Safe Usage of Registers in BSPlib (Preprint)

Arvid Jakobsson, Frederic Dabrowski, Wadoud Bousdira

▶ **To cite this version:**

Arvid Jakobsson, Frederic Dabrowski, Wadoud Bousdira. Safe Usage of Registers in BSPlib (Preprint). 2019. hal-01955283

HAL Id: hal-01955283
https://hal.science/hal-01955283

Preprint submitted on 2 Jan 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Safe Usage of Registers in BSPlib (Preprint)

Extended Abstract

Arvid Jakobsson*
Huawei France Research Center
Boulogne-Billancourt, France
arvid.jakobsson@huawei.com

Frédéric Dabrowski
Univ. Orléans, INSA Centre,
Val de Loire, LIFO EA 4022
Orléans, France
frederic.dabrowski@univ-orleans.fr

Wadoud Bousdira
Univ. Orléans, INSA Centre,
Val de Loire, LIFO EA 4022
Orléans, France
wadoud.bousdira@univ-orleans.fr

ABSTRACT

Bulk Synchronous Parallel (BSP) is a simple but powerful high-level model for parallel computation. Using BSPlib, programmers can write BSP programs in the general purpose language C. Direct Remote Memory Access (DRMA) communication in BSPlib is enabled using *registrations*: associations between the local memories of all processes in the BSP computation. However, the semantics of registration is non-trivial and ambiguously specified and thus its faulty usage is a potential source of errors. We give a formal semantics of BSPlib with which we characterize correct registration. Anticipating a static analysis, we give a simplified programming model that guarantees correct registration usage, drawing upon previous work on textual alignment.

CCS CONCEPTS

• **Computing methodologies** → **Parallel programming languages**; • **Theory of computation** → **Logic and verification**; *Operational semantics*;

KEYWORDS

Parallel programming, Bulk Synchronous Parallelism, Static Analysis, Communication

ACM Reference Format:
Arvid Jakobsson, Frédéric Dabrowski, and Wadoud Bousdira. 2019. Safe Usage of Registers in BSPlib (Preprint): Extended Abstract. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3297280.3297421

1 INTRODUCTION

Parallel architectures are known to be hard to program. Having a simple but realistic model of parallelism reduces difficulty. The model should allow the user to predict both behaviour and performance of parallel programs. The *Bulk Synchronous Parallel*-model (BSP) [13] has these qualities along with practical and efficient implementations.

BSP can either be implemented in a dedicated, *parallel language*, as done in BSML [2], or by extending a general purpose language as the BSPlib [8] library does for C. There are many advantages to the latter approach: no need to learn a new language and parallelism can be added to existing applications.

However, errors are easily introduced if the host language permits writing programs that are invalid in the underlying parallel model, such as non-participation in collective synchronization [1]. We argue that static analysis can and should be used to detect such errors, and thus bridge the gap between parallel languages and library embeddings of parallelism. As a step in this direction, we study how to prevent invalid register usage in BSPlib.

A BSPlib registration is an association between p memory addresses, one per process, allowing one process to reference memory objects on remote processes without knowing their address, thus enabling *Direct Remote Memory Access* (DRMA). At synchronization, the BSPlib runtime uses the registrations to route communication. Unfortunately, the BSPlib interface for manipulating registrations is informally defined with subtle corner cases that may provoke dynamic errors.

Other less error-prone schemes for DRMA exist. In the BSP paradigm, Yzelman et al. [14] use a communication container class to turn regular objects into distributed data structures. MPI [9] uses window objects to allow a process to reference remote memory. Like BSPlib registrations, windows act as handles and are created and removed collectively. Unlike registrations, windows can be removed in any order. In OpenSHMEM [4], DRMA operations are only allowed on "symmetric" objects that the runtime system ensures have the same relative address in each process.

Our first contribution is BSPlite, a formalization of BSPlib with registration, with which we characterize correct register usage. Tesson et al. [12] also formalize BSPlib, but do not consider registration. Gava et al. [7] formalize Paderborn's BSPlib [3], but their modelization of registration is simplified. To our knowledge, ours is the first formalization capturing the idiosyncrasies of BSPlib registration.

Our second contribution is a characterization of a subset of correct programs based on textual alignment [5]. With respect to previous work, the notion of textual alignment is generalized to all collective operations, and in a restricted sense to memory location, which requires an instrumentation of the semantics of programs which is slightly more complex. Our final goal (future work) is a static analysis that enforces this model and thus correct registration. We believe this is the first work towards *static* verification of BSPlib registration. Previous work, like BSP-Why [6], enables the user to prove the correctness of their program, including registrations. However, the user must do so manually, whereas we aim for full automation, and it is unclear if BSP-Why's modelization of registration is true to the BSPlib standard.

The article proceeds as follows: Section 2 introduces BSP, BSPlib and registration. Section 3 describes our BSPlib formalization and the instrumentation with which we then define correct registration

*Also with LIFO, l'Université d'Orléans

SAC '19, April 8–12, 2019, Limassol, Cyprus
2019. ACM ISBN 978-1-4503-5933-7/19/04…$15.00
https://doi.org/10.1145/3297280.3297421

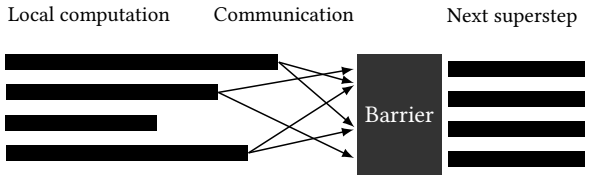


Figure 1: A BSP superstep with $p = 4$

`bsp_begin(int p)`	Start parallel section with p processes
`bsp_end()`	End parallel section
`bsp_pid()`	Return process ID
`bsp_nprocs()`	Return total number of processes p
`bsp_push_reg(void *adr, int len)`	Request creation of registration
`bsp_pop_reg(void *adr)`	Request removal of registration
`bsp_put(int pid, void *src, void *dst, int offs, int len)`	Request remote memory write
`bsp_sync()`	Synchronize and end superstep

Table 1: Subset of BSPlib primitives

(Section 4). In Section 5, we describe our simplified programming model for registers and prove it implies correct registration.

2 BSP & BSPLIB

The Bulk Synchronous Parallel (BSP) model is both a high-level model for reasoning about parallel computers and computations, and a practical programming method as implemented in Apache Hama [11], BSML [2] and BSPlib [8].

In BSP-as-a-model, parallel architectures are abstracted to p processor-memory pairs, pairwise connected by a network and governed by a synchronization unit. A BSP computation is a sequence of *supersteps*, each composed of three phases: isolated computation in each process, collective communication, and barrier synchronization (see Fig. 1). BSP also includes a model for reasoning on the cost of parallel computations, which we do not detail further here.

For BSP-as-a-programming-method, we focus on BSPlib. BSPlib is a library specification with several implementations for direct-mode BSP programming in C. It is characterized by a small but highly composable set of 20 primitives. The subset relevant here is summarized in Table 1. BSPlib programs follow *Single Program, Multiple Data* (SPMD) style. Formally, their execution can be seen as the parallel composition of p copies of the same program P, behaving differently as a function of the process identifier taken as argument: $P(0) \parallel P(1) \parallel \cdots \parallel P(p-1)$.

BPSlib registrations. BSPlib processes typically use DRMA for (buffered) communication, enabled by the `bsp_put` primitive. BSPlib registrations allow a process to write to a remote memory area using a local address as a handle. A registration is an association between p addresses[1], one per process. Collectively calling the functions `bsp_push_reg` and `bsp_pop_reg` requests the addition removal of a registration, respectively. A registration can be seen as a p-vector of addresses $\langle l_i \rangle_i$, where l_i is the argument of process i to `bsp_push_reg`. Registration and communication requests are executed at synchronization, and their effect is visible in the following superstep. Synchronization occurs when processes collectively call `bsp_sync()`.

[1]For simplicity, we ignore the extent of each memory area, which may vary per process.

```
1   bsp_begin(bsp_nprocs());
2   bsp_push_reg(&x,sizeof(x));
3   bsp_sync();
4   if (bsp_pid() == 0)
5     for (i = 0; i < bsp_nprocs(); i++)
6       bsp_put(i, &y, &x, 0, sizeof(y));
7   bsp_pop_reg(&x);
8   bsp_sync();
9   // at this point, x is unregistered and contains
10  // the value of y from process 0
11  bsp_end();
```

Figure 2: An example BSPlib program using registration

The BSPlib program in Fig. 2 uses registration to broadcast process 0's value of `y` into `x` of every other process. In the first superstep, a registration is created containing `x`'s address in each process (Line 2). The registration becomes available in the next superstep. There, process 0 requests the transmission `sizeof(y)` bytes from its memory starting at `&y` into the memory area containing `x` of each process `i` at offset 0 (Line 4-6). The deregistration at Line 7 is safe, since it takes effect in the next superstep.

Intuitively, the rules for registration are the following:

(a) A registration $\langle l_i \rangle_i$ is created when all processes call `bsp_push_reg(l_i)` in the same superstep, and becomes available in the next superstep.
(b) Registration requests must be **compatible**: the order of all pushes must be the same on all processes, and for the pops likewise. However, it does not matter how requests are interleaved within one superstep.
(c) A process can choose not to participate in a registration by passing `NULL` to `bsp_push_reg`.
(d) The same address can be registered multiple times. Only the last registration of an address in the registration sequence is active. The motivation is modularity: to allow addresses to be reused for communication in different parts of the code, possibly unbeknownst to each other.
(e) The last registration $\langle l_i \rangle_i$ is removed when all processes call `bsp_pop_reg(l_i)`, and becomes unavailable in the next superstep. A dynamic error occurs if the last pushed l_i is not at the same level in the registration sequence of all processes.

The running examples in Fig. 3, written in BSPlite (detailed in Section 2), illustrate these rules. In Example 1, execution proceeds without error. In the second superstep, the most recently pushed `y` is removed. In Example 2, the program attempts to remove `x`, but it is not yet registered and so an error is produced (rule (e)). In Example 3, a dynamic error occurs since the pop in the second superstep attempts to remove at different levels in the registration sequence (rule (e)). Example 4 is a simplified version of real BSPlib code, illustrating how the situation of Example 3 can be reproduced by dynamic allocation as `malloc` may return `NULL` [10, p. 143]. In Example 5, a dynamic error occurs when only process 1 pushes (rule (c)). Example 6 illustrates that the interleaving of requests in one superstep is irrelevant (rule (b)). In the next section we formalize these intuitions.

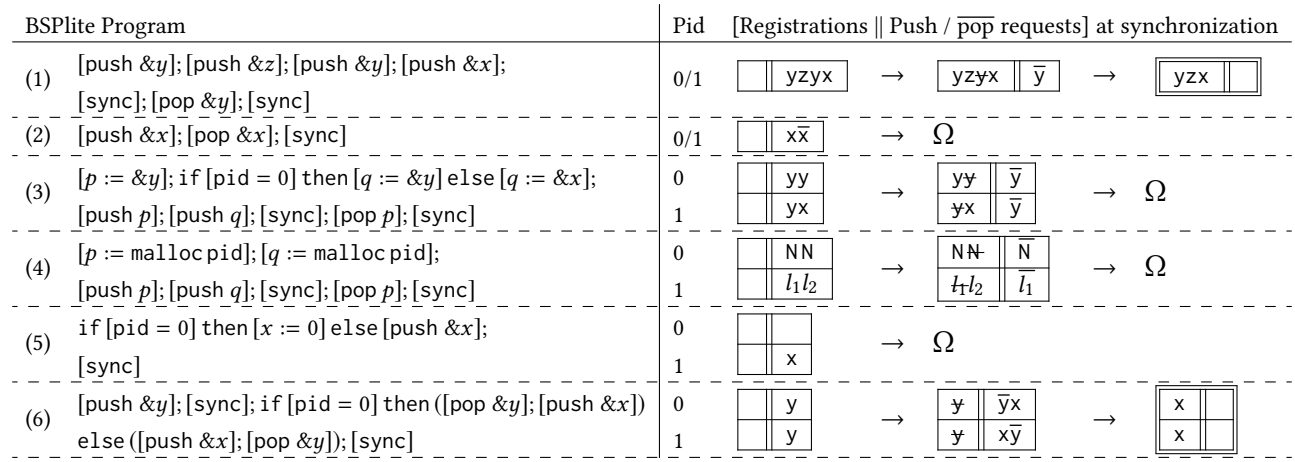


Figure 3: For each running example an execution with $p = 2$ is given depicting the registration sequence and requests before each synchronization. Here, x is the location of variable x, l_i is the ith address returned by `malloc` and `N` is `NULL`. The symbol Ω indicates a registration error. Labels are omitted for legibility.

AExp, $e ::= n \mid e_1\ op_a\ e_2 \mid pe$
$\mid \&x \mid \texttt{nprocs} \mid \texttt{pid}$

PExp, $pe ::= x \mid *e$

BExp, $b ::= \texttt{true} \mid \texttt{false}$
$\mid e_1\ op_r\ e_2 \mid b_1\ op_b\ b_2 \mid \textbf{not}\ b$

Cmd, $c ::= c_1; c_2$
$\mid \texttt{if}\ [b]^\ell\ \texttt{then}\ c_1\ \texttt{else}\ c_2$
$\mid \texttt{while}\ [b]^\ell\ \texttt{do}\ c_1$
$\mid [pe := e]^\ell \mid [pe := \texttt{malloc}\ e]^\ell$
$\mid [\texttt{free}\ e]^\ell \mid [\texttt{sync}]^\ell \mid [\texttt{push}\ e]^\ell$
$\mid [\texttt{pop}\ e]^\ell \mid [\texttt{put}\ e_1\ e_2\ e_3\ e_4\ e_5]^\ell$

$x \in \textbf{Var}, n \in \mathbb{N}, op_r \in \{=, <\}, op_b \in \{\&\&, ||\}, op_a \in \{+, -, *\}$

Figure 4: Syntax of BSPlite with registers

3 BSPLITE WITH REGISTERS

This section presents our first contribution: BSPlite, a formalization of BSPlib with registration. BSPlite is a labelled WHILE-language with pointers, dynamic allocation and parallel primitives (Fig. 4).

Arithmetic expressions are standard, and contain constants, arithmetic operations, pointer expressions and the address-of operator. Two special expressions, `nprocs` and `pid` model `bsp_nprocs` and `bsp_pid` respectively. A pointer expression is either a variable or the dereferencement of an arithmetic expression. Boolean expressions are standard and not detailed further.

In addition to standard sequence, conditional, loops and assignments, commands also include dynamic allocation (`malloc`, whose argument indicates desired allocation size) and deallocation (`free`). The parallel primitives `sync`, `push`, `pop` and `put` model their BSPlib counterpart. We assume that labels ℓ are unique.

3.1 Local Semantics

BSPlite has a two-level semantics: a local, small-step, semantics for the computation phase, and a global, big-step semantics for the superstep structure.

Values are either naturals or locations (see Fig. 5). A location is a base address-offset pair. We distinguish the special location `N` modelling `NULL`, defined as $(b_{\mathsf{N}}, 0)$ where b_{N} is a distinguished base address. The semantics is parameterized by a global, fixed environment σ mapping local variables to unique, non-null locations.

$l \in \textbf{Loc} = \textbf{Base} \times \mathbb{N}$

Val, $v ::= n \mid l$

$\sigma \in \textbf{Env} = \textbf{Var} \to \textbf{Loc}$

$\mathcal{H} \in \textbf{Heap} = \textbf{Loc} \hookrightarrow \textbf{Val}$

$\mathcal{H}_0 \in \textbf{Heap}$ such that
$\forall l \in \textbf{Loc}. \mathcal{H}_0\ l$ is defined if $\exists x \in \textbf{Var}. \sigma\ x = l,\ \mathcal{H}_0\ l = \text{undef}$ oth.

RReq, $r ::= \textsc{push}\ l \mid \textsc{pop}\ l$

CReq, $c ::= \textsc{put}\ j\ vs\ l\ n$

$st = (\mathcal{H}, rs, cs) \in \textbf{State} = \textbf{Heap} \times \textbf{RReq}^* \times \textbf{CReq}^*$

LocalConf, $\gamma ::= \langle c, st \rangle \mid st$

Figure 5: Domain of local semantics

Heaps are partial functions from (allocated) locations to values. The initial heap $\mathcal{H}_0$ allocates the location of all local variables. A subset of the standard semantics of arithmetic, boolean and pointer expressions is given in Fig. 6. For lack of space we do not give all the cases. We restrict pointer arithmetic to the pointer's base: we assume that invalid pointer usage is precluded by a pre-analysis.

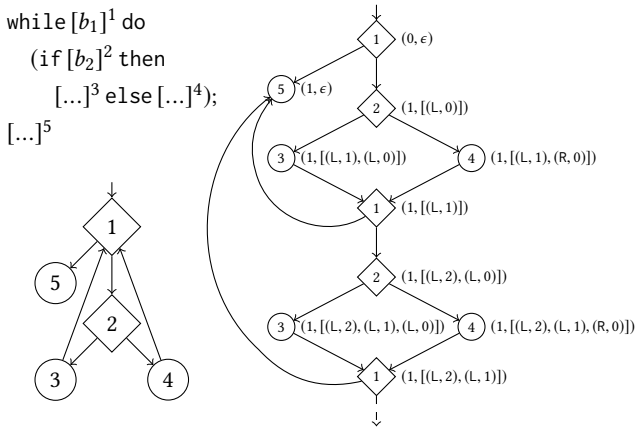
The local semantics operates over configurations that consist of an (optional) continuation, a heap, a list of registration requests and a list of communication requests to execute at the next synchronization. Registration requests contain only the concerned location, while `put`-requests contain the target process id, the list of values to transmit, the location referring to a register in the source process and an offset into this register.

3.1.1 Local Instrumentation. The semantics is instrumented to capture the *trace of actions* taken by the program. Actions are of type `push!`, `pop!` or `sync!`, and are generated by the corresponding commands. The instrumentation has no impact on execution: it only serves as a base for the definitions in Sections 4 and 5.

Each action stores the *path* [5] taken to reach the current program point. Intuitively, the path at a program point encodes the

$$
\begin{array}{lcl}
\mathcal{A}[\![\cdot]\!]_i^p : \textbf{AExp} \to (\textbf{Heap} \hookrightarrow \textbf{Val}) & & \\
\mathcal{A}[\![n]\!]_i^p\ \mathcal{H} & = & n \\
\mathcal{A}[\![\texttt{nprocs}]\!]_i^p\ \mathcal{H} & = & p \\
\mathcal{A}[\![\texttt{pid}]\!]_i^p\ \mathcal{H} & = & i \\
\mathcal{A}[\![pe]\!]_i^p\ \mathcal{H} & = & \mathcal{H}\,(\mathcal{P}[\![pe]\!]_i^p\ \mathcal{H}) \\
\mathcal{A}[\![\&x]\!]_i^p\ \mathcal{H} & = & \sigma\, x \\
 & \cdots & \\
\mathcal{P}[\![\cdot]\!]_i^p : \textbf{PExp} \to (\textbf{Heap} \hookrightarrow \textbf{Loc}) & & \\
\mathcal{P}[\![x]\!]_i^p\ \mathcal{H} & = & \sigma\, x \\
\mathcal{P}[\![*e]\!]_i^p\ \mathcal{H} & = & l \text{ if } \mathcal{A}[\![e]\!]_i^p\ \mathcal{H} = l, \text{ undef oth.} \\
\mathcal{B}[\![\cdot]\!]_i^p : \textbf{BExp} \to (\textbf{Heap} \hookrightarrow \textbf{Bool}) & & \\
 & \cdots &
\end{array}
$$

Figure 6: BSPlite expression semantics



a) Program and CFG **b) Execution**

Figure 7: Illustration of paths. Loop execution is visualized as nested conditionals.

history of choices previously taken at control flow branches, trimming choices of fully executed branches. Trimming ensures that the path only contains choices relevant to the current position.

Formally, a path $\delta \in \textbf{Path}$ is a pair (k, w) where k is the count of commands crossed on the outermost nesting level, and w is a list of choices. A choice is a pair of type $\{\mathsf{L}, \mathsf{R}\} \times \mathbb{N}$, and denotes the branch taken (L for true, R for false) and the count of commands crossed on the nesting level after that choice.

Consider the program depicted in Fig. 7 and its execution. Like illustrated, the execution's loop iterations can be understood as nested conditionals. Program points are decorated with their corresponding path at execution. Each step increments the count on the current level. Iterations of loop 1 also add a choice to the path, all of which are trimmed when leaving the loop for point 5. Similarly, the inner conditional 2 adds a choice that is trimmed after leaving the branches to program point 1. The choices made at conditional 2 is irrelevant to how loop 1 is reached, and choices at loop 1 is irrelevant to how point 5 is reached.

$$
\begin{array}{lcl}
\oplus : \textbf{Path} \times \mathbb{N} \to \textbf{Path} & & \\
(k_0, \epsilon) \oplus n & = & (k_0 + n, \epsilon) \\
(k_0, w +\!\!+ [(ch, k)]) \oplus n & = & (k_0, w +\!\!+ [(ch, k + n)]) \\
\odot : \textbf{Path} \times (\mathbb{N} \times \{\mathsf{L}, \mathsf{R}\}) \to \textbf{Path} & & \\
(k_0, w) \odot (ch, k) & = & (k_0, w +\!\!+ [(ch, k)]) \\
trim : \textbf{Lab} \times (\textbf{Lab}^* \times \textbf{Path}) \to (\textbf{Lab}^* \times \textbf{Path}) & & \\
trim\,\ell\,(\ell : \eta, (k_0, w +\!\!+ [(ch, k)])) & = & trim\,\ell\,(\eta, (k_0, w)) \\
trim\,\ell\,(\eta, \delta) & = & (\eta, \delta)
\end{array}
$$

Figure 8: Operators and functions on paths and nesting stack

Unlike a big-step semantics, once the small-step semantics enters a branch, there is no memory of the encompassing structure, and we can no longer discern when the conditional's body ends and thus when to trim the path. To remedy, we keep a *nesting stack* η. For each partially executed conditional, a label in the stack indicates the program point at which the corresponding choice must be trimmed from the path. Introducing artificial commands denoting the branch's end would solve the same problem, but pollute the syntax.

Actions push! and pop! store the concerned location and its *source*: the memory object from whence it was obtained. This is either a local variable, an instance of dynamic allocation denoted by its path or *unknown* (for communicated pointers and integer values). To track the source of pointers during execution, we introduce a shadow store o, called *origin*, that associates heap locations with its content's source such that if $\mathcal{H}\, l = l'$ then the source of l' is $o\, l$.

$$
\begin{array}{lcl}
\textbf{Src},\ s & ::= & x \mid \delta \mid \mathit{unknown} \\
o \in \textbf{Origin} & = & \textbf{Loc} \to \textbf{Src} \\
\textbf{Action},\ a & ::= & \mathsf{push!}\ \delta\,(l, s) \mid \mathsf{pop!}\ \delta\,(l, s) \mid \mathsf{sync!}\ \delta \\
\textbf{InstrState},\ I & ::= & \langle \delta, \eta, o \rangle
\end{array}
$$

For actions, we also define the projection π_{path} that gives their path, along with π_{offs} and π_{src} that give the offset and source of the location (if any). Finally, steps are instrumented with a *nesting flag*. The flag indicates when execution enters a branch ($e = \blacksquare$) and otherwise ($e = \square$).

3.1.2 Local Rules. A reduction step from configuration γ to γ' is a judgement parameterized by the number of processes p and local process id $i \in \mathbb{P} = \{0, ..., p-1\}$ and written $i, p \vdash_1 \gamma; I \to_\alpha \gamma'; I', as, e$ where $\alpha \in \{\iota, \kappa\}$ is the *termination type*. When $\alpha = \iota$, local computation is suspended and requesting synchronization. Any continuation in γ' is executed in the next superstep. When $\alpha = \kappa$ the continuation is the next step of local computation in the same superstep. The initial and final instrumentation states are given by I and I', the list as is either a singleton action or empty (written ϵ) and e is the nesting flag.

The rules (Fig. 9) are fairly standard, except handling of termination types, parallel primitives and instrumentation. Assignments are allowed to allocated addresses (ASSIGN). Suspension is initiated in SYNC. Dynamic allocation in MALLOC is handled by the predicate *fresh*, whose definition is omitted. Intuitively, $fresh\, n\, \mathcal{H}\, \mathcal{H}'\, b$ holds if $\mathcal{H}'$ is identical to $\mathcal{H}$ except that when n is positive, the locations $(b, 0)...(b, n-1)$ are defined in the former and the base-address

$$\frac{}{i,p \vdash_1 \langle [\mathsf{sync}]^\ell, st\rangle; \langle \delta, \eta, o\rangle \to_\iota st; \langle (\delta \oplus 1), \eta, o\rangle, [\mathsf{sync}!\ \delta], \square}\ \textsc{sync}$$

$$\frac{\mathcal{P}[\![pe]\!]^p_i\ \mathcal{H} = l \in \mathbf{Dom}(\mathcal{H}) \quad \mathcal{A}[\![e]\!]^p_i\ \mathcal{H} = v \quad \mathcal{H}[l \leftarrow v] = \mathcal{H}'}{\begin{array}{l} i,p \vdash_1 \langle [pe := e]^\ell, (\mathcal{H}, rs, cs)\rangle; \langle \delta, \eta, o\rangle \to_\kappa \\ \qquad (\mathcal{H}', rs, cs); \langle (\delta \oplus 1), \eta, o[l \leftarrow src^p_i(e, \mathcal{H}, o)]\rangle, \epsilon, \square \end{array}}\ \textsc{assign}$$

$$\frac{\mathcal{P}[\![pe]\!]^p_i\ \mathcal{H} = l \in \mathbf{Dom}(\mathcal{H}) \quad fresh\,(\mathcal{A}[\![e]\!]^p_i\ \mathcal{H})\,\mathcal{H}\,\mathcal{H}'\,b}{\begin{array}{l} i,p \vdash_1 \langle [pe := \mathtt{malloc}\ e]^\ell, (\mathcal{H}, rs, cs)\rangle; \langle \delta, \eta, o\rangle \to_\kappa \\ \qquad (\mathcal{H}'[l \leftarrow (b,0)], rs, cs); \langle (\delta \oplus 1), \eta, o[l \leftarrow \delta]\rangle, \epsilon, \square \end{array}}\ \textsc{malloc}$$

$$\frac{\begin{array}{l} \mathcal{A}[\![e]\!]^p_i\ \mathcal{H} = (b,0) \in \mathbf{Dom}(\mathcal{H}) \quad b \neq b_{\mathsf{N}} \quad \nexists x, \sigma\ x = (b,0) \\ dealloc\,\mathcal{H}\,b\,\mathcal{H}' \end{array}}{\begin{array}{l} i,p \vdash_1 \langle [\mathtt{free}\ e]^\ell, (\mathcal{H}, rs, cs)\rangle; \langle \delta, \eta, o\rangle \to_\kappa \\ \qquad (\mathcal{H}', rs, cs); \langle (\delta \oplus 1), \eta, o\rangle, \epsilon, \square \end{array}}\ \textsc{free}$$

$$\frac{\begin{array}{l} i,p \vdash_1 \langle c_1, st\rangle; I \to_\alpha st'; \langle \delta'', \eta'', o'\rangle, as, e \\ (\delta', \eta') = trim\,init(c_2)\,(\delta'', \eta'') \end{array}}{i,p \vdash_1 \langle c_1; c_2, st\rangle; I \to_\alpha \langle c_2, st'\rangle; \langle \delta', \eta', o'\rangle, as, \square}\ \textsc{seq\_1}$$

$$\frac{\begin{array}{l} i,p \vdash_1 \langle c_1, st\rangle; I \to_\alpha \langle c_1', st'\rangle; \langle \delta', \eta', o'\rangle, as, e \\ \eta'' = init(c_2) : \eta' \text{ if } e = \blacksquare,\ \eta' \text{ oth.} \end{array}}{i,p \vdash_1 \langle c_1; c_2, st\rangle; I \to_\alpha \langle c_1'; c_2, st'\rangle; \langle \delta', \eta'', o'\rangle, as, \square}\ \textsc{seq\_2}$$

$$\frac{\mathcal{B}[\![b]\!]^p_i\ \mathcal{H} = \mathtt{tt}}{\begin{array}{l} i,p \vdash_1 \langle \mathtt{if}\,[b]^\ell\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2, (\mathcal{H}, rs, cs)\rangle; \langle \delta, \eta, o\rangle \to_\kappa \\ \qquad \langle c_1, (\mathcal{H}, rs, cs)\rangle; \langle (\delta \oplus 1) \odot (\mathsf{L}, 0), \eta, o\rangle, \epsilon, \blacksquare \end{array}}\ \textsc{if\_tt}$$

$$\frac{\mathcal{B}[\![b]\!]^p_i\ \mathcal{H} = \mathtt{ff}}{\begin{array}{l} i,p \vdash_1 \langle \mathtt{if}\,[b]^\ell\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2, (\mathcal{H}, rs, cs)\rangle; \langle \delta, \eta, o\rangle \to_\kappa \\ \qquad \langle c_2, (\mathcal{H}, rs, cs)\rangle; \langle (\delta \oplus 1) \odot (\mathsf{R}, 0), \eta, o\rangle, \epsilon, \blacksquare \end{array}}\ \textsc{if\_ff}$$

$$\frac{\mathcal{B}[\![b]\!]^p_i\ \mathcal{H} = \mathtt{tt}}{\begin{array}{l} i,p \vdash_1 \langle \mathtt{while}\,[b]^\ell\ \mathtt{do}\ c_1, (\mathcal{H}, rs, cs)\rangle; \langle \delta, \eta, o\rangle \to_\kappa \\ \quad \langle c_1; \mathtt{while}\,[b]^\ell\ \mathtt{do}\ c_1, (\mathcal{H}, rs, cs)\rangle; \langle (\delta \oplus 1) \odot (\mathsf{L}, 0), \eta, o\rangle, \epsilon, \blacksquare \end{array}}\ \textsc{wh\_tt}$$

$$\frac{\mathcal{B}[\![b]\!]^p_i\ \mathcal{H} = \mathtt{ff}}{\begin{array}{l} i,p \vdash_1 \langle \mathtt{while}\,[b]^\ell\ \mathtt{do}\ c_1, (\mathcal{H}, rs, cs)\rangle; \langle \delta, \eta, o\rangle \to_\kappa \\ \qquad (\mathcal{H}, rs, cs); \langle (\delta \oplus 1), \eta, o\rangle, \epsilon, \square \end{array}}\ \textsc{wh\_ff}$$

$$\frac{\mathcal{A}[\![e]\!]^p_i\ \mathcal{H} = l \quad rs + [\textsc{push}\ l] = rs' \quad s = src^p_i(e, \mathcal{H}, o)}{\begin{array}{l} i,p \vdash_1 \langle [\mathtt{push}\ e]^\ell, (\mathcal{H}, rs, cs)\rangle; \langle \delta, \eta, o\rangle \to_\kappa \\ \qquad (\mathcal{H}, rs', cs); \langle (\delta \oplus 1), \eta, o\rangle, [\mathsf{push}!\ \delta\,(l,s)], \square \end{array}}\ \textsc{push}$$

$$\frac{\mathcal{A}[\![e]\!]^p_i\ \mathcal{H} = l \quad rs + [\textsc{pop}\ l] = rs' \quad s = src^p_i(e, \mathcal{H}, o)}{\begin{array}{l} i,p \vdash_1 \langle [\mathtt{pop}\ e]^\ell, (\mathcal{H}, rs, cs)\rangle; \langle \delta, \eta, o\rangle \to_\kappa \\ \qquad (\mathcal{H}, rs', cs); \langle (\delta \oplus 1), \eta, o\rangle, [\mathsf{pop}!\ \delta\,(l,s)], \square \end{array}}\ \textsc{pop}$$

$$\frac{\begin{array}{l} (\mathcal{A}[\![e_1]\!]^p_i\ \mathcal{H}, \ldots, \mathcal{A}[\![e_5]\!]^p_i\ \mathcal{H}) = (j, (b, offs), l, n_1, n_2) \\ vs = [\mathcal{H}\,(b, offs + 0), \ldots, \mathcal{H}\,(b, offs + (n_2 - 1))] \end{array}}{\begin{array}{l} i,p \vdash_1 \langle [\mathtt{put}\ e_1\ e_2\ e_3\ e_4\ e_5]^\ell, (\mathcal{H}, rs, cs)\rangle; \langle \delta, \eta, o\rangle \to_\kappa \\ \qquad (\mathcal{H}, rs, cs + [\textsc{put}\ j\ vs\ l\ n_1]); \langle (\delta \oplus 1), \eta, o\rangle, \epsilon, \square \end{array}}\ \textsc{put}$$

Figure 9: Local semantics of commands

b does not occur in $\mathcal{H}$. If $n = 0$, then $b = \mathsf{N}$. The new memory's content is undetermined. This memory can be deallocated by `free` (rule FREE). Intuitively, the predicate $dealloc\,\mathcal{H}\,b\,\mathcal{H}'$ holds if $\mathcal{H}$ and $\mathcal{H}'$ are identical, except that all locations of base b is undefined in the latter. The rules PUSH, POP and PUT append register requests and communication requests to the state.

Simple instructions are instrumented to increment the path using the $\oplus$-operator, (e.g. rule SYNC). Conditionals increment and append the appropriate choice using the $\odot$-operator (e.g. rule IF_TT). Conditionals also set the nesting flag $\blacksquare$. When the reduction of the first component of a sequence sets this flag, the second component's label (as given by *init*) is pushed to the nesting stack (SEQ_2), as a reminder to the *trim*-function to remove a choice once the first component is fully reduced (SEQ_1). These operators and functions are defined in Fig. 8, where + is list concatenation.

The *src*-function, whose definition is omitted for lack of space, gives the source of arithmetic expressions that evaluates to locations. Intuitively, if the expression is the address-of operator, then the name of the operand is given; if the expression is a pointer expression, then the origin is consulted; etc. This function is used when updating the origin at assignments and allocations (e.g. rule ASSIGN), and consulted along with the path when generating actions (e.g. rule PUSH).

3.1.3 Multistep Relation. We write $i,p \vdash \gamma; I \to_\alpha \gamma'; I', as$ if there is a sequence of small-steps from γ such that γ' lacks a continuation or is suspended (see Fig. 10). Again, I and I' are the sequence's initial and final instrumentation. The actions are accumulated in the list *as*.

3.2 Global Semantics

The global semantics calculates the BSP computation's superstep structure: it initiates local computation in each process, treats the resulting communication and registration requests before executing following supersteps.

The global semantics operates over global configurations consisting of p-vectors of local configurations and a *registration sequence*: a list of location p-vectors, each of which constitutes a registration (Fig. 11). Registration sequences are manipulated by appending new lists of registrations and by popping registrations ($\ominus$-operator). Pop returns a dynamic error (Ω) if the popped registrations is not present in the sequence, or if the last appearance of each component is not at the same position.

We formalize how global computation applies the registration requests of a superstep to the registration sequence by the function C (Fig. 12). It splits and transposes the list-vector into vector-lists of registrations to remove (by applying $C_\circ$) and registrations to add (by applying $C_\bullet$). If two components of the transposition's operand do not have the same length, then the result is undefined and thus also C, following the intuition of rule (b) in Section 2.

3.2.1 Global Rules. The global semantics of BSPlite programs is given in Fig. 13. One global step by p processes from global configuration Γ to Γ', written $p \vdash \Gamma \to_\alpha \Gamma'\ ; A$, assumes processes have the same termination type α and that the registration sequence is not in an erroneous state. When $\alpha = \iota$, indicating that all local processes are requesting synchronization, the final configuration is

$$\frac{i,p \vdash_1 \langle c,st\rangle; I \to_\kappa \langle c',st'\rangle; I'', as, e \quad i,p \vdash \langle c',st'\rangle; I'' \to_\alpha \gamma; I', as'}{i,p \vdash \langle c,st\rangle; I \to_\alpha \gamma; I', (as +\!\!+ as')}\ \textsc{step} \qquad \frac{i,p \vdash_1 \langle c,st\rangle; I \to_\iota \gamma; I', as, e}{i,p \vdash \langle c,st\rangle; I \to_\iota \gamma; I', as}\ \textsc{susp} \qquad \frac{i,p \vdash_1 \langle c,st\rangle; I \to_\alpha st'; I', as, e}{i,p \vdash \langle c,st\rangle; I \to_\alpha st'; I', as}\ \textsc{stop}$$

Figure 10: Local multi-step semantics of BSPlite commands

$$\mathbf{Rs} = (\mathbf{Loc}^p)^* \qquad \text{(Registration sequence)}$$

$$\ominus : (\mathbf{Rs} \cup \{\Omega\}) \times \mathbf{Loc}^p \to (\mathbf{Rs} \cup \{\Omega\})$$

$$st \ominus \langle x_i\rangle_i = \begin{cases} st_1 +\!\!+ st_2 & \text{if } st = st_1 +\!\!+ [\langle x_i\rangle_i] +\!\!+ st_2 \text{ and} \\ & \neg(\exists i \in \mathbb{P}, k \in \mathbb{N}\,.\, \pi_i(st_2[k]) = x_i) \\ \Omega & \text{oth.} \end{cases}$$

$$lookup : \mathbf{Rs} \times \mathbb{P} \times \mathbb{P} \times \mathbf{Loc} \hookrightarrow \mathbf{Loc}$$

$$lookup\ st\ pid_1\ pid_2\ l_1 = l_2$$
if $\exists st_1, st_2\,.\,(st = st_1 +\!\!+ [\langle x_i\rangle_i] +\!\!+ st_2$ and
$x_{pid_1} = l_1$ and $x_{pid_2} = l_2$ and $\neg(\exists k \in \mathbb{N}\,.\,\pi_{pid_1}(st_2[k])) = l_1)$

$$\mathbf{GlobalConf},\ \Gamma ::= \mathbf{LocalConf}^p \times (\mathbf{Rs} \cup \{\Omega\})$$

Figure 11: BSPlite domain of global semantics

$$(\cdot)^T : \forall A, (A^*)^p \hookrightarrow (A^p)^*$$
$$\langle x_i : xs_i\rangle_i^T = \langle x_i\rangle_i : \langle xs_i\rangle_i^T$$
$$\langle\epsilon\rangle_i^T = \epsilon$$
$$C_\circ, C_\bullet : (\mathbf{Rs} \cup \{\Omega\}) \times (\mathbf{Loc}^p)^* \to (\mathbf{Rs} \cup \{\Omega\})$$
$$C_\circ\ st\ [L_1, ..., L_n] = (st \ominus L_1) \ominus L_2 \ominus ... \ominus L_n \text{ if } st \neq \Omega,\ \Omega \text{ oth.}$$
$$C_\bullet\ st\ Ls_\bullet = st +\!\!+ Ls_\bullet \text{ if } st \neq \Omega,\ \Omega \text{ oth.}$$
$$C : \mathbf{Rs} \times (\mathbf{RReq}^*)^p \to (\mathbf{Rs} \cup \{\Omega\})$$
$$C\ st\ \langle rs_i\rangle_i = C_\bullet\ Ls'_\bullet\ (C_\circ\ st\ Ls'_\circ) \text{ if } (Ls'_\circ, Ls'_\bullet) = (Ls_\circ^T, Ls_\bullet^T),\ \Omega \text{ oth.}$$
where $Ls_\circ, Ls_\bullet = \langle[l \mid \textsc{pop}\ l \in rs_i]\rangle_i, \langle[l \mid \textsc{push}\ l \in rs_i]\rangle_i$

Figure 12: The function C formalizes the effect of registration requests on a registration sequence.

obtained by executing and removing the obtained register requests (by C) and communication requests (by $\mathcal{D}$). The function $\pi_{rs}(\gamma)$ retrieves the register requests from γ and $reset(\gamma)$ returns γ with register and communication requests removed. We do not detail $\mathcal{D}$ further, but note that it uses *lookup* on the registration sequence to route communication. The function *CommI*, whose definition is omitted here, instruments communication so that the source of overwritten pointers is set to *unknown*. The p-vector A collects each process's action trace. We finally define $Reach_p$ as the reflexive closure of global steps.

Modelling the SPMD-nature of BSPlib, an initial global configuration in BSPlite, $\Gamma_c = (\langle\langle c, (\mathcal{H}_0, \epsilon, \epsilon)\rangle; \langle(0, \epsilon), \epsilon, o_0\rangle\rangle_i, \epsilon)$, replicates a continuation c, the initial heap, request lists and registration sequence. It is instrumented with an empty path, empty nesting stack and initial origin $o_0 = (\lambda l.unknown)$.

The trace vectors resulting from executing the running examples with the instrumentation is given in Fig. 14.

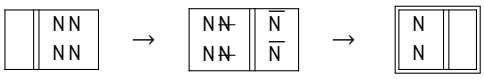
Lemma 1. *If $Reach_p(\Gamma, \Gamma', A)$ then the same source $s \neq unknown$ never appears twice in the same component of A associated with two locations of different base.*

Proof. See Appendix A.1. □

We call such vectors **consistent**. Intuitively, the same location may be allocated twice (e.g. `N`), but the path of each execution step is unique and so also the source by extension. The semantics also ensures that the source of each local variable x is always associated with the fixed location of that variable, i.e. $\sigma\ x$.

4 INSTRUMENTATION & CORRECTNESS

To define correctness we reason directly on trace vectors. We impose a slight restriction compared to the BSPlib standard. Consider an execution of Example 4 where `malloc` returns `N` twice for both processes (this could be caused by space constraints):



There is no dynamic error, but arguably by "accident": the popped registration is probably not the one intended by the user and as the originally considered execution shows, an error could occur. We impose an additional **source restriction**. Whenever a register is popped, then the location in the **pop**-request and the one in the registration sequence must come from the same known *source*: intuitively, the same local variable or same instance of dynamic allocation. This restriction ensures that a correct execution stays correct independently on the behaviour of `malloc`.

4.1 Correctness

Trace vectors can be seen as programs with actions as instructions. We give a local semantics of traces as functions over a stack-like state of locations, defining a local view of correctness, and then a global semantics for trace vectors. Anticipating our simplified programming model for guaranteeing for correctness, this will enable a definition of *global correctness from a local perspective.*

4.1.1 Local Correctness. Local action trace semantics is defined by $\mathcal{A}![\![\cdot]\!]$ (Fig. 15) that symbolically executes the trace, tracks the source of pushed locations, and verifies that each popped location has been pushed and committed, and, by the source restriction, with the same source used in the `pop!`-action. Local correctness of as amounts to $\mathcal{A}![\![as]\!] = \mathtt{tt}$. $\mathcal{A}![\![\cdot]\!]$ is defined by $\mathcal{A}!_1$ giving the effect of one action on the state and $\mathcal{A}!_{ss}$ giving the effect of all actions of one superstep. Finally $\mathcal{A}!'$ gives the effect of the whole trace. Any actions in the last superstep has no effect and so as_i is ignored by $\mathcal{A}!'$. By extension, a trace vector is locally correct if each component is.

4.1.2 Global Correctness. As the executions of Example 5 and 3 demonstrate, local correctness of a trace vector does not amount to its global correctness. They must also make compatible sequences of actions (unlike Example 5), and pops must occur at the same level in the registration sequence (unlike Example 3).

$$\frac{st_\Omega \neq \Omega \quad \forall i \in \mathbb{P}\,.\, i, p \vdash \gamma_i; I_i \rightarrow_\iota \gamma_i'; I_i', as_i \quad \mathcal{D}(\langle\gamma_i'\rangle_i, st_\Omega, \langle\gamma_i''\rangle_i) \quad CommI((\langle\gamma_i'; I_i'\rangle_i, st_\Omega)) = \langle I_i''\rangle_i}{p \vdash (\langle\gamma_i; I_i\rangle_i, st_\Omega) \rightarrow_\iota (\langle reset(\gamma_i''); I_i''\rangle_i, C(st_\Omega, \langle\pi_{rs}(\gamma_i')\rangle_i))\ ;\ \langle as_i\rangle_i}$$

$$\frac{st_\Omega \neq \Omega \quad \forall i \in \mathbb{P}\,.\, i, p \vdash \gamma_i; I_i \rightarrow_\kappa \gamma_i'; I_i', as_i}{p \vdash (\langle\gamma_i; I_i\rangle_i, st_\Omega) \rightarrow_\kappa (\langle\gamma_i'; I_i'\rangle_i, st_\Omega)\ ;\ \langle as_i\rangle_i} \qquad \frac{}{Reach_p(\Gamma, \Gamma, \langle\epsilon\rangle_i)} \qquad \frac{Reach_p(\Gamma, \Gamma'', A) \quad p \vdash \Gamma'' \rightarrow_\alpha \Gamma'\ ;\ A'}{Reach_p(\Gamma, \Gamma', A + A')}$$

Figure 13: Global big-step semantics of BSPlite programs and the reachability relation

Program	Action trace vector	LC	TA	SA	SF	GC
Example 1	⟨ [push! (y, y), push! (z, z), push! (y, y), push! (x, x), sync!, pop! (y, y), sync!], [push! (y, y), push! (z, z), push! (y, y), push! (x, x), sync!, pop! (y, y), sync!] ⟩	✓ ✓	✓	✓	✓	✓
Example 2	⟨ [push! (x, x), pop! (x, x), sync!], [push! (x, x), pop! (x, x), sync!] ⟩	✗ ✗	✓	✓	✗	✗
Example 3	⟨ [push! (y, y), push! (y, y), sync!, pop! (y, y), sync!], [push! (y, y), push! (x, x), sync!, pop! (y, y), sync!] ⟩	✓ ✓	✓	✗	✗	✗
Example 4	⟨ [push! (N, (0, ϵ)), push! (N, (1, ϵ)), sync!, pop! (N, (0, ϵ)), sync!], [push! (l_1, (0, ϵ)), push! (l_2, (1, ϵ)), sync!, pop! (l_1, (0, ϵ)), sync!] ⟩	✗ ✓	✓	✓	✗	✗
Example 5	⟨ [sync! (1, ϵ)], [push! (1, [(R, 0)]) (x, x), sync! (1, ϵ)] ⟩	✓ ✓	✗	✗	✗	✗
Example 6	⟨ [push! (0, ϵ) (y, y), sync! (1, ϵ), pop! (2, [(L, 0)]) (y, y), push! (2, [(L, 1)]) (x, x), sync! (2, ϵ)], [push! (0, ϵ) (y, y), sync! (1, ϵ), push! (2, [(R, 0)]) (x, x), pop! (2, [(R, 1)]) (y, y), sync! (2, ϵ)] ⟩	✓ ✓	✗	✗	✗	✓

Figure 14: Trace vectors resulting from executing running examples with $p = 2$. The right part of the table evaluates the traces' local correctness (LC), the vector's textual alignment (TA), source alignment (SA) and safety (SF), and global correctness (GC) as defined in Sections 4 and 5. Here, x is the location of variable x and x its source and l_i is the ith address returned by `malloc`. The action paths have been omitted in all examples but Example 5 and Example 6.

$r \in \textbf{Map} = (\textbf{Loc} \rightarrow \textbf{Src}^*)$

$\mathcal{A}!_1[\![\cdot]\!] : \textbf{Action} \rightarrow \textbf{Map} \hookrightarrow \textbf{Map}$

$\mathcal{A}!_1[\![\text{pop! } \delta\ (l, s)]\!]\ r = r[l \leftarrow ss]$ if $(r[l] = s' : ss) \wedge s \simeq s'$, undef oth.

$\mathcal{A}!_1[\![\text{push! } \delta\ (l, s)]\!]\ r = r[l \leftarrow s : r[l]]$

$\mathcal{A}!_1[\![\text{sync! } \delta]\!]\ r = \text{undef}$

$\mathcal{A}!_{ss}[\![\cdot]\!] : \textbf{Action}^* \rightarrow \textbf{Map} \hookrightarrow \textbf{Map}$

$\mathcal{A}!_{ss}[\![as]\!]\ r = fold\ \mathcal{A}!_1[\![\cdot]\!]\ as'\ r$

where $as' = [a \mid a = \text{pop!}\ \_\ \_ \in as] + [a \mid a = \text{push!}\ \_\ \_ \in as]$

$\mathcal{A}!'[\![\cdot]\!] : \textbf{Action}^* \hookrightarrow \textbf{Map}$

$\mathcal{A}!'[\![as]\!] = fold\ \mathcal{A}!_{ss}[\![\cdot]\!]\ [as_1, \ldots, as_{n-1}]\ (\lambda l.\epsilon)$

where $as_1 + [\text{sync! } \delta_1] + \ldots + [\text{sync! } \delta_{n-1}] + as_n = as$

such that $\forall 1 \leq i \leq n, (\text{sync! } \_) \notin as_i$

$\mathcal{A}![\![\cdot]\!] : \textbf{Action}^* \rightarrow \textbf{Bool}$

$\mathcal{A}![\![as]\!] = \texttt{tt}$ if $\mathcal{A}!'[\![as]\!]$ is defined, `ff` oth.

$s_1 \simeq s_2 \Leftrightarrow unknown \notin \{s_1, s_2\} \wedge s_1 = s_2$

$fold\ f\ [a_1, ..., a_n]\ r = (f\ a_n \circ \ldots \circ f\ a_1)\ r$

Figure 15: Local correctness of an action trace

$m \in \textbf{Matching} = (\mathbb{N} \times \mathbb{N}^*)^*$

$r \in \textbf{MapI} = \textbf{Loc} \rightarrow (\mathbb{N} \times \textbf{Src})^*$

$\mathcal{M}!_1 : \textbf{Action} \rightarrow (\mathbb{N} \times \mathbb{N}^* \times \textbf{MapI}) \hookrightarrow (\mathbb{N} \times \mathbb{N}^* \times \textbf{MapI})$

$\mathcal{M}!_1[\![\text{pop! } \delta\ (l, s)]\!]\ (k, ps, r) =$
$(k, k' : ps, r[l \leftarrow is])$ if $r[l] = (k', s') : is \wedge s \simeq s'$, undef oth.

$\mathcal{M}!_1[\![\text{push! } \delta\ (l, s)]\!]\ (k, ps, r) = (k + 1, ps, r[l \leftarrow (k, s) : r[l]])$

$\mathcal{M}!_1[\![\text{sync! } \delta]\!]\ (k, ps, r) \quad = \text{undef}$

$\mathcal{M}!_{ss} : \textbf{Action}^* \rightarrow (\textbf{Matching} \times \textbf{MapI}) \hookrightarrow (\textbf{Matching} \times \textbf{MapI})$

$\mathcal{M}!_{ss}[\![as]\!]\ (m, r) = \text{let } k' = k \text{ if } m = m' + [(k, \_)],\ 0 \text{ oth. in}$
$\text{let } (k'', ps, r') = fold\ \mathcal{M}!_1[\![\cdot]\!]\ as'\ (k', \epsilon, r) \text{ in}$
$(m + [(k'', ps)], r')$

where $as' = [a \mid a = \text{pop!}\ \_\ \_ \in as] + [a \mid a = \text{push!}\ \_\ \_ \in as]$

$\mathcal{M}!' : \textbf{Action}^* \hookrightarrow (\textbf{Matching} \times \textbf{MapI})$

$\mathcal{M}!'[\![as]\!] = fold\ \mathcal{M}!_{ss}[\![\cdot]\!]\ [as_1, \ldots, as_{n-1}]\ (\epsilon, \lambda l.\epsilon)$

where $as_1 + [\text{sync! } \delta_1] + \ldots + [\text{sync! } \delta_{n-1}] + as_n = as$

$\mathcal{M}! : (\textbf{Action}^*)^{\mathcal{P}} \rightarrow \textbf{Bool}$

$\mathcal{M}![\![\langle as_i\rangle_i]\!] = \texttt{tt}$ if $\exists ms, \forall i \in \mathbb{P}. \mathcal{M}!'[\![as_i]\!](\epsilon, \lambda l.\epsilon) = (ms, \_)$, `ff` oth.

Figure 16: Global correctness of a trace vector

To capture these requirements, we abstract the effect of a trace into a *matching* (see Fig. 16). The matching is one pair (k, ps) per superstep, where k counts the total number of pushes at the end of the superstep and ps is a list containing the index of each push removed by a pop in the superstep. $\mathcal{M}!$ extends $\mathcal{A}!$ to index the pushes added to the state, and to return the matching of locally correct traces. We then define a trace vector as being globally correct when each component has the same matching.

Before going further, we reconnect global correctness of traces with the semantics of BSPlite, and confirm that programs with correct traces do not have register errors:

Theorem 1. *If* $Reach_p(\Gamma_c, (\langle \gamma_i; I_i \rangle_i, st_\Omega), A)$ *and* $\mathcal{M}![\![A]\!] = \texttt{tt}$ *then* $st_\Omega \neq \Omega$.

Proof. See Appendix A.2. □

This characterization of correctness is independent of the underlying language. However, we have yet to simplify the task of writing correct programs. In the next section we define our simplified programming model that guarantees correctness.

5 SIMPLIFIED PROGRAMMING MODEL

Intuitively, the simplified programming model for registration imposes 3 conditions: (1) collective calls to sync, push and pop should be *textually aligned* [5], i.e. originate from the same textual position in each process; (2) the argument of collective calls to push and pop should be *source aligned*, i.e., refer to the same memory object: the same local variable or the same instance of dynamically allocated memory in all processes; (3) the sequence of actions of each process should be locally correct. These conditions in conjunction guarantee global correctness.

Consider again the running examples. Example 2 is not locally correct in any process, and hence disqualified. Due to the source restriction, Example 4 is not locally correct in process 0. In Example 3 the memory object referred to by q is not the same in each process, and hence the second push is not source aligned. Examples 5 and 6 are not textually aligned. Only Example 1 follows the model. We now formalize these notions using trace vectors.

Definition 1. A trace vector $\langle as_i \rangle_i$ is **textually aligned** if each component has the same length and the same path at each position:

$$\forall i, j \in \mathbb{P}.\ |as_i| = |as_j| = m \wedge$$
$$\forall 0 \leq k < m.\ \pi_{path}(as_i[k]) = \pi_{path}(as_j[k])$$

□

Definition 2. A textually aligned trace vector $\langle as_i \rangle_i$ is **source aligned** if all components have the same (known) source and offset at each position:

$$\forall i, j \in \mathbb{P}.\ \forall 0 \leq k < |as_i|.as_i[k] \neq \texttt{sync!}\_ \Rightarrow$$
$$\pi_{src}(as_i[k]) \simeq \pi_{src}(as_j[k]) \wedge \pi_{offs}(as_i[k]) = \pi_{offs}(as_j[k])$$

□

Definition 3. A consistent trace vector A that is textually aligned, source aligned and locally correct is called **safe**. □

By extension, a program in our model is one that only produces safe trace vectors. The trace vectors of the running examples are evaluated against these conditions in Fig. 14. As expected, the intuitions given above are consistent with the formalization. Finally, we prove that the programming model guarantees global correctness:

Theorem 2. *If A is safe then* $\mathcal{M}![\![A]\!] = \texttt{tt}$.

Proof. See Appendix A.3. □

This condition is inspired by observations of realistic and correct BSPlib code. We therefore argue that in addition to ensuring correctness, the condition is sufficiently permissive and coherent with the programmer's intuition of correctness.

6 CONCLUSION

This article argues that static analysis can be used to combine the benefits of writing parallel programs in a dedicated language, whose semantics matches the underlying parallel model, with the benefits of writing them in a more widely applicable general purpose language extended with parallel libraries. To illustrate our argument, we study errors caused by registration in BSPlib, a C library enabling Bulk Synchronous Parallelism. Registration is used to create associations between local and remote memory, but can provoke errors if done incorrectly. We have formalized BSPlib with registration, characterized correct executions and given a sufficient condition for correctness strong enough to capture realistic programs. Our next step is to develop a static analysis targeting this condition, and evaluate it on real-world BSPlib programs.

REFERENCES

[1] A. Aiken and D. Gay. 1998. Barrier Inference. In Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98). ACM, New York, NY, USA, 342–354. https://doi.org/10.1145/268946.268974

[2] O. Ballereau, F. Loulergue, and G. Hains. 1999. High-level BSP Programming: BSML and BSλ. In Proceedings of the first Scottish Functional Programming Workshop (Technical Report), P. Trinder and G. Michaelson (Eds.). Heriot-Watt University, Edinburgh, 43–52.

[3] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping. 2003. The Paderborn University BSP (PUB) library. Parallel Comput. 29, 2 (Feb. 2003), 187–207. https://doi.org/10.1016/S0167-8191(02)00218-1

[4] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. 2010. Introducing OpenSHMEM: SHMEM for the PGAS community. In Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model. ACM, New York, NY, USA, 2. http://dl.acm.org/citation.cfm?id=2020375

[5] F. Dabrowski. 2018. Textual Alignment in SPMD Programs. In Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC '18). ACM, New York, NY, USA, 1046–1053. https://doi.org/10.1145/3167132.3167254

[6] J. Fortin and F. Gava. 2016. BSP-Why: A Tool for Deductive Verification of BSP Algorithms with Subgroup Synchronisation. International Journal of Parallel Programming 44, 3 (June 2016), 574–597. https://doi.org/10.1007/s10766-015-0360-y

[7] F. Gava and J. Fortin. 2008. Formal Semantics of a Subset of the Paderborn's BSPlib. In Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies, 2008. PDCAT 2008. IEEE Press, Piscataway, NJ, USA, 269–276. https://doi.org/10.1109/PDCAT.2008.43

[8] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling. 1998. BSPlib: The BSP Programming Library. Parallel Comput. 24, 14 (Dec. 1998), 1947–1980. https://doi.org/10.1016/S0167-8191(98)00093-3

[9] Message Passing Interface Forum. 2012. MPI: A Message-Passing Interface Standard Version 3.0. http://www.mpi-forum.org/docs/mpi-3.0

[10] D. M. Ritchie, B. W. Kernighan, and M. E. Lesk. 1988. The C Programming Language. Prentice Hall, Englewood Cliffs, NJ, USA.

[11] K. Siddique, Z. Akhtar, E. J. Yoon, Y. Jeong, D. Dasgupta, and Y. Kim. 2016. Apache Hama: An emerging bulk synchronous parallel computing framework for big data applications. IEEE Access 4 (2016), 8879–8887.

[12] J. Tesson and F. Loulergue. 2008. Formal Semantics of DRMA-Style Programming in BSPlib. In Parallel Processing and Applied Mathematics, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski (Eds.). Vol. 4967. Springer Berlin Heidelberg, Berlin, Heidelberg, 1122–1129. http://link.springer.com/10.1007/978-3-540-68111-3_119

[13] L. G. Valiant. 1990. A Bridging Model for Parallel Computation. Commun. ACM 33, 8 (Aug. 1990), 103–111. https://doi.org/10.1145/79173.79181

[14] A.N. Yzelman and Rob H. Bisseling. 2012. An object-oriented bulk synchronous parallel library for multicore programming. Concurrency and Computation: Practice and Experience 24, 5 (April 2012), 533–553. https://doi.org/10.1002/cpe.1843

A PROOF SKETCHES

This section sketches the proofs of Lemma 1 and Theorems 1 and 2. While the full proofs have been developed, we do not include them here due to their length and lack of time. Instead, we defer their typesetting in a technical report to future work.

A.1 Proof Sketch For Lemma 1

We recall Lemma 1:

Lemma 1. *If $Reach_p(\Gamma, \Gamma', A)$ then the same source $s \neq unknown$ never appears twice in the same component of A associated with two locations of different base.*

Before proving this lemma, we show an auxiliary fact. Namely, that the path of each local execution step is a "fresh" path that has not appeared before. We formalize freshness by establishing a strict order on paths with the intention that, $\delta_1 < \delta_2$ if δ_1 is produced "before" δ_2 in the semantics and thus that δ_2 is fresh. We then show, by a standard rule induction on the local semantics, that the initial and final path of each local step is strictly ordered.

The proof of Lemma 1 consists of showing that for each instrumented state $\mathcal{H}, o$ and as in a local execution there exists a partial mapping ρ from sources to bases such that for each location-source pair (l, s) where $s \neq unknown$ and either

(1) (l, s) appears in as, or
(2) for some location l', $\mathcal{H}\, l' = l$ and $o\, l = s$, or
(3) for some variable x, $s = x$ and $l = \sigma\, x$.

we have $\rho\, s = \pi_{base}(l)$. We then say that the instrumented state is ρ-consistent.

Clearly from (1) follows that each source that appears in as is associated with at most one base, by the single-valuedness of ρ, as Lemma 1 requires.

We first show that the initial state of a local process is ρ-consistent with ρ being the empty map. We then show by rule induction on the local semantics that an appropriate ρ' exists that preserves ρ-consistent of the instrumented state.

- For all cases but MALLOC and MALLOC_NULL we take $\rho' = \rho$. In the cases MALLOC and MALLOC_NULL we take $\rho' = \rho[(\delta, \ell) \leftarrow \pi_{base}(l')]$ where l' is the newly allocated location and δ the current path.
  Since the path is fresh, (δ, ℓ) cannot already be defined in ρ. ρ'-consistency of the new instrumented state follows by considering any (l, s) pair that occurs in the new instrumented state.
- If a new action is added to the trace (rules PUSH, POP and SYNC) and it contains the location-source pair (l, s) then $\rho\, s = \pi_{base}(l)$ follows by the consistency of the instrumented state.
- For the case ASSIGN, we prove an additional fact: in a ρ-consistent instrumented state, the source given by the *src*-function for a location applied to ρ returns the base of that same location. Using this fact, we show the instrumented state resulting from the assignment ρ-consistent.
- Remaining cases are trivial.

To conclude, we show that ρ-consistency follows for multi-step execution by standard rule induction. Then, that the concept can be extended to global executions by showing the existence of p-vectors of mappings $\langle \rho_i \rangle_i$ such that the state of each processor i is ρ_i-consistent. We show that all reachable global configurations are $\langle \rho_i \rangle_i$-consistent. The result follows.

A.2 Proof Sketch For Theorem 1

We recall Theorem 1:

Theorem 1. *If $Reach_p(\Gamma_c, (\langle \gamma_i; I_i \rangle_i, st_\Omega), A)$ and $\mathcal{M}![\![A]\!] = \texttt{tt}$ then $st_\Omega \neq \Omega$.*

The proof strategy consists of establishing a correspondence between a p-vector of the maps in **MapI** that $\mathcal{M}!$ acts on and the registration sequences of **Rs** in the concrete semantics. In particular, the correspondence consists of a function from a p-vector of **MapI** to an element of **Rs**. Thus by definition, the correspondence only holds when the registration sequence is not in the error state.

Trivially, the initial empty registration sequence and an vector of empty maps correspond to each other.

Before considering reachable configurations, we establish an auxiliary fact relating $\mathcal{M}!_{ss}$ and C. First, the former is defined for action traces and the latter over vectors of lists of registration requests. We define a function *reqsToTrace* from registration requests to action traces in the obvious way.

We then show that if st and a vector of maps $\langle r_i \rangle_i$ correspond, if $\langle rs_i \rangle_i$ is a vector of registration requests such as resulting from the execution of one superstep, then the vector of maps that results from applying $\mathcal{M}!_{ss}\,(reqsToTrace(rs_i))$ pointwise to $\langle r_i \rangle_i$ conserves the correspondence with $C\ st\ \langle rs_i \rangle_i$. This follows from a showing similar relationship between $C_\circ, C_\bullet$ and $\mathcal{M}!_1$ followed by standard induction.

We then proceed by rule induction to show that all reachable configurations preserve the correspondence between registration sequence and the map-vector obtained by applying $\mathcal{M}!'$ pointwise to the reached the action vector. The reflection case of $Reach_p$ is trivial. In the step case of $Reach_p$ we consider the termination type α of the global step. In the case $\alpha = \kappa$, then the correspondence trivially holds. When $\alpha = \iota$ we note that the action trace of each process i is on the form $as_i = as_i' +\!\!+ [\texttt{sync!}\ \_]$ such that as_i' does not contain any synchronization actions, and that $as_i' = reqsToTrace(rs_i)$ where rs_i correspond to the list of register requests engendered by process i and *reqsToTrace*, three facts which follows from a standard rule induction on the local semantics. Preservation of the correspondence now follows since it is preserved by $\mathcal{M}!_{ss}$ and C.

As the correspondence is preserved by all reachable configurations with an action vector for which $\mathcal{M}!$ holds, and since the correspondence by definition only holds if the registration sequence of that configuration is no in the error state, we have the desired result.

A.3 Proof Sketch For Theorem 2

We recall Theorem 2:

Theorem 2. *If A is safe then $\mathcal{M}![\![A]\!] = \texttt{tt}$.*

We first show that a trace that is locally correct also has a matching, which is simple given the similarity of the functions $\mathcal{A}!$ and $\mathcal{M}!$.

It then suffices to show that that all pairs of action traces that are ρ-consistent, textually aligned, source aligned and where both traces have a matching actually have the same matching, implying

that all traces in the vector have the same matching and thus that the vector is globally correct.

This is done by showing that two such traces are equivalent modulo bases, and thus exactly equal if ρ of one trace is used to substitute each source-(base-offset) pair $(s, (b, o))$ appearing in the other with $(s, (\rho\, s, o))$. This follows from the definition of

(1) textual alignment: from which we know that the traces have the same length and their action have pointwise the same path. We can then show that two actions with the same path from the same program must be the same type of action (since they originate from the same instruction).
(2) source alignment: which ensures that the two actions at the same position in the traces have the same source and the same offset.

Hence, only bases differ between the traces.

From there we show that applied to $\mathcal{M}!$, two such traces have the same matching. Specifically, we show that a correspondence can be established between two elements of **MapI** that are in this way equivalent modulo bases. Trivially, two empty **MapI** are equivalent modulo base. We then show that $\mathcal{M}!_1$, applied to actions and maps equivalent modulo bases return the same matching and new maps that are also equivalent modulo bases. The result then follows from by induction on the length of the trace.