, denoted by $t_{|\omega}$. We note $\mathcal{D}(t)$ the set of positions in t. As usuallly defined, a substitution on $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is a mapping of $\mathcal{T}(\mathcal{F}, \mathcal{X})$ on itself, written out as $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ when there are only finitely many variables x_i not mapped to themselves.

Equality is a binary relation between terms meaning that the terms are identical, in the sense that replacement of one by the other in an expression does not change its value. Given a set E of axioms, we write $s \longleftrightarrow_E t$ if $s_{|\omega} = \sigma(l)$ and $t = s[\sigma(r)]_\omega$ for some position ω in $\mathcal{D}(s)$, substitution σ and equality $l = r$ (or $r = l$) in E. The term $s[\sigma(r)]_\omega$ denotes the term s which contains the subterm $\sigma(r)$ at position ω. The equation $s = t$ is deduced from E iff $s \overset{*}{\longleftrightarrow}_E t$ where the relation $\overset{*}{\longleftrightarrow}_E$ is the reflexive transitive closure of $\longleftrightarrow_E$.

With the need to give an operational (implementable) version of equality, while avoiding the halting problem intrinsically related to the symmetrical nature of equality, the central idea of rewriting is to impose directionnality in the use of equalities.

A rewrite rule is an ordered pair of terms denoted $l \to r$. The terms l and r are respectively called the left-hand side and the right-hand side of the rule. A rule $l \to r$ is applied by replacing an instance l by the same instance of r, but never the converse, contrary to equalities. A rewrite system R induces a binary relation on terms called the rewriting relation. From a logical point of view, the deduction rules are similar to equational logic without the Symmetry rule ($\frac{t=t'}{t'=t}$) and with the rewrite relation instead of equality. One can refer to [6, 11, 13] for more details on rewriting techniques.

In order to orient equations to rewrite rules, many orderings and termination proof techniques have been developed, surveys are developed in [2, 11, 12]. Many of these orderings have the subterm property denoted by $\trianglelefteq$: $s \trianglelefteq t$ iff s is a subterm of t. s is a proper subterm of t if $s \trianglelefteq t$ and $s \neq t$. In the formalization of the term algebra, we later use the lexicographic path ordering (*lpo*) $>_{lpo}$ defined by :

Definition 2: Let $>_\mathcal{F}$ be a precedence on $\mathcal{F}$. $s = f(s_1, \ldots, s_n) >_{lpo} t = g(t_1, \ldots, t_m)$ if

1) $f = g$ and $\{s_1 ; \ldots ; s_n\} >_{lpo}^{lex} \{t_1 ; \ldots ; t_n\}$ and $\forall j \in 1, \ldots, n$, $s >_{lpo} t_j$
2) $f >_\mathcal{F} g$ and $\forall j \in 1, \ldots, m$, $s >_{lpo} t_j$
3) $\exists i \in 1, \ldots, n$ such that either $s_i >_{lpo} t$ or $s_i = t$.

Proposition 2: *lpo* is a simplification ordering [11].

A strong termination property is often needed to normalize a rewrite system. In general, termination of a rewriting system is undecidable. Some approaches aim to relate termination with orderings (reduction or polynomials orderings, simplification orderings [10]). The termination of rewriting can be proved by

just comparing left and right-hand sides of rules: $l > r$ for each rule of R.

When a function is computed with rewrite rules, the property required for the rewrite system is therefore the uniqueness of the normal form for any term. Uniqueness of the normal form is implied by adding to termination another property called confluence. A rewrite system is confluent when two rewrite sequences beginning from the same term can always be extended to end with the same term. Although undecidable in general, confluence is decidable for terminating finite rewrite systems. Assuming termination, confluence is equivalent to local confluence, itself equivalent to the convergence of critical pairs [22]. All these properties are expressed as follows:

$$(1) \quad \overset{*}{\longleftrightarrow}_E = \overset{*}{\longrightarrow}_R \circ \overset{*}{\longleftarrow}_R$$
$$(2) \quad \overset{*}{\longleftarrow}_R \circ \overset{*}{\longrightarrow}_R \subseteq \overset{*}{\longrightarrow}_R \circ \overset{*}{\longleftarrow}_R$$
$$(3) \quad \longleftarrow_R \circ \longrightarrow_R \subseteq \overset{*}{\longrightarrow}_R \circ \overset{*}{\longleftarrow}_R$$

The Church-Rosser property (1) states the relation between replacement of equals by equals and rewriting. (2) (respec. (3)) expresses that the relation $\rightarrow_R$ is confluent (respec. locally confluent).

Lemma 1: [9]

- If $\rightarrow_R$ is terminating, then $\rightarrow_R$ is confluent iff $\rightarrow_R$ is locally confluent.
- If $\rightarrow_R$ is terminating and locally confluent, then $\rightarrow_R$ is Church-Rosser. In other words, to prove the equality $t =_E t'$, it is equivalent to rewrite t and t' to the same term.

The local confluence of a rewriting system R can be checked on special patterns computed from pairs of rules called critical pairs. Critical pairs characterize minimal conflicts that can happen when two rewrite rules apply to a same term and overlap each other. Then the term can be rewritten in two different ways that may or may not have a common normal form. If the relation $\rightarrow_R$ is terminating, the KB completion procedure aims to compute critical pairs between rules of R. When critical pairs are not convergent, they are added to R after they have been oriented. Of course, adding new rules implies computing new critical pairs and the process is recursively applied. A critical pair is an equation obtained by superposition of left-hand sides of two rewrite rules. During the completion process, equations are simplified by rewriting, and tautologies, i.e. equations of the form $s = s$ are removed. A completion process has three possible outcomes: either it terminates, or fails on an unorientable equality, or diverges, that is, generates infinitely many new rules. A critical pair $\sigma(g[r]_{|\omega}) = \sigma(d)$ between rule $l \rightarrow r$ and $g \rightarrow d$ is computed if there is a minimal unifier σ of $g_{|\omega}$ and l.

Lemma 2: [9] Let E be a set of equations and let R be the set of rewrite rules obtained from E by a KB completion process. Suppose that R is a terminating confluent rewrite system. Then the equational theory induced by E is semantically equivalent to the semantics induced by R.

In the literature, we say that R is a complete rewrite system if R is confluent and terminating.

Now in the SPMD program application, let us define the first order terms on which the rewrite relation will be applied.

Definition 3: The set of first order terms $\mathcal{T}(\mathcal{F}, X)$ is the set of terms built from the set of function symbols $\mathcal{F} = \{if_{TT}, if_{FF}, if_{\top}, seq, sync\} \cup \mathbb{L}$. if_{TT}, if_{FF}, $if_{\top}$ have an arity equal to 3, the arity of seq is equal to 2, $sync$ and all labels are constants (arity = 0). The variables in X are all other symbols.

Consider an SPMD program P such that on each piece of code in P the head instruction of which is a conditional statement, we perform `Labelling` and `CP Analysis`. On each part of code resulting from this procedure, we infer a first order term the following way:

- from $if_{TT}[(b)]^l$ `then` s `else` s', we build the term $if_{TT}(l, s, s')$
- from $if_{FF}[(b)]^l$ `then` s `else` s', we build the term $if_{FF}(l, s, s')$
- from $if_{\top}[(b)]^l$ `then` s `else` s', we build the term $if_{\top}(l, s, s')$
- from $s\,;\, s'$ contained below if_{TT}, if_{FF} or $if_{\top}$, we build the term $seq(s, s')$.

This encoding consists in capturing in a simple way the point of the original SPMD program in which the synchronization has to be checked. For example, the program (v) of Figure 3 corresponds to the term of Figure 4:

$$if_{FF}(0, skip, if_{TT}(1, if_{\top}(2, sync, sync), skip))$$

Figure 4: First order term of (v)

In the following of the paper, transformation process by rewriting will be executed on each term coded in this way.

In Figure 5, we give the set of axioms E which we use to simplify terms. Axioms (1) and (2) express the natural semantics of conditional statements. Axiom (3) states that whatever the value of b, (tt or ff) the program executes the instruction s. Axioms (4) and (5) ride up the condition labelled by the same label in both branches of a conditional statement if these branches contain the same statements in the same case (case tt or case ff). Axiom (8) is a generalization of (6) and (7). These three axioms allow to take out `sync` instruction of `if` statement. So, whatever the branch executed by the processes, all of them execute the same textual instance of `sync`.

The intuition of these axioms is the same as the small step semantics defined in [8]. The first order term that is built captures for each process, the semantics of its piece of

program.

(1) $if_{TT}(l,\, s,\, s') = s$
(2) $if_{FF}(l,\, s,\, s') = s'$
(3) $if_{\top}(l,\, s,\, s) = s$
(4) $if_{\top}(l',\, if_{\top}(l,\, s,\, s'),\, if_{\top}(l,\, s,\, s")) = if_{\top}(l,\, s,\, if_{\top}(l',\, s',\, s"))$
(5) $if_{\top}(l',\, if_{\top}(l,\, s,\, s'),\, if_{\top}(l,\, s",\, s')) = if_{\top}(l,\, if_{\top}(l',\, s,\, s"),\, s')$
(6) $if_{\top}(l,\, seq(s,\, sync),\, seq(s',\, sync)) = seq(if_{\top}(l,\, s,\, s'),\, sync)$
(7) $if_{\top}(l,\, seq(sync,\, s),\, seq(sync,\, s')) = seq(sync,\, if_{\top}(l,\, s,\, s'))$
(8) $if_{\top}(l,\, seq(seq(s_1,\, sync),\, s_2),\, seq(seq(s_3,\, sync),\, s_4)) = seq(seq(if_{\top}(l,\, s_1,\, s_3),\, sync),\, if_{\top}(l,\, s_2,\, s_4))$

Figure 5: Transformation axioms.

In order to orient the equations of E from left to right, we use a lpo ordering :

Definition 4: Let $>_{\mathbb{N}}$ be the ordering on natural numbers, and let $>_{\mathcal{F}}$ be a precedence on $\mathcal{F}$ such that $if_{\top} >_{\mathcal{F}} seq$, and such that $\forall l,\, l' \in \mathbb{N},\, l >_{\mathcal{F}} l'$ iff $l >_{\mathbb{N}} l'$ then $>_{lpo}$ is a lexicographic path ordering on $\mathcal{T}(\mathcal{F},\, X)$.

In this definition, we just adapted the definition of lexicographic path ordering[24] by integrating the $>_{\mathbb{N}}$ ordering.

Proposition 3: If we orient the axioms of E from left to right, using $>_{lpo}$ defined as above, we get a set R of rewrite rules such that for every rule $l \to r$ in R, $l >_{lpo} r$.

Proof: Clearly, (1), (2), (3) are oriented from left to right by a subterm property. Remember that the lexicographic path ordering which is a simplification ordering enjoys the subterm property.

To orient (4), let us note α the term $if_{\top}(l',\, if_{\top}(l,\, s,\, s'),\, if_{\top}(l,\, s,\, s"))$;

$\alpha >_{lpo} if_{\top}(l,\, s,\, if_{\top}(l',\, s',\, s"))$ iff

a) $\{l' ;\, if_{\top}(l,\, s,\, s');\, if_{\top}(l,\, s,\, s")\} >^{lex}_{lpo} \{l ;\, s;\, if_{\top}(l',\, s',\, s")\}$
b) $\alpha >_{lpo} l$
c) $\alpha >_{lpo} s$ and
d) $\alpha >_{lpo} if_{\top}(l',\, s',\, s")$.

(a) holds according $>_{lpo}$ by using $>_{\mathbb{N}}$ on labels l and l'. Indeed, we are sure that l and l' are two different labels and that they correspond to two natural numbers, one of them greater than the other. If we affect an integer to each if statement in a decreasing order each time a statement is encountered in the program, then the outermost label is greater than the innermost and this corresponds to the direction of our transformation. (b) and (c) are valid thanks to the subterm property. To check (d), we have

(c1) $\{l' ;\, if_{\top}(l,\, s,\, s') ;\, if_{\top}(l,\, s,\, s")\} >^{lex}_{lpo} \{l' ;\, s' ;\, s"\}$ which holds by the subterm property,

(c2) $\alpha >_{lpo} l', \alpha >_{lpo} s'$ and $\alpha >_{lpo} s"$. (c2) holds by the subterm property.

For similar reasons, (5) is oriented from left to right.

Axiom (6) is oriented from left to right since $if_{\top} >_{\mathcal{F}} seq$ and because the left term of (6) is bigger than $if_{\top}(s,\, s',\, l)$ and $sync$ according to $>_{lpo}$. We justify the orientation of (7) and (8) for the same reasons. ■

Let us call R the set of axioms of E all oriented from left to right ; note that R is a finite set of rewrite rules. Indeed, the set $\mathbb{L}$ of labels is finite since it depends on the number of conditional statements of the program.

In order to use the rules of R in a deterministic way, and to preserve the expressiveness of the axioms, the following consists in running the KB completion procedure on R, in order to get a nœtherian confluent system. This is achieved by computing critical pairs between all the rewrite rules and by adding those that are not trivial as equations ; then the algorithm orients them and again, it computes new critical pairs and so on, until the state of the rewrite system is stable. The execution of KB completion on R provides 4 non-trivial critical pairs. Let us explain in detail how one of them is computed :

1) The left-hand side of rule (4) at position 1 can be unified with the left-hand side of rule (3) at position ϵ and the substitution σ such that $\sigma(s) = \sigma(s')$. Therefore the term $if_{\top}(l',\, if_{\top}(l,\, s,\, s),\, if_{\top}(l,\, s,\, s"))$ can be rewritten by (3) at position 1 to the term $if_{\top}(l',\, s,\, if_{\top}(l,\, s,\, s"))$. On the other hand, the same term $if_{\top}(l',\, if_{\top}(l,\, s,\, s),\, if_{\top}(l,\, s,\, s"))$ can be rewritten by (4) at position ϵ to $if_{\top}(l,\, s,\, if_{\top}(l',\, s,\, s"))$. This yields pc1 :
$if_{\top}(l',\, s,\, if_{\top}(l,\, s,\, s")) = if_{\top}(l,\, s,\, if_{\top}(l',\, s,\, s"))$
which can be oriented according $>_{lpo}$ by using $>_{\mathbb{N}}$ on labels l and l' by using the fact $l' >_{\mathbb{N}} l$ as for the orientation of (4).
2) pc2 computed between (3) and (4) at position 2 :
$if_{\top}(l',\, if_{\top}(l,\, s,\, s'),\, s) = if_{\top}(l,\, s,\, if_{\top}(l',\, s',\, s))$
which can be oriented from left to right according to $>_{lpo}$.
3) pc3 is computed by unification of rules (3) and (5) at position 1 :
$if_{\top}(l',\, s,\, if_{\top}(l,\, s",\, s)) = if_{\top}(l,\, if_{\top}(l',\, s,\, s"),\, s)$
pc3 is oriented from left to right.
4) Finally, pc4 is computed from unification of rules (3) and (5) at position 2 :
$if_{\top}(l',\, if_{\top}(l,\, s,\, s'),\, s') = if_{\top}(l,\, if_{\top}(l',\, s,\, s'),\, s')$
which is oriented from left to right according to that $l' >_{\mathbb{N}} l$.

Finally, for any $l,\, l' \in \mathbb{N}$, such that $l' >_{\mathbb{N}} l$, we get the complete set of rewrite rules R presented in figure 6. It is not so difficult to show that R is complete. However, it would be interesting to automatically check this property using a rewriting tool such that for example Saigawa [5].

With this complete system R, and given a first order term t derived from a piece of code of a textually unaligned SMPD program P, we reduce t to its normal form by R. Then we get possibly another term t' (depending if there are possibilities to rewrite t) such that the code corresponding to t' under some conditions is textually aligned. There is no difficulty to infer

the SMPD program from the new term t'.

(1) $if_{TT}(l,\ s,\ s') \rightarrow s$
(2) $if_{FF}(l,\ s,\ s') \rightarrow s'$
(3) $if_{\top}(l,\ s,\ s) \rightarrow s$
(4) $if_{\top}(l',\ if_{\top}(l,\ s,\ s'),\ if_{\top}(l,\ s,\ s")) \rightarrow$
$if_{\top}(l,\ s,\ if_{\top}(l',\ s',\ s"))$
(5) $if_{\top}(l',\ if_{\top}(l,\ s,\ s'),\ if_{\top}(l,\ s",\ s')) \rightarrow$
$if_{\top}(l,\ if_{\top}(l',\ s,\ s"),\ s')$
(6) $if_{\top}(l,\ seq(s,\ sync),\ seq(s',\ sync)) \rightarrow$
$seq(if_{\top}(l,\ s,\ s'),\ sync)$
(7) $if_{\top}(l,\ seq(sync,\ s),\ seq(sync,\ s')) \rightarrow$
$seq(sync,\ if_{\top}(l,\ s,\ s'))$
(8) $if_{\top}(l,\ seq(seq(s_1,\ sync),\ s_2),\ seq(seq(s_3,\ sync),\ s_4))$
$\rightarrow seq(seq(if_{\top}(l,\ s_1,\ s_3),\ sync),\ if_{\top}(l,\ s_2,\ s_4))$
(9) $if_{\top}(l',\ s,\ if_{\top}(l,\ s,\ s")) \rightarrow if_{\top}(l,\ s,\ if_{\top}(l',\ s,\ s"))$
(10) $if_{\top}(l',\ if_{\top}(l,\ s,\ s'),\ s) \rightarrow if_{\top}(l,\ s,\ if_{\top}(l',\ s',\ s))$
(11) $if_{\top}(l',\ s,\ if_{\top}(l,\ s",\ s)) \rightarrow if_{\top}(l,\ if_{\top}(l',\ s,\ s"),\ s)$
(12) $if_{\top}(l',\ if_{\top}(l,\ s,\ s'),\ s') \rightarrow if_{\top}(l,\ if_{\top}(l',\ s,\ s'),\ s')$

Figure 6: A complete rewriting system.

Let us continue the illustration of our method with the term t of Figure 4. Using the complete rewrite system of Figure 6, we obtain the normal form of t:

$if_{FF}(0,\ skip,\ if_{TT}(1,\ if_{\top}(2,\ sync,\ sync),\ skip)) \rightarrow_R$
$if_{FF}(0,\ skip,\ if_{TT}(1,\ sync,\ skip)) \rightarrow_R$
$if_{FF}(0,\ skip,\ sync) \rightarrow_R sync$

Figure 7: Normal form of term of (v)

Proposition 4: Let P be a textually unaligned SMPD program and let P' be the SPMD program obtained by performing `Labelling` and `PC Analysis` and the transformation by rewriting over all pieces of code of P that contain conditional statements with `sync` statements. Then P and P' are semantically equivalent.

Proof: let P" be the SPMD program obtained after running `Labelling` and `PC Analysis`. We have shown in Proposition 1, that P and P" are semantically equivalent. Consider now a piece of code in P" that is a conditional statement. The way we infer the first order term $t = if_{\mathcal{A}(l)}(l,\ s,\ s')$ from the instruction $if_{\mathcal{A}(l)}[(b)]^l$ `then` s `else` s' expresses a natural semantics of boolean conditions except that the condition does not appear explicitly in the term since it is represented by its label. Reducing t by our complete rewriting system R yields a normal form of t that is semantically equivalent to t in the axiomatic theory defined by R. After that, we can write the program P' as the original program P where all pieces of conditional code are replaced by equivalent code obtained from normal forms. Then P and P' are semantically equivalent. ■

Theorem 1: Let P be a textually unaligned SPMD program ; if every piece of code in P that contains a `sync` instruction in a conditional statement is translated to a first order term t which contains only if_{TT} or if_{FF} function symbols, then after reducing t by R and reconstituting the new equivalent SPMD program P', then P' is textually aligned.

Proof: Reduction by the rewrite system R for rules (1) and (2) removes the branching in the code. Then, all processes follow the same execution path, and then synchronize together at the same point. Then P' is textually aligned. ■

Theorem 2: Let P be a textually unaligned SPMD program ; let us consider every piece of code c in P that is a conditional instruction which contains a `sync` instruction. If each c in P verifies : let t be the first order term corresponding to c and let t' be the R-normal form of t such that t' does not contain any sub-term of the form $if_{\top}(\ldots,\ sync,\ldots)$ (i.e. under $if_{\top}$ there is no $sync$) then P' obtained from t' is textually aligned.

Proof: Rules (6), (7) and (8) allow to rewrite a term such that the symbol function $sync$ is removed from the term the root of which is equal to $if_{\top}$. If there is no $sync$ under $if_{\top}$ in all normal forms obtained by reduction, then all processes use the same textual instance of synchronization, and then P' is textually aligned. ■

V. Conclusion and Future Work

The work we have proposed is at the crossroads of two domains, the parallel programming and the term rewriting. Based on a denotational semantics of SPMD programs formally defined in [8], we present a complete method to transform textually unaligned programs into textually aligned programs. Our method is based on static analysis of the program and only some pieces of code are targeted and then changed if necessary. In the near future, we are interested to deal with `while` statements. We think that it will be not so trivial to extend our work to the loops, it will be probably necessary to integrate the notion of repeated labels.

This work is related to code refactoring. However, we present a program transformation method which is correct and automatically implementable.

In this context of semantics, the static analysis of replicate synchronization has been implemented as a plug-in to the Frama-C analysis platform. Initial experimentation suggests that the analysis performs well, with results in seconds for programs ranging up to hundreds of lines. For more details, see [19]. It would be interesting to combine the implementation of the analysis with the transformation method by rewriting for SPMD programs which are not textually aligned. The new code in the transformed program could be a way to learn the user how to write more easily textually aligned code with collective operations. The coding of the transformation par rewriting is in progress.

References

[1] Knüpfer A., Hilbrich T., Protze J., and Schuchart J. Dynamic analyses to support program development with the textually aligned property for openshmem collectives. In *Second workshop of OPENSHMEM and Related Technologies*, volume 9397, pages 105–118. Springer, 2015. 1

[2] Middeldorp A. and Zantema H. Simple termination of rewrite systems. *Theoretical Computer Science*, 9:127–158, 1997. 4

[3] K. Yelick A. Kamil. Concurrency analysis for parallel programs with textually aligned barriers. In *International Workshop on Languages and Compilers for Parallel Computing*. ACM, 2005. 1

[4] Alexander Aiken and David Gay. Barrier inference. In *25th ACM SIGPLAN Symposium on Principles of Programming (POPL)*, pages 342–354. ACM, 1998. 1

[5] T. Aoto, N. Hirokawa, A. Middeldorp, J. Nagele, N. Nishida, K. Shintani, and H. Zankl. Confluence competition 2018. In *3rd International Conference on Formal Structures for Cumputation and Deduction*, pages 32:1–32:5, 2018. 6

[6] J. Avenhaus and Madlener K. Term rewriting and equational reasoning. In R. B. Bnaerji, editor, *Formal technics in Artificial Intelligence*, pages 1–43. Elsiever Sciences Publisher, 1990. 4

[7] Rob H. Bisseling. *Parallel Scientific Computation: A Structured Approach using BSP and MPI*. Oxford University Press, 2004. 1

[8] Frédéric Dabrowski. A denotational semantics of textually aligned SPMD Programs. *Journal of Logical and Algebraic Methods in Programming*, 2019. 1, 2, 3, 5, 7

[9] Knuth D.E. and Bendix P.B. Simple word problems in universal algebras. *Computational Problems in Abstract Algebra*, pages 263–297, 1970. 5

[10] N. Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982. 4

[11] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation 3*, pages 69–116, 1987. 4

[12] Contejean E., Marché C., Tomàs A.P., and Urbain X. Mechanically proving termination using polynomial interpretations. *Journal of Automated Reasoning*, 34(4):325–363, 2005. 4

[13] Baader F. and Nipkow T. Term rewriting systems and all that. *Cambridge Universit Press*, 1998. 4

[14] M.J. Flynn. Some computer organizations and their effectiveness. In *Trans. on Computers*, volume C-21(9), pages 948–960. IEEE, 1972. 1

[15] F. Gava and J. Fortin. Formal Semantics of a Subset of the Paderborn's BSPlib. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT)*. IEEE, 2008. 1

[16] I. Grudenic and N. Bogunovic. Modeling and Verification of MPI Based Distributed Software. In B. Mohr, J. Larsson Träff, J. Worringen, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI User's Group Meeting*, LNCS 4192, pages 123–132. Springer, 2006. 1

[17] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob Bisseling. BSPlib: The BSP Programming Library. *Parallel Computing*, 24:1947–1980, 1998. 1

[18] G. Huet and D. Oppen. Equations and rewrite rules: A survey. In R. Book, editor, *Formal Languages: Perspectives and Open Problems*, 1980. 4

[19] A. Jackobsson, F. Dabrowski, W. Bousdira, F. Loulergue, and G. Hains. Replicated Synchronisation for Imperative BSP Programs. In Proœdia Computer Science, editor, *International Conference on Computational Science (ICCS)*, 77, Zurich, Switzerland, 2017. Elsiever. 7

[20] Frédéric Loulergue. BSλ_p: Functional BSP Programs on Enumerated Vectors. In J. Kazuki, editor, *International Symposium on High Performance Computing*, number 1940 in LNCS, pages 355–363. Springer, October 2000. 1

[21] Frédéric Loulergue, Frédéric Gava, and David Billiet. Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction. In *International Conference on Computational Science (ICCS)*, volume 3515 of *LNCS*, pages 1046–1054. Springer, 2005. 1

[22] Newman M-H. On theories with a combinatorial definition of equivalence. *Annals of Math*, pages 223–243, 1942. 5

[23] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2005. 3

[24] J. Steinbach. Extensions ans comparison of simplification orderings. In *Conference on Rewriting Techniques and Applications*, number 355 in LNCS, pages 434–448. Springer Verlag, 1989. 6

[25] Jeremiassen T. and Eggers S. Static analysis of barrier synchronization in explicitly parallel programs. In Parallel Architectures and Compilation Techniques, editors, *IFIP*, pages 171–180, Montreal, Canada, August 1994. 1

[26] Julien Tesson and Frédéric Loulergue. Formal Semantics for the DRMA Programming Style Subset of the BSPlib Library. In J. Weglarz, R. Wyrzykowski, and B. Szymanski, editors, *Seventh International Conference on Parallel Processing and Applied Mathematics (PPAM 2007), Workshop on Language-Based Parallel Programming Models*, number 4967 in LNCS, pages 1122–1129. Springer, 2008. 1

[27] G. Winskel. *The Formal Semantics of Programming Languages*. Foundations of Computing Series. MIT Press, 1993. 1

[28] A.N. Yzelman, R.H. Bisseling, D. Roose, and K. Meerbergen. MulticoreBSP for C: A High-Performance Library for Shared-Memory Parallel Programming. *International Journal of Parallel Programming*, pages 1–24, 2013. 1

[29] Y. Zhang and E. Duesterwaki. Barrier matching for programs with textually unaligned barriers. In *12th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, pages 194–204. ACM, 2007. 1