International Journal of Parallel Programming (2021) 49:802–819
https://doi.org/10.1007/s10766-021-00710-5

On Single-Valuedness in Textually Aligned SPMD Programs

Frédéric Dabrowski[1]

Received: 13 November 2020 / Accepted: 17 March 2021 / Published online: 4 May 2021

Abstract
Single-valuedness is a property of an expression occurring in a SPMD program and states that concomitant evaluations of this expression lead to the same value at all processes. Although widely used, this property still lacks a formal definition, which is necessary to tackle the subtleties of the notion of concomitance. First, we propose such a definition in which the states of all processes can be compared when they reach *textually aligned* program points. These program points, of which all processes execute exactly the same textual instances, act as *logical global synchronization barriers*. Single-valuedness is then defined at these program points. Secondly, we show how textual alignment and single-valuedness can be used to ensure proper use of Direct Remote Memory Access (DRMA) in BSP programs.

Keywords SPMD · BSP · Collective primitives · textual alignment · single-valuedness · formal semantics

1 Introduction

In the SPMD programming model [1, 2], a collection of parallel processes executes a *Single Program on Multiple Data*. Unlike the *Single Instruction, Multiple Data* (SIMD) [3] model, where all processors execute the same instructions at the same pace, the SPMD model allows replicated processes to follow distinct flows of control. Several communication means may be proposed by SPMD programming languages (MPI[4], OpenMP[5], BSPlib[6], ...). The most popular are *Direct Remote Memory Access* (DRMA) and message passing. In both cases, *collective operations* (or *collectives* for short) play a major role. They expose a simple synchronization scheme: all processes execute the same sequence of collectives, performing a global synchronization for some of them. Broadcasts, reductions and global barriers are examples of collectives. However, behind the

✉ Frédéric Dabrowski
frederic.dabrowski@univ-orleans.fr

[1] Univ. Orléans, INSA Centre,Val de Loire, LIFO EA 4022, Orléans, France

apparent simplicity of the model, the ability to execute distinct instruction streams with no restriction may lead to programming errors.

To preclude these types of errors, one can introduce a strict 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 collectives [7–9]. It is noteworthy that this distinction had already been present in the early definition of the SPMD model, quoting [10]: "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, the result being that the two flows are mixed up (e.g., MPI, implementations of the BSP model [11, 12]). This simple observation highlights the need for tools capable of reconstructing the global control flow in library based implementations.

Current practices show that global barriers are usually *textually aligned*, which means that all processes synchronize on the same textual occurrences. In other words, their use is confined to global control flow. The same applies to other kinds of collective operations. This model not only simplifies programming, but also it is a prerequisite for some program analysis [13–15]. In previous work [16–18], we formally defined textual alignment and prove that enforcing textual alignment of synchronization barriers is a sufficient condition to avoid deadlocks. In this paper, we consider another relevant property: *single-valuedness*. This property states that an expression occurring in the text of a program evaluates to the same value at all processes. For some collectives, not only processes must execute the same instruction but also they must execute it on the same data. For example, in MPI all processes must pick the same source process when performing a broadcast. By combining textual alignment and single-valuedness, one can enforce this stronger property. The main contribution of this paper is to show (Sect. 7) :

- how textual alignment can be used to provide a formal definition of the single-valuedness property. This approach solves a problem raised by Aiken et al. in [19, 20]; namely the comparison of values computed by distinct programs at programs points occuring between global synchronization barriers.
- how, when combined with textual alignment, single-valuedness can be used to enforce proper synchronization of DRMA programs

In Sect. 3 we present a subset of the BSPlib language supporting DRMA communications. In Sect. 4 we present BSP_{DRMA} a toy language that mimics the features of this BSPlib subset. We define its operational semantics and two kinds of errors we aim to rule out, deadlocks and stack mismatches. Sections 6 and 7 introduce formal definitions of textual alignment and single valuedness and show how these properties can be used to rule out errors introduced in sect. 4. We conclude in Sect. 8.

2 Related Works

In [19, 20], Aiken and Gay introduced the concept of structural correctness in non textually aligned programs. This property ensures the absence of deadlocks. Unlike our work it only considers parameterless primitives. Their work had been used for the design of the Titanium language [21, 22]. A latter proposal introduced textually aligned barriers in Titanium by revisiting structural correctness. This proposal was finally replaced by a dynamic approach [23] after it has been observed that it was flawed [23, 24]. A recent work also considers dynamic approaches to the problem of textual alignment of collectives [25]. In [26], the authors consider an empirical static analysis for detecting Multi-Valued expressions, which is used to lower the number of dynamic checks. Barrier checking for non-textually aligned is also studied in [27].

The formal definition of textual alignment was first introduced in [17]. In [28] the authors propose a sufficient condition to ensure correct usage of registers in BSPlib programs. This condition requires that all BSP actions are textually aligned (i.e. concomitant in a logical sense) but also that memory locations used by concomitant DRMA operations are the same up-to renaming. The condition is expressed over execution traces and provides no programming methodology to ensure it. No formal definition of the meaning of "computing the same value at the same time" was provided. This paper improves over [28], it provide such a definition (based on textual alignement) and shows how it can be used to provide a sufficient condition to ensure correctness in the sense of the later. Some formal definitions of the semantics of BSP programs have already been proposed [29, 30], some were mechanized in Coq [31].

3 BSP

In the Bulk Synchronous Parallel (BSP) model, a static number of processes progress at the same pace, synchronizing on global barriers. Between two consecutive synchronizations, processes perform local computations and issue communication requests. These requests will be handled at the time of the next synchronization barrier. Communication requests are either message-passing requests or direct remote memory access requests. Several implementation of the BSP model exists (BSPlib [6], BSML [32], BSPOnMPI [33], MulticoreBSP [34],...). Here we focus on BSP DRMA primitives as proposed by BSPlib and provide a short description of their behavior.

Each process has access to its identifier and to the total number of processes through the functions

`bsp_pid_t bsp_pid(void)` and `bsp_pid_t bsp_nprocs(void)`

The execution of a program proceeds in successive steps, which are local computation steps issuing communication requests, terminated by global synchronization barriers. Requests issued during a step are served at the end of this step (Fig. 1). Synchronization barriers are performed by a collective call to

`void bsp_sync(void)`

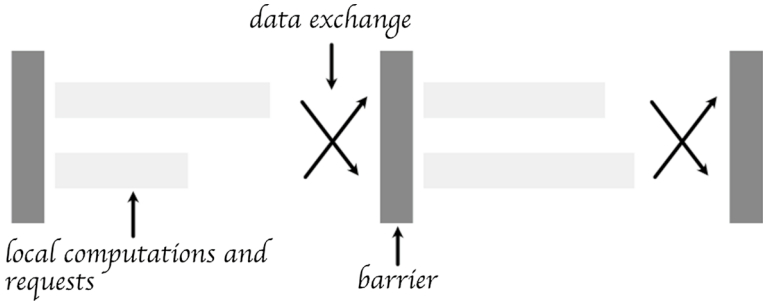

Fig. 1 Execution of a BSP program

Processes can communicate by accessing the memory of other processes. To refer to a remote memory location processes rely on a mapping between local addresses. This mapping is built programmatically. Processes collectively push memory locations on a distributed stack. As the stacks have the same size at every processes, they can be seen as a global stack of p-tuples (where p is the number of processes). Elements of a tuple are the physical adresses at each process of a replicated logical adress. For example in (Fig. 2) both process 0 and process 1 have pushed the address of the variable `y` which is *a* at process 0 and *b* at process 1. In this case the address *a* of process 0 is mapped to the address *b* of process 1 (and vice versa). Pushing a memory location on the stack is done by a call to

```
void bsp_push_reg(const void* addr, bsp_size_t size)
```

where `addr` is the local memory location and `size` is the length of the buffer. A line of the global stack can be removed by a collective call to

```
void bsp_pop_reg(const void* addr)
```

Finally, a process can issue a read or write request to a remote location on process `pid` by calling the following functions

- `void bsp_put(bsp_pid_t pid, const void * src, void *dst, bsp_size_t offset, bsp_size_t size)`: writes `size` bytes from the location `src` to the offset `offset` of the remote location `dst`.
- `void bsp_get(bsp_pid_t pid, const void * src, void *dst, bsp_size_t offset, bsp_size_t size)`: reads `size` bytes from the offset `offset` of the remote location `src` to the location `dst`.

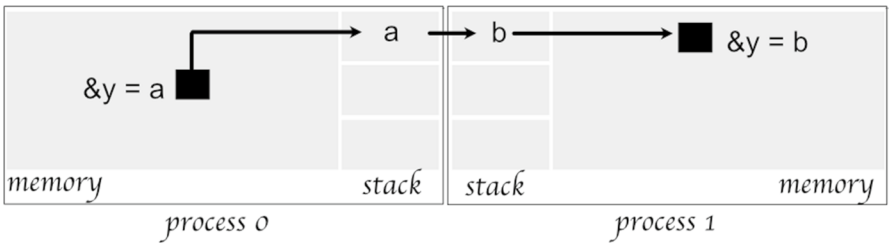

Fig. 2 Two processes registering physical addresses into their stacks

Example 1 Here, each process writes its id in the memory of its right neighbor. First each process pushes the address of variable `y` and perform a barrier to update the stack (`bsp_put_reg` and `bsp_pop_reg` issue requests that are served at the end of the current step). Then each process requires a write of the value of `x` to the distant register matching the address of `y` in the stack. Finally, processes pop the addresses and perform a barrier to execute the communication.

```
y = 0;
bsp_push_reg(&y, sizeof(int));
bsp_sync();
bsp_put((bsp_pid() + 1) % bsp_nprocs(), &x, &y, 0,
  sizeof(int));
bsp_pop_reg(&y),
bsp_sync();
```

In this paper we consider two kinds of errors:

- deadlocks: occur when one process is blocked on a barrier while another process is terminated (we only consider partial correctness)
- stack mismatch: two processes pop locations occurring in distinct lines of the stack. As we have seen each line is a mapping between remote addresses, processes must thus agree on which line to remove.

In the next section we present a simple language and its semantics. This language is used to formalize these kinds of errors.

4 BSP_{DRMA}

The language BSP_{DRMA} is a variation of the While language, extended by a minimal set of primitives dedicated to BSP-like DRMA programming. It supports global synchronization, dynamic register allocation, push and pop operations and updates of remote registers. It is akin to the subset of BSP of Sect. 3, besides a few limitations whose aim is to simplify the presentation. Dynamic allocation is supported only in the context of DRMA communications, there are no heap allocated structures. We use the name *register* instead of memory location to reflect this. Registers hold data of size one. We modify the communication scheme to avoid nondeterminism due to concurrent writes to remote registers. More precisely, messages received by a process are buffered on a per source process basis and can be read separately by the target process. These restrictions are made without loss of generality:

– in practice, shared memory locations are used to exchange contiguous data (dynamic structures are serialized, processes don't exchange pointers),
– concurrent writes are communication errors that we do not deal with in this paper.

4.1 Syntax

We consider a countable set of variables *Var* that we note x, y, z, ... An expression a is a term built from integers, variables, arithmetic operations and the constants *pid* and *nprocs*. The two constants denote respectively the current process id and the total number of processes. Statements are decorated with labels taken in a countable set *Lab*, elements of which are noted ℓ (possibly with subscript). The syntax of the language is given in Fig. 3. Each label occurs at most once in a statement. When not necessary labels are omitted.

The instruction `skip` does nothing and returns control to the rest of the computation. An instruction $x := a$ stores the value of the expression a in the variable x. Sequences, conditionals and loops behave as usual.

– global synchronizations are performed by `sync`. It is a blocking instruction that must be performed collectively by all processes. Pending requests are realized at the time the synchronization occurs.
– a fresh register is allocated and stored in the variable x by the instruction `init` x.
– a register stored in x is pushed (resp. popped) by the instruction `push` x (resp. `pop` x).
– An instruction $x[a] \leftarrow y$ performs a request to update the register paired with the register stored in x at process a. The new value is that of y.
– an instruction $[\texttt{with}\ x \leftarrow y[a]]\ \{s\}$ stores in x the last value written by processor a in the register paired with the register stored in y and executes s. The value was written at the previous step. If no such value exists, s is ignored and the control returns immediately to the rest of the computation.

As stated before, remote writes in BSP_{DRMA} differs a bit from what can be found in BSPlib. Here, a register is rather like a buffer in which other processes can write at a reserved position. Let's rephrase the example of the previous Sect. in BSP_{DRMA} to illustrate this:

$$a ::= pid \mid nprocs \mid x \mid \ldots \qquad \text{(expressions)}$$

$$i ::= \texttt{skip} \mid x := a \mid \texttt{sync} \mid \texttt{init}\ x \mid \texttt{push}\ x \mid \texttt{pop}\ x \mid x[a] \leftarrow y \qquad \text{(instructions)}$$

$$s ::= [i]^{\ell} \mid s; s \mid [\texttt{if}\ a]^{\ell}\ \{s\}\ \{s\} \mid [\texttt{while}\ a]^{\ell}\ \{s\} \mid [\texttt{with}\ x \leftarrow y[a]]^{\ell}\ \{s\} \qquad \text{(statements)}$$

Fig. 3 Syntax of BSP_{DRMA}

$x := pid$;
`init` z; `push` z; `sync`;
$z[(pid + 1) \bmod nprocs] \leftarrow x$;
`pop` z; `sync`
[`with` $y \leftarrow z[(pid - 1) \bmod nprocs]$] {`skip`};

We use the local variable y to store the value written in z by the right neighbor of the process. Specifying the emitter rather than reading the last written value rules out nondeterminism.

4.2 Semantics

We give an operational semantics for our language as a small-step transition system. The semantics records execution paths (sequences of labels) followed by processes during the execution. These annotations will be used in the definitions of textual alignment and single-valuedness. They have no effect on the behavior of programs and can be erased. The purpose of this semantics is to provide annotations that are needed to define textual alignement and single-valuedness in the following sections. It does not improve the description of programs made by previous BSP semantics. However it does improve the annotations and formal definitions we gave in [17] which are now much more simple.

4.3 Definitions

A path pt is a finite sequence of labels. A register is a triple (u, pt, ℓ) where u belongs to a countable set of names $\sqcap$, pt is a path and ℓ a label. A value $v \in \mathcal{V}$ is either an integer or a register. An environment $E \in Env$ is a mapping from variables to values.

The semantics of expressions is given by a function $[\![.]\!] : Env \times nat \to \mathcal{V}$ where the second parameter is the process id at which the evaluation takes place. The special constant pid returns the process id so we have $[\![pid]\!](E, i) = i$. The special constant $nprocs$ returns the number of processes so we have $[\![nprocs]\!](E, i) = p$. Unlike the process id, the number p of process, which is unique for a given execution, is left implicit to improve readability. The semantics of remaining expressions is as usual. For the sake of simplicity, we assume that $[\![.]\!](E, i)$ is a total function.

The semantics is a transition system over global states, which consists of vectors of size p where p is the number of processes. Components of vectors are processes states. They have the form of tuples $(s, (E, S, B, R), pt)$ where, at process i,

- s is either a statement or the termination symbol $\bullet$
- E is the environment of process i
- S is the contribution of process i to the stack
- B is the buffer of process i; it is a function mapping registers and process ids to values. It is a partial function. If $B(u)[j] = v$ then process j has requested an update of u with value v at i.

– R is the history of requests performed by process i since the beginning of the current step. Requests, noted r, are defined by

$$\begin{array}{rll} r ::= & push(u, pt, \ell) & \text{push request} \\ | & pop(u, pt, \ell) & \text{pop request} \\ | & write((u, pt, \ell), i, v) & \text{message request} \end{array}$$

A $write((u, pt, \ell), i, v)$ denotes a write of value v at the location paired with u at process i. We note $R_1 \cdot R_2$ the concatenation of R_1 and R_2.
– pt is the sequence of labels crossed by i since the beginning of the execution.

As usual a context C denotes the rest of the computation, it has the form of a statement with a hole. Given a context C and a statement s we note $C[s]$ for the result of placing s in the hole in C. Contexts are defined by the grammar:

$$C ::= [\,] \mid [\,];s$$

We generalize the notation to the termination symbol $\bullet$ by the equations: $[\bullet] = \bullet$ and $[\bullet];s = s$. Given a function f, We note $f[x \mapsto v]$ the function defined by

$$f[x \mapsto v](y) = \begin{cases} v & \text{if} x = y \\ f(y) & \text{otherwise} \end{cases}$$

If f is partial, we note $dom(f)$ its definition domain. We note $\Gamma \cdot \gamma$ a sequence Γ extended with the element γ.

A global transition, $\nu \to \nu'$ moves from one global state to the next. Figure 4 gives the two rules that define global transitions.

– Rule local specifies individual computation steps. The vector is updated according to the result of a transition of the picked component (see below). Note that processes may execute `sync` instructions occurring at distinct labels.
– Rule sync specifies global steps which occur when all process reach a synchronization barrier. Communications requested during the computation step are served. All components of the vector are updated according to the result. The definition of $\rhd$ is given in Fig. 5. We note $R^{\downarrow}$ (resp. $R^{\uparrow}$) the sequence, in order, of registers pushed (resp popped) in R.

In both cases we record the labels crossed by processes. Other rules specify local transitions. They have the form

$$pt, \ell \vdash_i s, E, B \to s', E', r$$

where i is the process id, pt is the sequence of labels crossed so far by i and ℓ is the current label. Executing s with environment E and buffer B leads to the statement s' and the environment E' performing the request r (or ϵ if no request is performed).

– The `skip` instruction, assignment, conditional and loops behave as usual (rules skip, assign, if1,if2,while1 and while2).

$$\frac{}{pt, \ell \vdash_i [\texttt{skip}]^\ell, E, B \to \bullet, E, \epsilon} \text{ skip}$$

$$\frac{[\![a]\!](E, i) = v}{pt, \ell \vdash_i [x := a]^\ell, E, B \to \bullet, E[x \mapsto v], \epsilon} \text{ assign}$$

$$\frac{fresh(B) = u}{pt, \ell \vdash_i [\texttt{init } x]^\ell, E, B \to \bullet, E[x \mapsto (u, pt, \ell)], \epsilon} \text{ init}$$

$$\frac{E(x) = (u, pt, \ell)}{pt, \ell \vdash_i [\texttt{push } x]^\ell, E, B \to \bullet, E, push(u, pt, \ell)} \text{ push}$$

$$\frac{E(x) = (u, pt, \ell)}{pt, \ell \vdash_i [\texttt{pop } x]^\ell, E, B \to \bullet, E, pop(u, pt, \ell)} \text{ pop}$$

$$\frac{E(y) = v \qquad E(x) = (u, pt', \ell') \qquad [\![a]\!](E, i) = j}{pt, \ell \vdash_i [x[a] \leftarrow y]^\ell, E, B \to \bullet, E, write(u, j, v)} \text{ send}$$

$$\frac{E(y) = (u, pt', \ell') \qquad [\![a]\!](E, i) = j \qquad B(u)[j] = v}{pt, \ell \vdash_i [\texttt{with } x \leftarrow y[a]]^\ell \ \{s\}, E, B \to s, E[x \mapsto v], \epsilon} \text{ receive}_1$$

$$\frac{E(y) = (u, pt', \ell') \qquad [\![a]\!](E, i) = j \qquad B(u)[j] \ undefined}{pt, \ell \vdash_i [\texttt{with } x \leftarrow y[a]]^\ell \ \{s\}, E, B \to \bullet, E, \epsilon} \text{ receive}_2$$

$$\frac{[\![a]\!](E, i) \neq 0}{pt, \ell \vdash_i [\texttt{if } a]^\ell \ \{s_1\} \ \{s_2\}, E, B \to s_1, E, \epsilon} \text{ if}_1$$

$$\frac{[\![a]\!](E, i) = 0}{pt, \ell \vdash_i [\texttt{if } a]^\ell \ \{s_1\} \ \{s_2\}, E, B \to s_2, E, \epsilon} \text{ if}_2$$

$$\frac{[\![a]\!](E, i) \neq 0}{pt, \ell \vdash [\texttt{while } a]^\ell \ \{s\}, E, B \to s; [\texttt{while } a]^\ell \ \{s\}, E, \epsilon} \text{ while}_1$$

$$\frac{[\![a]\!](E, i) = 0}{pt, \ell \vdash [\texttt{while } a]^\ell \ \{s\}, E, B \to \bullet, E, \epsilon} \text{ while}_2$$

$$\frac{pt, \ell \vdash_i s, E, B \to s', E', r \qquad \pi_i(\boldsymbol{v}) = (C[s], (E, S, B, R), pt) \qquad \pi_i(\boldsymbol{v'}) = (C[s'], (E', S, B, R \cdot r), pt \cdot \ell)}{\boldsymbol{v} \to \boldsymbol{v'}} \text{ local}$$

$$\frac{\boldsymbol{w} \triangleright \boldsymbol{w'} \qquad \forall i. \pi_i(\boldsymbol{v}) = (C_i[[\texttt{sync}]^{\ell_i}], \pi_i(\boldsymbol{w}), pt_i) \qquad \forall i. \pi_i(\boldsymbol{v'}) = (C_i[\bullet], \pi_i(\boldsymbol{w'}), pt_i \cdot \ell_i)}{\boldsymbol{v} \to \boldsymbol{v'}} \text{ sync}$$

Fig. 4 Dynamic Semantics

$$
\begin{array}{ll}
\Gamma/\epsilon & = \Gamma \\
(\Gamma \cdot \gamma)/(\Gamma' \cdot \gamma') & = \begin{cases} \Gamma/\Gamma' & \text{if } \gamma = \gamma' \\ (\Gamma/(\Gamma' \cdot \gamma')) \cdot \gamma & \text{otherwise} \end{cases} \\
\epsilon/\Gamma \cdot \gamma & = \mathbf{error} \\
\mathbf{error} \cdot \gamma & = \mathbf{error}
\end{array}
$$

Fig. 5 Exchange

- An instruction `init` x generates a fresh register stored in variable x (rule init). The new register is annotated with pt and ℓ. We assume a function *fresh* that maps a buffer to a fresh register, two buffers with the same domain are mapped to the same register.
- An instruction `push` x (resp. `pop` x) performs a request to push (resp. pop) the register (u, pt, ℓ) stored in x. A request $push(u, pt, \ell)$ (resp. $pop(u, pt, \ell)$) is issued (rules push and pop).
- An instruction $x[a] \leftarrow y$ performs a request to write the value v stored in y to a remote register paired with the register stored in x . The value j of a is the target process. A request $write(u, j, v)$ is issued (rule send).
- A statement $[\texttt{with}\, x \leftarrow y[a]]^{\ell}\ \{s\}$ reads the message sent to the current process to the register stored in y and stores it in x (rule receive$_1$). The control is then returned to the statement s. If no such message exists, the control is simply returned to the rest of the computation (rule receive$_2$).

Given an environment E, the initial state $init(E)$ is $(E, \epsilon, \emptyset, \epsilon)$. We note $E \vdash s \rightsquigarrow_i (s', st, pt)$ if $\langle (s, init(E), \epsilon), \ldots, (s, init(E), \epsilon) \rangle \rightarrow^* v$ and $\pi_i(v) = (s', st, pt)$. The relation $\rightsquigarrow_i$ is the projection on process i of an execution. We note $\rightarrow^*$ the reflexive transitive closure of v and say that v' is reachable from v if $v \rightarrow v^*$. The semantics is deterministic in the sense defined below. Indeed, local transition are deterministic and "scheduling" choice are not significant. Moreover, thanks to the `with` constructs, communications are also deterministic.

Lemma 1 *Let v, v_1 and v_2 be vectors. If $v \rightarrow^* v_1$ and $v \rightarrow^* v_2$ then there exists v' such that $v_1 \rightarrow^* v'$ and $v_2 \rightarrow^* v'$.*

5 Programming Errors

As stated before, we intend to rule out two kinds of errors: deadlocks and stack mismatches. In this section we introduce their formal definitions. A deadlock occurs when a process is blocked at a barrier waiting for a terminated process. All processes are stuck (we do not consider infinite loops).

Definition 1 A deadlock occurs in v, if the following property holds

$$deadlock(v) = \exists i, j.\pi_i(v) = (C[\texttt{sync}], -, -) \wedge \pi_j(v) = (\bullet, -, -).$$

A statement is *well-synchronized* if no deadlock occurs in any state reachable from an initial state.

Example 2 The following program is not well-synchronized because some processes perform less synchronizations than others

$$x := pid;\texttt{while}(x > 0)\{x := x - 1;\texttt{sync}\}$$

More precisely all processes but 0 (which is terminated) are stuck on the first barrier. Thanks to determinism, deadlocks are reproducible and are easily observed by programmers.

A stack mismatch occurs when two processes perform incompatible push and pop requests. Intuitively, requests are compatible if all processes perform the same number of push/pop request and if "concomitant" pop requests refer to the same positions in the stacks. Intuitively, a mapping between remote registers, as defined by stacks can be removed but it cannot be modified (see example 3).

Definition 2 A stack mismatch occurs in v if there exists i and j such that $\pi_i(v) = (C_i[\texttt{sync}], (-, S_i, -, R_i), -)$, $\pi_j(v) = (C_j[\texttt{sync}], (-, S_j, -, R_j), -)$ and

$$\begin{array}{ll} |R_i^{\downarrow}| \neq |R_j^{\downarrow}| & \vee \\ |R_i^{\uparrow}| \neq |R_j^{\uparrow}| & \vee \\ (\exists k.(R_i^{\uparrow})_k = u_i \wedge (R_j^{\uparrow})_k = u_j \wedge pos(S_i, u_i) \neq pos(S_j, u_j) & \end{array}$$

A statement is *well-matched* if no stack mismatch occurs in any state reachable from an initial state.

Example 3 The following program is not well-matched because process 0 tries to pop the first line while other processes try to pop the second line.

$$\texttt{init}\ x;\texttt{init}\ y;\texttt{push}\ x;\texttt{push}\ y;\texttt{sync};\texttt{if}(pid = 0)\{\texttt{pop}\ x\}\{\texttt{pop}\ y\};\texttt{sync}$$

But the following statement is well-matched (processes may pop any line).

$$\texttt{init}\ x;\texttt{init}\ y;\texttt{push}\ x;\texttt{push}\ y;\texttt{sync};\texttt{pop}\ x;\texttt{sync}$$

Remember that we only consider programming errors related to misuses of collectives. In particular, we don't consider local errors such as trying to pop or to use non pushed registers In the next section, we will show how to use textual alignment and single-valuedness to define a programming methodology that rules out the two kind of errors we have introduced.

6 Textual Alignment

Instances of textually aligned labels are crossed by all processes at the same pace, at least from a logical point of view. Intuitively, textually aligned code blocks could be executed in a pure SIMD mode. We will return to this remark later. Some programs are obviously classified as textually aligned, whereas others require more explanations to justify their classification.

Example 4 Consider the following statements written in C.

The first statement (line 1) is clearly not textually aligned. Processes execute *sync* instructions that occur in distinct branches. Obviously, label equality is the weaker reasonable condition. On the contrary, it is obvious that the second statement (line 3) should be considered textually aligned. In the third statement (lines 5,6), all processes perform the same number of iterations of the loop. Yet, they call the *sync* primitive at distinct iterations. Although the behaviors of all processes are observationally equivalent, this statement should not be considered as textually aligned (think of loop unrolling). On the opposite, the last statement (lines 8,9) is textually aligned.

```
   if (bsp_pid() > 0) {bsp_sync()} else {bsp_sync()}

   if (bsp_pid() < bsp_nprocs() ) {bsp_sync()} else {
bsp_sync()}

   x = 0;
   while (x<bsp_nprocs()) {if (x = bsp_pid())
bsp_sync(); x = x + 1}

   x = 0;
   while (x < 3) {if (x == 2) bsp_sync(); x = x + 1}
```

Intuitively, a label ℓ is said to be textually aligned if, whenever a process reaches a *textual occurrence* of ℓ, other processes will eventually reach the same occurrence (we consider partial correctness only). An obvious way to distinguish occurrences of a label ℓ is to consider the set of execution paths leading to ℓ. However, for our purpose, this definition is not appropriate as exemplified by the following statement:

$$\texttt{if } b \texttt{ then } x := 0 \texttt{ else } x := 1 \texttt{ end;} x := x + 1$$

In this case, we would like to consider that whichever branch is taken, the same textual occurence of the last assignment is reached. We note Δ_l the function that retain, from a path, the labels of loops surrounding ℓ in a program statement s (we omit the statement which is always clear from the context). Obviously, the information extracted by Δ_ℓ is sufficient to distinguish distinct occurrences of ℓ in the execution trace of processes.

Definition 3 A label ℓ in a statement s is *textually aligned* if for all E, if exists $i < p$ such that $E \vdash_i s \rightsquigarrow (s', st_i, pt_j)$ where $entry(s') = \ell$ then for all $j < p$, there exists st_j and pt_j such that $E \vdash_j s \rightsquigarrow (s', st_j, pt_j)$ and $\Delta_\ell(pt_i) = \Delta_\ell(pt_j)$.

This definition relate local executions rather that global executions. This is because, in general, the execution of the same textual occurrence of a label at distinct processes may be separated by arbitrarily many synchronizations. Although, in this paper, we will consider programs with textually aligned barriers, this property cannot be assumed a priori. For such programs it will be the case that the same textual occurrences of labels are always reached during the same steps by all processes. Indeed, instances of textually aligned label occur in the same order in distinct processes as stated by the following lemma.

Lemma 2 *Let ℓ_1 and ℓ_2 be two textually aligned program points in s and let E be an environment.*

- $E \vdash_i s \rightsquigarrow (s_i, st_i, pt_i)$ and $E \vdash_i s \rightsquigarrow (s'_i, st'_i, pt'_i)$
- $E \vdash_j s \rightsquigarrow (s_j, st_j, pt_j)$ and $E \vdash_j s \rightsquigarrow (s'_j, st'_j, pt'_j)$
- $entry(s_i) = entry(s_j) = \ell_1$ and $entry(s'_i) = entry(s'_j) = \ell_2$ ($entry$: entry label of the statement)
- $\Delta_{\ell_1}(pt_i) = \Delta_{\ell_1}(pt_j)$ and $\Delta_{\ell_2}(pt'_i) = \Delta_{\ell_2}(pt'_j)$

then if $pt_i \prec pt'_i$ we also have $pt_j \prec pt'_j$ where $\prec$ is the prefix order.

We omit the proof of this intermediate result, the interested reader can refer to our previous Coq[35] developments [36].

Proposition 1 *If all barriers of a statement are textually aligned then this statement is well-synchronized.*

Proof Let i and j be two process ids. We prove that for all $n > 0$ if $E \vdash_i s \rightsquigarrow (C[[\texttt{sync}]^\ell], w_i, pt_i)$ where i has crossed n barriers then there exists a vector v such that $\langle \dots (s, init(E), \epsilon) \dots \rangle \rightarrow^* v$, $\pi_j(v) = (C[[\texttt{sync}]^\ell], w_j, pt_j)$ and $\Delta_\ell(pt_i) = \Delta_\ell(pt_j)$. The proof is by induction on n. Suppose that $E \vdash_i s \rightsquigarrow (C[[\texttt{sync}]^\ell], w_i, pt_i)$, then by hypothesis we have $E \vdash_j s \rightsquigarrow (C[[\texttt{sync}]^\ell], w_j, pt_j)$ and $\Delta_\ell(pt_i) = \Delta_\ell(pt_j)$. We have to prove that these two local state are part of the same synchronisation. Suppose that the local state of $(C[[\texttt{sync}]^\ell], w_j, pt_j)$, which we call A_j, corresponds to another synchronization (otherwise we are done). We distinguish two cases, whether A_j occurs in a synchronization preceding $A_i = (C[[\texttt{sync}]^\ell], w_i, pt_i)$ or not.

- Suppose A_j was part of a previous synchronization. Then we have a previous state of i of the form (s'_i, w'_i, pt'_i) such that $pt'_i \prec pt_i$ and, by induction hypothesis, $\Delta_\ell(pt'_i) = \Delta_\ell(pt_j)$. Consequently $\Delta_\ell(pt_i) = \Delta_\ell(pt'_i)$ which is incompatible with $pt'_i \prec pt_i$.

– Now suppose A_j was not part of a previous synchronization. We assumed A_j is not part of same synchronization as A_i. Then there exists a state $A'_j = (C'[[\texttt{sync}]^{\ell'}, w'_j, pt'_j)$ which is part of the same synchronization as A_i and $pt'_j \prec pt_j$. But by textual alignment, there exists $A'_i = (C[[\texttt{sync}]^{\ell'}], w'_i, pt'_i)$ such that $\Delta_{\ell'}(pt'_i) = \Delta'_{\ell}(pt_j)$. We have $pt_i \prec pt'_i$ otherwise A_i and A'_i denote the same local state and then we have $\ell = \ell'$ and $\Delta_{\ell}(pt_j) = \Delta_{\ell}(pt_i) = \Delta_{\ell}(pt'_i) = \Delta_{\ell}(pt'_j)$ which leads to a contradiction. Finally we get a contradiction by Lemma 2.

From this result it is immediate that assuming a deadlock leads to a contradiction. □

In this section we have shown how textual alignment can be used as a sufficient condition to ensure correctness of parameterless collectives such as global synchronization barriers. More elaborated collectives require not only the execution of the same instruction but also coherency in the actual values the instruction is used with. This is the case, for example, of the broadcast instruction in MPI for which all processes must agree on the source process. In the next section, we consider single-valuedness and show how it can be used to prove correctness of *push* and *pop* instructions.

7 Single Valuedness

As informally defined in the literature, a variable is single-valued if all processes map it to the same value at the same time. Quoting [19],

> [...] "at the same time" is a slippery notion in a setting with asynchronous execution. Only at global synchronization points (i.e., barriers, broadcasts, and the start and end of execution) is it possible to assert anything useful about the state of all processes,

In this section, we show how textual alignment can be used to define a logical notion of time that is more effective than the one induced by global synchronization points. This section is the main contribution of the paper. In particular, we show:

– how to use this logical time to define formally the single-valuedness property (Sect. 7.1)
– how, when combined with textual alignment, this property can be used to define a correctness criterion to enforce a proper use of collectives (Sect. 7.2)

7.1 Definition

Before we present the definition of single-valuedness, we must give some precisions about the equivalence relation on which it it is based. Basically, single-valuedness refers to equality. However, we need to consider a slightly weaker relation that does not distinguish memory locations allocated at the same logical time (as defined by

textual alignement). More precisely, we identify memory location carrying the same time annotations. Two values v_1 and v_2 are equivalent, noted $v_1 \backsimeq v_2$, if

- $v_1, v_2 \in \mathit{Int}$ and $v_1 = v_2$ or,
- $v_1 = (u, pt, \ell)$ and $v_2 = (u', pt', \ell')$ where $\ell = \ell'$ and $\Delta_\ell(pt) = \Delta_\ell(pt')$

As mentioned earlier, we rely on textually aligned program points to define single-valuedness. Given a textually aligned program point ℓ, a variable x is single-valued at ℓ if whenever two processes reach ℓ "at the same time", the occurences of x at each process hold equivalent values. The following definition states this property more formally.

Definition 4 A variable x in a statement s is *single-valued* at label ℓ if ℓ is textually aligned and if for all E, if there exists $i < p$ and $j < p$ such that $E \vdash_i s \rightsquigarrow (s_i, (E_i, S_i, B_i, R_i), pt_i)$ and $E \vdash_j s \rightsquigarrow (s_j, (E_j, S_j, B_j, R_j), pt_j)$ where $\mathit{entry}(s_i) = \mathit{entry}(s_j) = \ell$ and $\Delta_\ell(pt_i) = \Delta_\ell(pt_j)$ then $E_i(x) \backsimeq E_j(x)$.

The notion of single value easily extends to expressions by requiring that their evaluation leads to equivalent values in the local environment.

Example 5 In the following examples, the pushed variable is single-valued in the first statement but not in the second.

$$s_1 = [\texttt{init}\ x]^{\ell_1};[\texttt{push}\ x]^{\ell_2}$$

$$s_2 = \texttt{if}\ [b]^{\ell_1}\ \texttt{then}\ [\texttt{init}\ x]^{\ell_2}\ \texttt{else}\ [\texttt{init}\ x]^{\ell_3}\ \texttt{end};[\texttt{push}\ x]^{\ell_4}$$

Indeed, in the first example, we have $\mathit{entry}([\texttt{push}\ x]^{\ell_2}) = \mathit{entry}([\texttt{push}\ x]^{\ell_2}) = \ell_2$, $\Delta_{\ell_2}(\ell_1) = \Delta_{\ell_2}(\ell_1) = \epsilon$ and $(u, \epsilon, \ell_1) \backsimeq (v, \epsilon, \ell_1)$ for the run

$$[] \vdash_0 s_1 \rightsquigarrow ([\texttt{push}\ x]^{\ell_2}, ([x \mapsto (u, \epsilon, \ell_1)], -, -, -), \ell_1])$$
$$[] \vdash_1 s_1 \rightsquigarrow ([\texttt{push}\ x]^{\ell_2}, ([x \mapsto (v, \epsilon, \ell_1)], -, -, -), \ell_1])$$

In the second example, we have $\mathit{entry}([\texttt{push}\ x]^{\ell_4}) = \mathit{entry}([\texttt{push}\ x]^{\ell_4}) = \ell_4$ $\Delta_{\ell_4}(\ell_1\ell_2) = \Delta_{\ell_4}(\ell_1\ell_3) = \epsilon$ but $(u, \epsilon, \ell_2) \not\backsimeq (v, \epsilon, \ell_3)$ for the run

$$[] \vdash_0 s_2 \rightsquigarrow ([\texttt{push}\ x]^{\ell_4}, ([x \mapsto (u, \epsilon, \ell_2)], -, -, -), \ell_1\ell_2])$$
$$[] \vdash_1 s_2 \rightsquigarrow ([\texttt{push}\ x]^{\ell_4}, ([x \mapsto (v, \epsilon, \ell_3)], -, -, -), \ell_1\ell_3])$$

7.2 Correctness Criterion

In this section, we show how to use the single-valuedness property to avoid stack mismatches. Intuitively, if *push* and *pop* instructions are textually aligned, all processes will perform the same sequence of *push*/*pop* instructions (up-to parameters). This is sufficient to avoid stack mismatches due to *push* instructions.

However, we must also ensure that concomitant *pop* instructions must refer to the same line in the stack. This last requirement is met by enforcing concomittent *push*/*pop* to operate over single-valued memory locations. Global synchronisation barriers must also be textually aligned. The following proposition formalizes this intuition.

Proposition 2 *If all barriers in s are textually aligned and if push/pop instructions in s are textually aligned and single-valued then s is well matched.*

Proof We prove a stronger property which is that for every reachable state v there exists R, R' and S such that for each process i we have $\pi_i(v) = (C_i[\texttt{sync}], (-, S_i, -, R_i), -)$ and

$$R_i^{\downarrow} \backsimeq R \wedge R_i^{\uparrow} \backsimeq R' \wedge S_i \backsimeq S$$

where $\backsimeq$ is applied point-wise. Because barriers are textually aligned, we can reason on a single step and conclude by induction on the number of barriers crossed so far. So suppose the property holds at the beginning of the step. By hypothesis and by Lemma 2, push/pop instructions at ℓ that occurs in i at path pt_i also occurs, in same order, in j at path pt_j and $\Delta_\ell(pt) = \Delta_\ell(pt_j)$. By the single value hypothesis we have requests that are performed in the same order with compatible (with respect to $\backsimeq$) values. Moreover, the stacks are equivalent because of the hypothesis (pushed/pop values were equivalent at the end of the previous step). If there is no previous step, the initial state trivially satisfies the conditions. □

In this section, we have shown how textual alignement can be used to define an effective notion of logical time. We have used this logical time to provide the first, as far as we know, formal definition of single-valuedness in SPMD programs. Finally, we have shown how to combine textual alignment and single-valuedness to define sufficient conditions to ensure correctness of DRMA collectives in a BSP like language. Here, we have focused on issues related to the collective nature of *push* and *pop* instructions. It is still possible for a program to get stuck because processes collectively fail, for example by trying to *pop* a non-valid register. Such behavior can be ruled out by simple local correctness properties. This issue was already considered in [28].

8 Conclusion

We have formally defined sufficient conditions to ensure correctness of collective DRMA communications la BSP. These sufficient conditions rely on the formal definitions of two important properties, namely textual alignment and single-valuedness. The formal definition of textual alignment improves on our previous work in term of simplicity. As far as we know this is the first formal definition of single-valuedness based on textual alignment. We have shown that the latter

permits to define the former not only at synchronization points but also at all textually aligned points. Indeed, textually aligned program points behave as synchronization points at the logical level. We are currently working on a static analysis. We also expect to study the interaction of our analysis with the PARCOACH [37] dynamic checker. In this context, we expect our analysis to reduce the overhead of its instrumentation by removing checks that operate over textually aligned program points. Another direction is to consider more collective communication schemes like broadcast, map, reduce and others.

References

1. 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)
2. 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)
3. Flynn, Michael J.: Some computer organizations and their effectiveness. IEEE Trans. Comput. **21**(9), 948–960 (1972)
4. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.1
5. OpenMP Architecture Review Board. OpenMP application program interface version 3.0. http://www.openmp.org/mp-documents/spec30.pdf, May (2008)
6. van Duijn, Mick, Visscher, Koen, Visscher, Paul : BSPLib: a fast, and easy to use C++ implementation of the Bulk Synchronous Parallel (BSP) threading model. http://bsplib.eu/
7. Loulergue, Frédéric, Hains, Gaétan: *Functional parallel programming with explicit processes: Beyond SPMD*, pages 530–537. Springer Berlin Heidelberg, Berlin, Heidelberg, (1997)
8. Loulergue, Frédéric, Gava, Frédéric, Billiet, David: Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction. volume 3515 of *LNCS*, pages 1046–1054. Springer, (2005)
9. Gava, F., Loulergue, F.: A static analysis for bulk synchronous parallel ml to avoid parallel nesting. *Future Generation Computer Systems*, 21(5):665 – 671, (2005). Parallel computing technologies
10. Darema, Frederica: *SPMD Computational Model*, pages 1933–1943. Springer US, Boston, MA, (2011)
11. Hill, Jonathan M. D., McColl, Bill, Stefanescu, Dan C., Goudreau, Mark W., Lang, Kevin , Rao, Satish B., Suel, Torsten, Tsantilas, Thanasis, Bisseling, Rob H.: Bsplib: The bsp programming library. *Parallel Comput.*, 24(14):1947–1980, December (1998)
12. Yzelman, A.N., Bisseling, R.H., Roose, D., Meerbergen, K.: Multicorebsp for c: A high-performance library for shared-memory parallel programming. Int. J. Parallel Program. **42**(4), 619–642 (2014)
13. Kamil, Amir, Yelick, Katherine: 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
14. Chen, C., Huo, W., Li, L., Feng, X., Xing, K.: 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)
15. Chatarasi, Prasanth, Shirako, Jun, Kong, Martin, Sarkar, Vivek: *An Extended Polyhedral Model for SPMD Programs and Its Use in Static Data Race Detection*, pages 106–120. Springer International Publishing, Cham, (2017)
16. Jakobsson, Arvid, Dabrowski, Frédéric., Bousdira, Wadoud, Loulergue, Frédéric, Hains, Gaetan: Replicated synchronization for imperative BSP programs. Procedia Computer Science, 108:535–544, : International Conference on Computational Science, ICCS 2017, 12–14 June 2017. Zurich, Switzerland (2017)

17. Dabrowski, Frederic: Textual Alignment in SPMD Programs. In *SAC '18: Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, Pau, France, April (2018)
18. Dabrowski, Frédéric.: A denotational semantics of textually aligned spmd programs. J. Logic. Algeb. Methods Program. **108**, 90–104 (2019)
19. Aiken, Alexander, Gay, David: 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
20. Gay, D.: *Barrier Inference*. PhD thesis, University of California, Berkeley, (1998)
21. Yelick, Kathy, Semenzato, Luigi, Pike, Geoff, Miyamoto, Carleton, Liblit, Ben, Krishnamurthy, Arvind, Hilfinger, Paul, Graham, Susan , Gay, David, Colella, Phil , Aiken, Alex: Titanium: a high-performance java dialect. *Concurrency: Practice and Experience*, 10(11-13):825–836, (1998)
22. Hilfinger P. N., (editor), Bonachea, Dan, Gay, David , Graham, Susan, Liblit, Ben, Pike, Geoff, Yelick, Katherine: Titanium Language Reference Manual, Version 1.16.8. Technical Report UCB// CSD-04-1163x, Computer Science, UC Berkeley, (2004)
23. Kamil, Amir, Yelick, Katherine: *Enforcing Textual Alignment of Collectives Using Dynamic Checks*, pages 368–382. Springer Berlin Heidelberg, Berlin, Heidelberg, (2010)
24. Kamil, A.: Problems with the titanium type system for alignment of collectives. unpublished note, (2006)
25. Knüpfer, Andreas, Hilbrich, Tobias, Protze,Joachim, Schuchart, Joseph: *Dynamic Analysis to Support Program Development with the Textually Aligned Property for OpenSHMEM Collectives*, pages 105–118. Springer International Publishing, Cham, (2015)
26. Huchant, Pierre, Saillard, Emmanuelle, Barthou, Denis, Carribault, Patrick: Multi-valued expression analysis for collective checking. In Ramin Yahyapour, editor, *Euro-Par 2019: Parallel Processing*, pages 29–43, Cham, (2019). Springer International Publishing
27. Zhang, Yuan, Duesterwald, Evelyn: 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
28. Bousdira, Wadoud, Jakobsson, Arvid, Dabrowski, Frederic: Safe Usage of Registers in BSPlib. In *SAC 2019*, Limassol, Cyprus, April (2019)
29. Gava, Frédéric, Fortin, Jean: Formal semantics of a subset of the paderborn's bsplib. In *Proceedings of the 2008 Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT '08, page 269–276, USA, (2008). IEEE Computer Society
30. Tesson, Julien, Loulergue, Frédéric: Formal semantics of drma-style programming in bsplib. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski, editors, *Parallel Processing and Applied Mathematics*, pages 1122–1129, Berlin, Heidelberg, (2008). Springer Berlin Heidelberg
31. Fortin, Jean, Gava, Frédéric: Towards mechanised semantics of hpc: The bsp with subgroup synchronisation case. In *Proceedings of the ICA3PP International Workshops and Symposiums on Algorithms and Architectures for Parallel Processing - Volume 9532*, page 222–237, Berlin, Heidelberg, (2015). Springer-Verlag
32. Software. BSML: Bulk synchronous parallel ml, a library for BSP programming in OCaml. https://traclifo.univ-orleans.fr/BSML/
33. Software. BSPOnMPI, a platform independent software library for developing parallel programs. http://bsponmpi.sourceforge.net/
34. Software. MultiCoreBSP, BSP programming on modern multicore processors. http://www.multicorebsp.com/
35. The Coq Development Team. Coq. https://coq.inria.fr
36. Dabrowski, Frédéric: Jlamp (2019), coq artefact. https://github.com/DabrowskiFr/coq-jlamp2018
37. Huchant, Pierre: Saillard, Emmanuelle, Barthou, Denis, Brunie, Hugo, Carribault, Patrick: PARCOACH Extension for a Full-Interprocedural Collectives Verification. In Second International Workshop on Software Correctness for HPC Applications, Dallas, United States (2018)