A Verification Technique for Deterministic Parallel Programs (extended version)

S. Darabi, S.C.C. Blom, and M. Huisman

University of Twente, the Netherlands

Abstract. A commonly used approach to develop parallel programs is to augment a sequential program with compiler directives that indicate which program blocks may potentially be executed in parallel. This paper develops a verification technique to prove correctness of compiler directives combined with functional correctness of the program. We propose syntax and semantics for a simple core language, capturing the main forms of deterministic parallel programs. This language distinguishes three kinds of basic blocks: parallel, vectorized and sequential blocks, which can be composed using three different composition operators: sequential, parallel and fusion composition. We show that it is sufficient to have contracts for the basic blocks to prove correctness of the compiler directives, and moreover that functional correctness of the sequential program implies correctness of the parallelized program. We formally prove correctness of our approach. In addition, we define a widely-used subset of OpenMP that can be encoded into our core language, thus effectively enabling the verification of OpenMP compiler directives, and we discuss automated tool support for this verification process.

1 Introduction

A common approach to handle the complexity of parallel programming is to write a sequential program augmented with parallelization compiler directives that indicate which part of code might be parallelized. A parallelizing compiler consumes the annotated sequential program and automatically generates a parallel version. This parallel programming approach is often called *deterministic parallel programming*, as the parallelization of a deterministic sequential program augmented with correct compiler directives is always deterministic. Deterministic parallel programming is supported by different languages and libraries such as OpenMP [2] and is often used for financial and scientific applications [17,1,13,5].

Although it is relatively easy to write parallel programs in this way, careless use of compiler directives can easily introduce data races and consequently non-deterministic program behaviour. This paper proposes a static technique to prove that parallelization as indicated by the compiler directives does not introduce such non-determinism. Moreover it also shows how our technique reduces functional verification of the parallelized program to functional verification of the sequential program. We develop our verification technique over a core deterministic parallel programming language called PPL (for Parallel Programming

Language). To show practical usability of our approach, we present how a commonly used subset of OpenMP can be encoded into PPL and then be verified in our approach. We also discuss tool support for this process.

In essence, PPL is a language for the composition of code blocks. We identify three kinds of *basic blocks*: a *parallel block*, a *vectorized block* and a *sequential block*. Basic blocks are composed by three binary block composition operators: *sequential composition*, *parallel composition* and *fusion composition* where the fusion composition allows two parallel basic blocks to be merged into one. An operational semantics for PPL is presented.

Our verification technique requires each basic block to be specified by an *iteration contract* [8] that describes which memory locations are read and written by a thread. Moreover, the program itself should be specified by a global contract. To verify the program, we show that the block compositions are memory safe (i.e. data race free) by proving that for all *independent iterations* (i.e. the iterations that might run in parallel) all accesses to shared memory are non-conflicting, meaning that they are disjoint or they are read accesses. If all block compositions are memory safe, then it is sufficient to prove that the sequential composition of all the basic blocks w.r.t. program order is memory safe and functionally correct, to conclude that the parallelized program is functionally correct.

The main contributions of this paper are the following:

- A core language, PPL, and an operational semantics which captures the main forms of parallelization constructs in deterministic parallel programming.
- A verification approach for reasoning about data race freedom and functional correctness of PPL programs.
- A soundness proof that all verified PPL programs are indeed data race free and functionally correct w.r.t. their contracts.
- Tool support that addresses the complete process of encoding of OpenMP into PPL and verification of PPL programs.

This paper is organized as follows. After some background information, Section 3 explains syntax and semantics of PPL. Section 4 presents our verification technique for reasoning about PPL programs and also discusses soundness of our verification approach. Section 5 explains how our approach is applied to verification of OpenMP programs. Finally, we conclude with related and future work.

2 Background

We present some background information on OpenMP, Permission-based Separation Logic and the notion of iteration contract.

2.1 OpenMP

This section illustrates the most important OpenMP features by an example. We verify this example later in Section 5 where the program contract and the iteration contracts are added. The example in Fig. 1 is a sequential C program augmented by OpenMP compiler directives (*pragmas*). The pivotal parallelization annotation in OpenMP is *omp parallel* which determines the parallelizable

```
1  /*@ Program Contract (PC) @*/
2  void adm(int L,int a[],int b[],int c[],
3  int d[]){
4  #pragma omp parallel {
5  #pragma omp for schedule(static) nowait
6  for(int i=0;i<L;i++) //Loop L1
7  /*@ Iteration  Contract 1 (IC1) @*/
8      { c[i]=a[i];  }
9  #pragma omp for schedule(static) nowait
10 for(int i=0;i<L;i++) //Loop L2
11 /*@ Iteration  Contract 2 (IC2) @*/
12     { c[i]=c[i]+b[i];  }
13 #pragma omp for
14 for(int i=0; i<L; i++) //Loop L3
15 /*@ Iteration  Contract 3 (IC3)  @*/
16     { d[i]=a[i]*b[i];  }
17 }}
```

Fig. 1. OpenMP Example

code block (called *parallel region*). Threads are forked upon entering a parallel region and joined back into a single thread at the end of the region.

The example shows a parallel region with three for-loops L1, L2, and L3. The loops are marked as *omp for* meaning that they are parallelizable (i.e. their iterations are allowed to be executed in parallel). To precisely define the behaviour of threads in the parallel region, *omp for* annotations are extended by *clauses*. For example the combined use of the *nowait* and *schedule(static)* clauses indicates that it is safe to *fuse* the parallel loops L1 and L2, meaning that the corresponding iterations of L1 and L2 are executed by the same thread without waiting. The clause *nowait* implies that it is safe to eliminate the implicit barrier at the end of *omp for*. The clause *schedule(static)* ensures that the OpenMP compiler assigns the same thread to corresponding iterations of the loops. In OpenMP all variables which are not local to a parallel region are considered as *shared* by default unless they are explicitly declared as private (using *private* clause) when they are passed to a parallel region.

2.2 Syntax and Semantics of Specification Language

Our verification technique is based on Permission-based Separation Logic [12,9]. Separation logic [20] is an extension of Hoare logic [15], originally proposed to reason about pointer programs. Separation logic is also suited for modular verification of concurrent programs [18]: two threads working on disjoint parts of the heap do not interfere and thus can be verified in isolation.

The basis of our specification language is a separation logic for C [22], extended with fractional permissions [12,9] to denote the right to either read from or write to a location. Any fraction in the interval $(0, 1)$ denotes a *read permission*, while 1 denotes a *write permission*. Permissions can be split and combined, but soundness of the logic prevents the sum of the permissions for a location over all threads to exceed 1. This guarantees that if permission specifications can be verified, the program is data race free. The set of permissions that a thread holds are often called its *resources*. In earlier work, we have shown that this logic is suitable to reason about kernel programs [7] and parallel loops [8].

Formulas F in our logic are built from first-order logic formulas b, permission predicates $\mathsf{Perm}(l, f)$, conditional expressions $(\cdot ? \cdot : \cdot)$, separating conjunction $\star$,

and universal separating conjunction $\bigstar$ over a finite set I. The syntax of formulas is formally defined as follows:

$$F ::= b \mid \mathsf{Perm}(l, f) \mid b?F : F \mid F \star F \mid \bigstar_{i \in I} F(i)$$

where b is a side-effect free boolean expression, l is a side-effect free expression of type location, and f is a side-effect free expression of type fraction.

To define the semantics of formulas, we assume the existence of the following domains: Loc, the set of memory locations, $\mathsf{VarName}$, the set of variable names, Val, the set of all values, which includes the memory locations, and Frac, the set of fractions ($[0, 1]$). A heap is a map from locations to values $h : \mathsf{Loc} \to \mathsf{Val}$. A heap mask is a map from locations to fractions $\pi : \mathsf{Loc} \to \mathsf{Frac}$ with unit element $\pi_0 : l \mapsto 0$. A store is function from variable names to values: $\sigma : \mathsf{VarName} \to \mathsf{Val}$.

Our semantics mixes concepts of Implicit Dynamic Frames [21] and Separation Logic with fractional permissions: formulas can access the heap directly, fractional permissions to access the heap are provided by the Perm predicate. Moreover, a strict form of self-framing is enforced.

The semantics of expressions depends on a store, a heap, and a heap mask and yields a value: $\sigma, h, \pi \, [e\rangle \, v$. The store and the heap are used to determine the value, the heap mask is used to determine if the expression is correctly framed. For example, the rule for array access is:

$$\frac{\sigma, h, \pi \, [e\rangle \, i \qquad \pi(\sigma(a) + i) > 0}{\sigma, h, \pi \, [a[e]\rangle \, h(\sigma(a) + i)}$$

The semantics of a formula, given in Fig. 2, depends on a store, a heap, and a heap mask and yields a heap mask: $\sigma, h, \pi \, [F\rangle \, \pi'$. The given mask π represents the permissions by which the formula F is framed. The yielded mask π' represents the additional permissions provided by the formula. Hence, a boolean expression is valid if it is true and yields no additional permissions, while evaluating a Perm predicate yields additional permissions to the location, provided the expressions are properly framed. We overload standard addition $+$, summation Σ, and comparison operators to be respectively used as pointwise addition, summation and comparison over the heap masks.

Finally, a formula F is valid for a given store σ, heap h and mask π if starting with the empty heap mask π_0 the required heap mask of F is less than π:

$$\sigma, h, \pi \models F, \text{ if } \sigma, h, \pi_0 \, [F\rangle \, \pi' \wedge \pi' \leq \pi$$

2.3 Iteration Contract

An *iteration contract* specifies the variables read and written by one iteration of the loop. In [8], we prove that if the iteration contract can be proven correct without any further specifications, the iterations are independent and the loop is parallelizable. If a loop has dependences, we can add additional specifications that capture these dependences, and describe how resources are transferred to another iteration of the loop. For example the iteration contract of $\mathsf{L1}$ consists of: a precondition $\mathsf{Perm}(\mathsf{c[i]}, 1) \star \mathsf{Perm}(\mathsf{a[i]}, 1/4)$ and a post-condition $\mathsf{Perm}(\mathsf{c[i]}, 1) \star \mathsf{Perm}(\mathsf{a[i]}, 1/4) \star (\mathsf{c[i]} == \mathsf{a[i]})$.

3 Syntax and Semantics of Deterministic Parallelism

This section presents the abstract syntax and semantics of PPL, our core language for deterministic parallelism.

3.1 Syntax

Fig. 3 presents the PPL syntax. The basic building block of a PPL program is a *block*. Each block has a single entry point and a single exit point. Blocks are composed using three binary composition operators: *parallel composition* $||$, *fusion composition* $\oplus$ and *sequential composition* $;$. The entry block of the program is the outermost block. *Basic blocks* are: a *parallel* block $\mathsf{Par}\ (\mathsf{N})\ \mathsf{S}$; a *vectorized* block $\mathsf{Vec}\ (\mathsf{N})\ \mathsf{V}$; and a *sequential* block S, where N is a positive integer variable that denotes the number of parallel threads, i.e., the block's *parallelization level*, S is a sequence of statements and V is a sequence of guarded assignments $b \Rightarrow \mathsf{ass}$. We assume a restricted syntax for fusion composition such that its operands are parallel basic blocks with the same parallelization levels. Each basic block has a local read-only variable $\mathsf{tid} \in [\mathsf{0..N})$ called *thread identifier* where N is the block's parallelization level. We generalize the term *iteration* to refer to the computations of a single thread in a basic block. So a parallel or vectorized block with parallelization level N has N iterations. For simplicity, but without loss of generality, threads have access to a single shared array which we refer to as *heap*. We assume all memory locations in the heap are allocated initially. A thread may update its local variables by performing a local computation ($v := e$), or by reading from the heap ($v := \mathsf{mem}(e)$). A thread may update the heap by writing one of its local variables to it ($\mathsf{mem}(e) := v$).

3.2 Semantics

The behaviour of PPL programs is described using a small step operational semantics. Throughout, we assume existence of the finite domains: $\mathsf{VarName}$, the set of variable names, Val, the set of all values, which includes the memory locations, Loc, the set of memory locations and $[\mathsf{0..N})$ for thread identifiers. We write $+\!\!+$ to concatenate two statement sequences ($\mathsf{S} +\!\!+ \mathsf{S}$). To define the program state, we use the following definitions.

$$\frac{}{\sigma, h, \pi\,[b\rangle\,\mathsf{true}}{\sigma, h, \pi\,[b\rangle\,\pi_0} \qquad \frac{\sigma, h, \pi\,[e_1\rangle\,l \qquad \sigma, h, \pi\,[e_2\rangle\,f \qquad \pi(l) + f \leq 1}{\sigma, h, \pi\,[\mathsf{Perm}(e_1, e_2)\rangle\,\pi_0[l \mapsto f]}$$

$$\frac{\sigma, h, \pi\,[b\rangle\,\mathsf{true} \qquad \sigma, h, \pi\,[F_1\rangle\,\pi'}{\sigma, h, \pi\,[b?F_1 : F_2\rangle\,\pi'} \qquad \frac{\sigma, h, \pi\,[b\rangle\,\mathsf{false} \qquad \sigma, h, \pi\,[F_2\rangle\,\pi'}{\sigma, h, \pi\,[b?F_1 : F_2\rangle\,\pi'}$$

$$\frac{\sigma, h, \pi\,[F_1\rangle\,\pi' \qquad \sigma, h, \pi + \pi'\,[F_2\rangle\,\pi''}{\sigma, h, \pi\,[F_1 \star F_2\rangle\,\pi' + \pi''} \qquad \frac{\forall i \in I : \sigma, h, \pi\,[F(i)\rangle\,\pi_i \quad \pi + \Sigma_{i\in I}\pi_i \leq 1}{\sigma, h, \pi\,[\bigstar_{i\in I} F(i)\rangle\,\Sigma_{i\in I}\pi_i}$$

Fig. 2. Semantics of Formulas in Permission-based Separation Logic

Parallel Programming Language:

$\mathsf{Block} ::= \big(\mathsf{Block} \mid\mid \mathsf{Block}\big) \mid \big(\mathsf{Block} \oplus \mathsf{Block}\big) \mid \big(\mathsf{Block} \fatsemi \mathsf{Block}\big) \mid \mathsf{Par(N)\ S} \mid \mathsf{S}$
$\mathsf{S} \quad ::= s;\mathsf{S} \mid \mathsf{skip}$
$s \quad ::= \mathsf{ass} \mid \mathsf{if}\ (b)\ \{\mathsf{S}\}\ \mathsf{else}\ \{\mathsf{S}\} \mid \mathsf{while}\ (b)\ \{\mathsf{S}\} \mid \mathsf{Vec(N)\ V}$
$\mathsf{V} \quad ::= b \Rightarrow \mathsf{ass};\mathsf{V} \mid \mathsf{skip}$
$\mathsf{ass} \quad ::= v := e \mid v := \mathsf{mem}(e) \mid \mathsf{mem}(e) := v$
$b \quad ::=$ boolean expression over private memory
$e \quad ::=$ arithmetic expression over private memory
$v \quad ::=$ thread local variable

Fig. 3. Abstract Syntax for Parallel Programming Language

$h \in \mathsf{Heap} \triangleq \mathsf{Loc} \to \mathsf{Val}$	heap, modeled as a single shared array
$\gamma \in \mathsf{Store} \triangleq \mathsf{VarName} \to \mathsf{Val}$	program store, accessible to all threads
$\sigma \in \mathsf{PrivateMem} \triangleq \mathsf{VarName} \to \mathsf{Val}$	private memory, accessible to a single thread

Now we define BlockState. We distinguish various kinds of block states: an initial state Init, composite block states ParC and SeqC, a state in which a parallel basic block should be executed Par, a local state Local in which a vectorized or a sequential basic block should be executed, and a terminated block state Done.

$\mathsf{EB} \in \mathsf{BlockState} \triangleq$

$\mathsf{Init(Block)}\mid$	initial block states
$\mathsf{ParC(EB, EB)}\mid \mathsf{SeqC(EB, Block)}\mid$	composite block states
$\mathsf{Par}(\mathbb{LS})\mid$	parallel basic block states
$\mathsf{Local(LS)}\mid$	thread local states
Done	terminated block state

The Init state consists of a block statement Block. The ParC state consists of two block states, and the SeqC state contains a block state and a block statement Block; they capture all the states that a parallel composition and a sequential composition of two blocks might be in, respectively. The basic block state Par captures all the states that a parallel basic block Par (N) S might be in during its execution. It contains a mapping $\mathbb{LS} \in [0..\mathsf{N}) \to \mathsf{LocalState}$, that maps each thread to its local state, which models the parallel execution of the threads. There are three kinds of local states: a vectorized state Vec, a sequential state Seq, and a terminated sequential state Done.

$\mathsf{LS} \in \mathsf{LocalState} \triangleq$

$\mathsf{Vec}(\Sigma, \mathsf{E}, \mathsf{V}, \sigma, \mathsf{S})\mid$	vectorized basic block states
$\mathsf{Seq}(\sigma, \mathsf{S})\mid$	sequential basic block states
Done	terminated sequential basic block states

The Vec block state captures all states that a vectorized basic block Vec (N) V might be in during its execution. It consists of $\Sigma \in [0..\mathsf{N}) \to \mathsf{PrivateMem}$, which maps each thread to its private memory, the body to be executed V, a private memory σ, and a statement S. As vectorized blocks may appear inside a sequential block, keeping σ and S allows continuation of the sequential basic block after termination of the vectorized block. To model vectorized execution, the state contains an auxiliary set $\mathsf{E} \subseteq [0..\mathsf{N})$ that models which threads have

$$\frac{}{\mathsf{Init}(\mathsf{Block}_1||\mathsf{Block}_2),\gamma,h \to_p \mathsf{ParC}(\mathsf{Init}(\mathsf{Block}_1),\mathsf{Init}(\mathsf{Block}_2)),\gamma,h}\ [\textbf{Init ParC}]$$

$$\frac{}{\mathsf{Init}(\mathsf{Block}_1;\mathsf{Block}_2),\gamma,h \to_p \mathsf{SeqC}(\mathsf{Init}(\mathsf{Block}_1),\mathsf{Block}_2),\gamma,h}\ [\textbf{Init SeqC}]$$

$$\frac{}{\mathsf{Init}(\mathsf{Par(N)}\ \mathsf{S}_1 \oplus \mathsf{Par(N)}\ \mathsf{S}_2),\gamma,h \to_p \mathsf{Init}(\mathsf{Par(N)}\ \mathsf{S}_1 +\!\!+ \mathsf{S}_2),\gamma,h}\ [\textbf{Init Fuse}] \quad \frac{\mathsf{LS},h \to_s \mathsf{LS}',h'}{\mathsf{Local}(\mathsf{LS}),\gamma,h \to_p \mathsf{Local}(\mathsf{LS}'),\gamma,h'}\ [\textbf{Lift Seq}]$$

$$\frac{\mathbb{LS} \triangleq \lambda t \in [0..\mathsf{N}).\mathsf{Seq}(\gamma[\mathsf{tid} := t],\mathsf{S})}{\mathsf{Init}(\mathsf{Par(N)}\ \mathsf{S}),\gamma,h \to_p \mathsf{Par}(\mathbb{LS}),\gamma,h}\ [\textbf{Init Par}] \quad \frac{}{\mathsf{Init}(\mathsf{S}),\gamma,h \to_p \mathsf{Local}(\mathsf{Seq}(\gamma[\mathsf{tid} := 0],\mathsf{S})),\gamma,h}\ [\textbf{Init Seq}]$$

$$\frac{\mathsf{EB}_1,\gamma,h \to_p \mathsf{EB}_1',\gamma,h'}{\mathsf{ParC}(\mathsf{EB}_1,\mathsf{EB}_2),\gamma,h \to_p \mathsf{ParC}(\mathsf{EB}_1',\mathsf{EB}_2),\gamma,h'}\ [\textbf{ParC Step1}] \quad \frac{\mathsf{EB}_2,\gamma,h \to_p \mathsf{EB}_2',\gamma,h'}{\mathsf{ParC}(\mathsf{EB}_1,\mathsf{EB}_2),\gamma,h \to_p \mathsf{ParC}(\mathsf{EB}_1,\mathsf{EB}_2'),\gamma,h'}\ [\textbf{ParC Step2}]$$

$$\frac{}{\mathsf{ParC}(\mathsf{Done},\mathsf{Done}),\gamma,h \to_p \mathsf{Done},\gamma,h}\ [\textbf{ParC Done}] \quad \frac{}{\mathsf{Local}(\mathsf{Done}),\gamma,h \to_p \mathsf{Done},\gamma,h}\ [\textbf{Local Done}]$$

$$\frac{\mathsf{EB},\gamma,h \to_p \mathsf{EB}',\gamma,h'}{\mathsf{SeqC}(\mathsf{EB},\mathsf{Block}),\gamma,h \to_p \mathsf{SeqC}(\mathsf{EB}',\mathsf{Block}),\gamma,h'}\ [\textbf{SeqC Step}] \quad \frac{}{\mathsf{SeqC}(\mathsf{Done},\mathsf{Block}),\gamma,h \to_p \mathsf{Init}(\mathsf{Block}),\gamma,h}\ [\textbf{SeqC Done}]$$

$$\frac{i \in \mathsf{dom}(\mathbb{LS}) \quad \mathbb{LS}(i),h \to_s \mathsf{LS}',h'}{\mathsf{Par}(\mathbb{LS}),\gamma,h \to_p \mathsf{Par}(\mathbb{LS}[i := \mathsf{LS}']),\gamma,h'}\ [\textbf{Par Step}] \quad \frac{\forall i \in \mathsf{dom}(\mathbb{LS}).(\mathbb{LS}(i) = \mathsf{Done})}{\mathsf{Par}(\mathbb{LS}),\gamma,h \to_p \mathsf{Done},\gamma,h}\ [\textbf{Par Done}]$$

Fig. 4. Operational semantics for program execution

already executed the current instruction. Only when E equals $[0..\mathsf{N})$, the next instruction is ready to be executed. Finally, the Seq block state consists of private memory σ and a statement S.

We model the *program state* as a triple of block state, program store and heap (EB,γ,h) and *thread state* as a pair of local state and heap (LS,h). The program store is constant within a block and it contains all global variables (e.g. the initial address of arrays). To simplify our notation, each thread receives a copy of the program store as part of its private memory when it initializes. The operational semantics is defined as a transition relation between program states: $\to_p \subseteq (\mathsf{BlockState} \times \mathsf{Store} \times \mathsf{Heap}) \times (\mathsf{BlockState} \times \mathsf{Store} \times \mathsf{Heap})$, (Fig. 4), using an auxiliary transition relation between thread local states $\to_s \subseteq (\mathsf{LocalState} \times \mathsf{Heap}) \times (\mathsf{LocalState} \times \mathsf{Heap})$, (Fig. 5), and a standard transition relation $\to_{ass} \subseteq (\mathsf{PrivateMem} \times \mathsf{S} \times \mathsf{Heap}) \times (\mathsf{PrivateMem} \times \mathsf{Heap})$ to evaluate assignments, (Fig. 6). The semantics of expression e and boolean expression b over private memory σ, written $\mathcal{E}[\![e]\!]_\sigma$ and $\mathcal{B}[\![b]\!]_\sigma$ respectively, is standard and not discussed any further. We use the standard notation for function update: given a function $f : A \to B$, $a \in A$, and $b \in B$:

$$f[a := b] = x \mapsto \begin{cases} b & , \ x = a \\ f(x), & \text{otherwise} \end{cases}$$

Program execution starts in a program state $(\mathsf{Init}(\mathsf{Block}),\gamma,h)$ where Block is the program's entry block. Depending on the form of Block, a transition is made

$$\frac{}{\mathsf{Seq}(\sigma, \mathsf{while}\,(b)\,\{\mathsf{S}\};\mathsf{S}'), h \rightarrow_s \mathsf{Seq}(\sigma, \mathsf{if}\,(b)\,\{\mathsf{S} +\!\!+ \mathsf{while}\,(b)\,\{\mathsf{S}\}\}\,\mathsf{else}\,\{\mathsf{skip}\};\mathsf{S}'), h}\,[\textbf{While}]$$

$$\frac{\mathcal{B}[\![b]\!]_\sigma}{\mathsf{Seq}(\sigma, \mathsf{if}(b)\{\mathsf{S}_1\}\mathsf{else}\{\mathsf{S}_2\};\mathsf{S}), h \rightarrow_s \mathsf{Seq}(\sigma, \mathsf{S}_1 +\!\!+ \mathsf{S}), h}\,[\textbf{if}^{\textbf{true}}]$$

$$\frac{\neg\mathcal{B}[\![b]\!]_\sigma}{\mathsf{Seq}(\sigma, \mathsf{if}(b)\{\mathsf{S}_1\}\mathsf{else}\{\mathsf{S}_2\};\mathsf{S}), h \rightarrow_s \mathsf{Seq}(\sigma, \mathsf{S}_2 +\!\!+ \mathsf{S}), h}\,[\textbf{if}^{\textbf{false}}]$$

$$\frac{\sigma, \mathsf{ass}, h \rightarrow_{ass} \sigma', h'}{\mathsf{Seq}(\sigma, \mathsf{ass};\mathsf{S}), h \rightarrow_s \mathsf{Seq}(\sigma', \mathsf{S}), h'}\,[\textbf{Ass}] \quad \frac{}{\mathsf{Seq}(\sigma, \mathsf{skip}), h \rightarrow_s \mathsf{Done}, h}\,[\textbf{Seq Done}]$$

$$\frac{\Sigma \triangleq \lambda t \in [0..\mathsf{N}).\sigma[\mathsf{tid} := t]}{\mathsf{Seq}(\sigma, \mathsf{Vec}(\mathsf{N})\ \mathsf{V};\mathsf{S}), h \rightarrow_s \mathsf{Vec}(\Sigma, \emptyset, \mathsf{V}, \sigma, \mathsf{S}), h}\,[\textbf{Init Vec}]$$

$$\frac{i \in \mathsf{dom}(\Sigma)\backslash\mathsf{E} \quad \mathcal{B}[\![b]\!]_{\Sigma(i)} \quad \Sigma(i), \mathsf{ass}, h \rightarrow_{ass} \sigma', h'}{\mathsf{Vec}(\Sigma, \mathsf{E}, b \Rightarrow \mathsf{ass};\mathsf{V}, \sigma, \mathsf{S}), h \rightarrow_s \mathsf{Vec}(\Sigma[i := \sigma'], \mathsf{E} \cup \{i\}, b \Rightarrow \mathsf{ass};\mathsf{V}, \sigma, \mathsf{S}), h'}\,[\textbf{Vec Step1}]$$

$$\frac{i \in \mathsf{dom}(\Sigma)\backslash\mathsf{E} \quad \neg\mathcal{B}[\![b]\!]_{\Sigma(i)}}{\mathsf{Vec}(\Sigma, \mathsf{E}, b \Rightarrow \mathsf{ass};\mathsf{V}, \sigma, \mathsf{S}), h \rightarrow_s \mathsf{Vec}(\Sigma, \mathsf{E} \cup \{i\}, b \Rightarrow \mathsf{ass};\mathsf{V}, \sigma, \mathsf{S}), h}\,[\textbf{Vec Step2}]$$

$$\frac{}{\mathsf{Vec}(\Sigma, \mathsf{dom}(\Sigma), b \Rightarrow \mathsf{ass};\mathsf{V}, \sigma, \mathsf{S}), h \rightarrow_s \mathsf{Vec}(\Sigma, \emptyset, \mathsf{V}, \sigma, \mathsf{S}), h}\,[\textbf{Vec Sync}] \quad \frac{}{\mathsf{Vec}(\Sigma, \mathsf{E}, \mathsf{skip}, \sigma, \mathsf{S}), h \rightarrow_s \mathsf{Seq}(\sigma, \mathsf{S}), h}\,[\textbf{Vec Done}]$$

Fig. 5. Operational semantics for thread execution

into an appropriate block state, leaving the heap unchanged. The evaluation of a ParC state non-deterministically evaluates one of its block states (i.e. EB_1 or EB_2), evaluation of a sequential block is done by evaluating the local state. The evaluation of a SeqC state evaluates its block state EB step by step when this evaluation is done, the subsequent block is initiated.

The evaluation of a parallel basic block is defined by the rules **Par Step** and **Par Done**. To allow all possible interleavings of the threads in the block's thread pool, each thread has its own local state LS, which can be executed independently, modeled by the mapping $\mathbb{LS}$. A thread in the parallel block terminates if there is no more statement to be executed and a parallel block terminates if all threads executing the block are already terminated.

The evaluation of sequential basic block's statements as defined in Fig. 5 is standard except when it contains a vectorized basic block. A sequential basic block terminates if there is no instruction left to be executed (**Seq Done**). The execution of a vectorized block (defined by the rules **Init Vec**, **Vec Step**, **Vec Sync** and **Vec Done** in Fig. 5) is done in lock-step, i.e. all threads execute the same instruction no thread can proceed to the next instruction until all are done, meaning that they all share the same program counter. As explained, we capture this by maintaining an auxiliary set, E, which contains the identifier of the threads that have already executed the vector instruction (i.e. the guarded assignment $b \Rightarrow \mathsf{ass}$). When a thread executes a vector instruction, its thread

$$\frac{}{\sigma, v := e, h \rightarrow_{ass} \sigma[v := \mathcal{E}[\![e]\!]_\sigma], h}\ [\textbf{LAss}]$$

$$\frac{}{\sigma, v := \mathsf{mem}(e), h \rightarrow_{ass} \sigma[v := h(\mathcal{E}[\![e]\!]_\sigma)], h}\ [\textbf{rdsh}] \qquad \frac{}{\sigma, \mathsf{mem}(e) := v, h \rightarrow_{ass} \sigma, h[\mathcal{E}[\![e]\!]_\sigma := v]}\ [\textbf{wrsh}]$$

Fig. 6. Operational semantics for assignments

identifier is added to E (rules **Vec Step**). The semantics of vector instructions (i.e. guarded assignments) is the semantics of assignments if the guard evaluates to true and it does nothing otherwise. When all threads have executed the current vector instruction, the condition $\mathsf{E} = \mathsf{dom}(\Sigma)$ holds, and execution moves on to the next vector instruction of the block (with an empty auxiliary set) (rule **Vec Sync**).The semantics of assignments as defined in Fig. 6 is standard and does not require further discussion.

4 Verification Approach

This section discusses our verification technique for reasoning about PPL programs, as well as soundness of our verification approach.

4.1 Verification

For the verification of PPL programs, we assume that each basic block is specified by an iteration contract. We distinguish two kinds of formulas in an iteration contract: resource formulas (in permission-based separation logic) and functional formulas (in first-order logic). For an individual basic block if its iteration contract is proven correct, then the basic block is data race free and it is functionally correct w.r.t. its iteration contract. To verify the correctness of the program, using standard permission-based separation logic rules, the contracts of all composite blocks should be given. However, our verification approach requires only the basic blocks to be specified at the cost of an extra proof obligation that ensures that the heap accesses of all iterations which are not ordered sequentially are non-conflicting (i.e. they are disjoint or they are read accesses). If this condition holds, correctness of the PPL program can be derived from the correctness of a linearised variant of the program. The rest of this section discusses the formalization of our approach.

To verify a program, we require each basic block of the program to be specified by an iteration contract which consists of: a resource contract $rc(i)$, and a functional contract $fc(i)$, where i is the block's iteration variable. The functional contract consists of a precondition $P(i)$, and a postcondition $Q(i)$. We also require the program to be globally specified by a contract G which consists of the program's resource contract $RC_{\mathcal{P}}$ and the program's functional contract $FC_{\mathcal{P}}$ with the program's precondition $P_{\mathcal{P}}$ and the program's postcondition $Q_{\mathcal{P}}$.

Let $\mathbb{P}$ be the set of all PPL programs and $\mathcal{P} \in \mathbb{P}$ be an arbitrary PPL program assuming that each basic block in $\mathcal{P}$ is identified by a unique label. We define

$\mathbb{B}_{\mathcal{P}} = \{b_1, b_2, ..., b_n\}$, as the finite set of basic block labels of the program $\mathcal{P}$. For a basic block b with parallelization level m, we define a finite set of iteration labels $I_b = \{0^b, 1^b, ..., (m-1)^b\}$ where i^b indicates the i^{th} iteration of the block b. Let $\mathbb{I}_{\mathcal{P}} = \bigcup_{b \in \mathbb{B}_{\mathcal{P}}} I_b$ be the finite set of all iterations of the program $\mathcal{P}$.

To state our proof rule, we first define the set of all iterations which are not ordered sequentially, the *incomparable iteration pairs*, $\mathfrak{I}_{\perp}^{\mathcal{P}}$ as:

$$\mathfrak{I}_{\perp}^{\mathcal{P}} = \{(i^{b_1}, j^{b_2}) | i^{b_1}, j^{b_2} \in \mathbb{I}_{\mathcal{P}} \wedge b_1 \neq b_2 \wedge i^{b_1} \not\prec_e j^{b_2} \wedge j^{b_2} \not\prec_e i^{b_1}\}$$

where $\prec_e \subseteq \mathbb{I}_{\mathcal{P}} \times \mathbb{I}_{\mathcal{P}}$ is the least partial order which defines an extended happens-before relation. The extension addresses the iterations which are happens-before each other because their blocks are fused. We define $\prec_e$ based on two partial orders over the program's basic blocks: $\prec \subseteq \mathbb{B}_{\mathcal{P}} \times \mathbb{B}_{\mathcal{P}}$ and $\prec_{\oplus} \subseteq \mathbb{B}_{\mathcal{P}} \times \mathbb{B}_{\mathcal{P}}$. The former is the standard happens-before relation of blocks where they are sequentially composed by $\fatsemi$ and the latter is an happens-before relation w.r.t. fusion composition $\oplus$. They are defined by means of an auxiliary partial order generator function $\mathcal{G}(\mathcal{P}, \delta) : \mathbb{P} \times \{\fatsemi, \oplus\} \rightarrow \mathbb{B}_{\mathcal{P}} \times \mathbb{B}_{\mathcal{P}}$ such that: $\prec = \mathcal{G}(\mathcal{P}, \fatsemi)$ and $\prec_{\oplus} = \mathcal{G}(\mathcal{P}, \oplus)$. We define $\mathcal{G}$ as follows:

$$\mathcal{G}(\mathcal{P}, \delta) = \begin{cases} \mathbb{G} \cup \{(b', b'') | b' \in \mathbb{B}_{\mathcal{P}'} \wedge b'' \in \mathbb{B}_{\mathcal{P}''}\}, & \text{if } \mathcal{P} = \mathcal{P}' \bullet \mathcal{P}'' \ \wedge \ \delta = \bullet \\ \mathbb{G}, & \text{if } \mathcal{P} = \mathcal{P}' \bullet \mathcal{P}'' \ \wedge \ \delta \neq \bullet \\ \emptyset, & \text{if } \mathcal{P} \in \{\mathsf{Par(N)\ S, S}\} \end{cases}$$

where $\mathbb{G} = \mathcal{G}(\mathcal{P}', \delta) \cup \mathcal{G}(\mathcal{P}'', \delta)$.

The function $\mathcal{G}$ computes the set of all iteration pairs of the input program $\mathcal{P}$ which are in relation w.r.t. the given composition operator δ. This computation is basically a syntactical analysis over the input program. Now we define the extended partial order $\prec_e$ as:

$$\forall i^b, j^{b'} \in \mathbb{I}_{\mathcal{P}}. i^b \prec_e j^{b'} \Leftrightarrow (b \prec b') \vee \big((b \prec_{\oplus} b') \ \wedge \ (i = j)\big)$$

This means that the iteration i^b happens-before the iteration $j^{b'}$ if b happens-before b' (i.e. b is sequentially composed with b') or if b is fused with b' and i and j are corresponding iterations in b and b'.

We extend the program logic that we introduced in [8] with the proof rule **b-linearise**. We first define the *block level linearisation* (*b-linearisation* for short) $blin : \mathbb{P} \rightarrow \mathbb{P}_{\fatsemi}$ as a program transformation which substitutes all non-sequential compositions by a sequential composition. We define $\mathbb{P}_{\fatsemi}$ as a subset of $\mathbb{P}$ in which only sequential composition $\fatsemi$ is allowed as composition operator.

Fig. 7 presents the rule **b-linearise**. In the rule, $rc_b(i)$ and $rc_{b'}(j)$ are the resource contracts of two different basic blocks b and b' where $i^b \in I_b$ and $j^{b'} \in I_{b'}$. Application of the rule results in two new proof obligations. The first ensures

$$\frac{\big(\forall (i^b, j^{b'}) \in \mathfrak{I}_{\perp}^{\mathcal{P}}. (RC_{\mathcal{P}} \rightarrow rc_b(i) \star rc_{b'}(j))\big) \quad \{RC_{\mathcal{P}} \star P_{\mathcal{P}}\} blin(\mathcal{P}) \{RC_{\mathcal{P}} \star Q_{\mathcal{P}}\}}{\{RC_{\mathcal{P}} \star P_{\mathcal{P}}\} \mathcal{P} \{RC_{\mathcal{P}} \star Q_{\mathcal{P}}\}} \ [\textbf{b–linearise}]$$

Fig. 7. Proof rule for b-linearisation reduction of PPL programs.

that all heap accesses of all incomparable iteration pairs (the iterations that may run in parallel) are non-conflicting (i.e. all block compositions in $\mathcal{P}$ are memory safe). This reduces the correctness proof of $\mathcal{P}$ to the correctness proof of its b-linearised variant $blin(\mathcal{P})$ (the second proof obligation). Then the second proof obligation is discharged in two steps: (1) proving the correctness of each basic block against its iteration contract (using the proof rule introduced in [8]) and (2) proving the correctness of $blin(\mathcal{P})$ against the program contract.

4.2 Soundness

Next we show that a PPL program with provably correct iteration contracts and a global contract that is provable in our logic extended with the rule **b-linearise** is indeed data race free and functionally correct w.r.t. its specifications. To show this, we prove soundness of the **b-linearise** rule, as well as data race freedom of all verified programs.

For the soundness proof, we show that for each *program execution* there exists a corresponding *b-linearised execution* with the same functional behaviour (i.e. they end in the same terminal state if they start in the same initial state) if all independent iterations are non-conflicting. From the rule's assumption, we know that if the precondition holds for the initial state of the b-linearised execution (which is also the initial state of the program execution) then its terminal state satisfies the postcondition. As both executions end in the same terminal state, the postcondition thus also holds for the program execution. To prove that there exists a matching b-linearised execution for each program execution, we first show that any valid program execution can be normalized w.r.t. program order and second that any normalized execution can be mapped to a b-linearised execution. To formalize this argument, we first define: an *execution*, an *instrumented execution*, and a *normalized execution*.

We assume all program's blocks including basic and composite blocks have a block label and program's statements are labelled by the label of the block to which they belong. Also there exists a total order over the block labels.

Definition 1. *(Execution). An* execution *of a program $\mathcal{P}$ is a finite sequence of state transitions* $\mathsf{Init}(\mathcal{P}), \gamma, h \rightarrow_p^* \mathsf{Done}, \gamma, h'$.

To distinguish between valid and invalid executions, we instrument our operational semantics with *heap masks*. A heap mask models the access permissions to every heap location. It is defined as a map from locations to fractions $\pi : \mathsf{Loc} \rightarrow \mathsf{Frac}$ where Frac is the set of fractions ($[0, 1]$). Any fraction $(0, 1)$ is read and 1 is write permission. The instrumented semantics ensures that each transition has sufficient access permissions to the heap locations that it accesses. We first add a heap mask π to all block state constructors (Init, ParC, SeqC and so on) and local state constructors (Vec, Seq and Done). Then we extend the operational semantics rules such that in each block initialization state with heap mask π an extra premise should be discharged, which states that there are $n \geq 2$ heap masks $\pi_1, ..., \pi_n$, one for each newly initialized state such that $\Sigma_i^n \pi_i \leq \pi$. The heap masks are carried along by the computation and termination transitions without any extra premises, while in the termination transitions

heap masks of the terminated blocks are forgotten as they are not required after termination. As an example, we provide the instrumented versions of the rules **Init ParC**, **ParC Done**, **rdsh**, and **wrsh**.

$$\frac{\pi_1 + \pi_2 \leq \pi}{\mathsf{Init}(\mathsf{Block}_1||\mathsf{Block}_2, \pi), \gamma, h \rightarrow_{p,i} \mathsf{ParC}(\mathsf{Init}(\mathsf{Block}_1, \pi_1), \mathsf{Init}(\mathsf{Block}_2, \pi_2), \pi), \gamma, h}\ [\textbf{Init ParC}]$$

$$\frac{}{\mathsf{ParC}(\mathsf{Done}(\pi_1), \mathsf{Done}(\pi_2), \pi), \gamma, h \rightarrow_{p,i} \mathsf{Done}(\pi), \gamma, h}\ [\textbf{ParC Done}]$$

$$\frac{l = \mathcal{E}[\![e]\!]_\sigma \quad \pi(l) > 0}{\sigma, v := \mathsf{mem}(e), h, \pi \rightarrow_{ass,i} \sigma[v := h(l)], h, \pi}\ [\textbf{rdsh}] \qquad \frac{l = \mathcal{E}[\![e]\!]_\sigma \quad \pi(l) = 1}{\sigma, \mathsf{mem}(e) := v, h, \pi \rightarrow_{ass,i} \sigma, h[l := v], \pi}\ [\textbf{wrsh}]$$

where $\rightarrow_{p,i}$ and $\rightarrow_{ass,i}$ denote program and assignment transition relations in the instrumented semantics respectively. If a transition cannot satisfy its premises it blocks.

Definition 2. *(Instrumented Execution). An* instrumented execution *of a program $\mathcal{P}$ is a finite sequence of state transitions $\mathsf{Init}(\mathcal{P}, \pi), \gamma, h \rightarrow^*_{p,i} \mathsf{Done}(\pi), \gamma, h'$ where the set of all instrumented executions of $\mathcal{P}$ is written as $\mathbb{IE}_\mathcal{P}$.*

Lemma 1. *Assuming that (1). $\vdash \forall(i^b, j^{b'}) \in \mathfrak{I}^\mathcal{P}_\perp.RC_\mathcal{P} \rightarrow rc_b(i) \star rc_{b'}(j)$ and (2). $\forall b \in \mathbb{B}_\mathcal{P}.\{\bigstar_{i\in[0..N_b)} rc_b(i)\}\mathsf{Block}_b\{\bigstar_{i\in[0..N_b)} rc_b(i)\}$ are valid for a program $\mathcal{P}$ (i.e. every basic block in $\mathcal{P}$ respects its iteration contract), for any execution E of the program $\mathcal{P}$, there exists a corresponding instrumented execution.*

Proof. Given an execution E, we assign heap masks to all program states that the execution E might be in. The program's initial state is assigned by a heap mask $\pi \leq 1$. Assumption (1) implies that all iterations which might run in parallel are non-conflicting which implies that for all **Init ParC** transitions, there exist π_1 and π_2 such that $\pi_1 + \pi_2 \leq \pi'$ where π' is the heap mask of the state in which **Init ParC** evaluates. In all computation transitions the successor state receives a copy of the heap mask of its predecessor. Assumption (2) implies that all iterations of all parallel and vectorized basic blocks are non-conflicting. This implies that for an arbitrary **Init Par** or **Init Vec** transition which initializes a basic block b, there exists $\pi_1, ..., \pi_n$ such that $\Sigma^n_i \pi_i \leq \pi_b$ holds in b's initialization transition and in all computation transitions of an arbitrary iteration i of the block b the premises of **rdsh** and **wrsh** transitions is satisfiable by π_i. □

Lemma 2. *All instrumented executions of a program $\mathcal{P}$ are data race free.*

Proof. The proof proceeds by contradiction. Assume that there exists an instrumented execution that has a data race. Thus, there must be two parallel threads such that one writes to and the other one reads from or writes to a shared heap location e. Because all instrumented executions are non-blocking, the premises of all transitions hold. Therefore, $\pi_1(e) = 1$ holds for the first thread, and $\pi_2(e) > 0$ for the second thread either it writes or reads. Also because the program starts with one single main thread, both threads should have a single common ancestor thread z such that $\pi_x(e) + \pi_y(e) \leq \pi_z(e)$ where x and y are the ancestors of the first and the second thread respectively. A thread only gains permission from

its parent; therefore $\pi_1(e) + \pi_2(e) \leq \pi_z(e)$ holds. Permission fractions are in the range $[0, 1]$ by definition, therefore $\pi_1(e) + \pi_2(e) \leq 1$ holds. This implies that if $\pi_1(e) = 1$, then $\pi_2(e) \leq 0$ which is a contradiction. □

A normalized execution is an instrumented execution that respects the program order, which is defined using an auxiliary labelling function $\mathcal{L} : \mathbb{T} \to \mathbb{B}^{all}_{\mathcal{P}} \times \mathbb{L}$ where $\mathbb{T}$ is the set of all transitions, $\mathbb{L}$ is the set of labels $\{I, C, T\}$, and $\mathbb{B}^{all}_{\mathcal{P}}$ is the set of block labels (including both composite and basic block labels).

$$\mathcal{L}(t) = \begin{cases} (LB(\mathsf{block}), I), & \text{if } t \text{ initializes a block } \mathsf{block} \\ (LB(s), C), & \text{if } t \text{ computes a statement } s \\ (LB(\mathsf{block}), T), & \text{if } t \text{ terminates a block } \mathsf{block} \end{cases}$$

where LB returns the label of each block or statement in the program. We assume the precedence order $I < C < T$ over $\mathbb{L}$. We say transition t with label (b, l) is less than t' with label (b', l') if $(b \leq b') \vee (b > b' \to (l' = T \wedge b \in LB_{sub}(b')))$ where $LB_{sub}(b)$ returns the label set of all blocks of which b is composed.

Definition 3. *(Normalized Execution). An instrumented execution labelled by $\mathcal{L}$ is* normalized *if the labels of its transitions are in non-decreasing order.*

We transform an instrumented execution to a normalized one by safely commuting the transitions whose labels do not respect the program order.

Lemma 3. *For each instrumented execution of a program $\mathcal{P}$, there exists a normalized execution such that they both end in the same terminal state.*

Proof. Given an instrumented execution $IE = (s_1, t_1) : (s_2, t_2) : IE'$, if $\mathcal{L}(t_1) > \mathcal{L}(t_2)$, a state s_x exists such that a new instrumented execution $IE'' = (s_1, t_2) : (s_x, t_1) : IE'$ can be constructed by swapping two adjacent transitions t_1 and t_2. As the swap is on an instrumented execution which from Lemma 2 is data race free, any accesses of t_1 and t_2 to a shared heap location must be reads. Because t_1 and t_2 are adjacent transitions, no other write may happen in between; therefore the swap preserves the functionality of IE, yielding the same terminal state for IE and IE''. Thus, the corresponding normalized execution of IE obtained by applying a finite number of such swaps, yields the same terminal state as IE. □

Lemma 4. *For each normalized execution of a program $\mathcal{P}$, there exists a b-linearised execution $blin(\mathcal{P})$, such that they both end in the same terminal state.*

Proof. An execution of $blin(\mathcal{P})$ is constructed by applying the map $\mathcal{M} : \mathsf{BlockState} \to \mathsf{BlockState}$ to each state of a normalized execution. $\mathcal{M}$ is defined as:

$$\mathcal{M}(s) = \begin{cases} \mathsf{Init}(blin(\mathcal{P})), & \text{if } s = \mathsf{Init}(\mathcal{P}) \\ \mathsf{SeqC}(\mathcal{M}(\mathsf{EB}_1), \mathsf{Block}_2), & \text{if } s = \mathsf{ParC}(\mathsf{EB}_1, \mathsf{Init}(\mathsf{Block}_2)) \\ \mathcal{M}(\mathsf{EB}_2), & \text{if } s = \mathsf{ParC}(\mathsf{Done}, \mathsf{EB}_2) \\ \mathsf{SeqC}(\mathsf{Par}(\mathbb{LS}_1), \mathsf{Block}_2), & \text{if } s = \mathsf{Par}(\mathbb{LS}_1 +\!\!+ \mathbb{LS}^0_2) \\ s, & \text{otherwise} \end{cases}$$

where $\mathbb{LS}_2^0$ is the initial mapping of thread local states of Block_2 and $\mathsf{Par}(\mathbb{LS}_1 +\!\!+ \mathbb{LS}_2^0)$ indicates the state of two fused parallel blocks $\mathsf{Par}(\mathbb{LS}_1)$ and $\mathsf{Par}(\mathbb{LS}_2^0)$ where $+\!\!+$ is overloaded and indicates pairwise concatenation of statements in the local states $\mathbb{LS}_1$ and $\mathbb{LS}_2^0$ (i.e. $\mathsf{S}_1 +\!\!+ \mathsf{S}_2$). □

Definition 4. *(Validity of Hoare Triple). The Hoare triple* $\{RC_\mathcal{P} \star P_\mathcal{P}\}\mathcal{P}\{RC_\mathcal{P} \star Q_\mathcal{P}\}$ *is valid if for any execution E (i.e.* $\mathsf{Init}(\mathcal{P}), \gamma, h \rightarrow_p^* \mathsf{Done}, \gamma, h'$*) if* $\gamma, h, \pi \vDash RC_\mathcal{P} \star P_\mathcal{P}$ *is valid in the initial state of E, then* $\gamma, h', \pi \vDash RC_\mathcal{P} \star Q_\mathcal{P}$ *is valid in its terminal state.*

The validity of $\gamma, h, \pi \vDash RC_\mathcal{P} \star P_\mathcal{P}$ and $\gamma, h', \pi \vDash RC_\mathcal{P} \star Q_\mathcal{P}$ is defined by the semantics of formulas presented in Section 2.2.

Theorem 1. *The rule* ***b-linearise*** *is sound.*

Proof. Assume (1). $\vdash \forall (i^b, j^{b'}) \in \mathfrak{I}_\perp^\mathcal{P}. RC_\mathcal{P} \rightarrow rc_b(i) \star rc_{b'}(j)$ and (2). $\vdash \{RC_\mathcal{P} \star P_\mathcal{P}\}$ $blin(\mathcal{P})\{RC_\mathcal{P} \star Q_\mathcal{P}\}$. From assumption (2) and the soundness of the program logic used to prove it [8], we conclude (3). $\forall b \in \mathbb{B}_\mathcal{P}.\{\bigstar_{i \in [0..N_b)} rc_b(i)\}\mathsf{Block}_b$ $\{\bigstar_{i \in [0..N_b)} rc_b(i)\}$. Given a program $\mathcal{P}$, implication (3), assumption (1) and, Lemma 1 imply that there exists an instrumented execution *IE* for $\mathcal{P}$. Lemma 3 and Lemma 4 imply that there exists an execution E' for the b-linearised variant of $\mathcal{P}$, $blin(\mathcal{P})$, such that both *IE* and E' end in the same terminal state. The initial states of both *IE* and E' satisfy the precondition $\{RC_\mathcal{P} \star P_\mathcal{P}\}$. From assumption (2) and the soundness of the program logic used to prove it [8], $\{RC_\mathcal{P} \star Q_\mathcal{P}\}$ holds in the terminal state of E' which thus also holds in the terminal state of *IE* as they both end in the same terminal state. □

Finally, we show that a verified program is indeed data race free.

Proposition 1. *A verified program is data race free.*

Proof. Given a program $\mathcal{P}$, with the same reasoning steps mentioned in the Theorem 1, we conclude that there exists an instrumented execution *IE* for $\mathcal{P}$. From Lemma 2 all instrumented executions are data race free. Thus, all executions of a verified program are data race free. □

5 Verification of OpenMP Programs

Finally, this section discusses the practical applicability of our approach, by showing how it can be used for verification of OpenMP programs. We demonstrate this in detail on the OpenMP program presented in Section 2.1. More OpenMP examples are available online[1]. Below we precisely identify a commonly used subset of OpenMP programs that can be verified in our approach.

We verify OpenMP programs in the following three steps: (1) specifying the program (i.e. providing an iteration contract for each loop and writing the

[1] See the online version of the VerCors toolset at `http://www.utwente.nl/vercors/`.

Program Contract (PC):

```
/*@ invariant  a != NULL && b != NULL && c != NULL && d != NULL && L>0;
invariant  \length(a)==L && \length(b)==L && \length(c)==L && \length(d)==L;
context  \forall*  int k; 0 <= k && k < L; Perm(a[k],1/2);
context  \forall*  int k; 0 <= k && k < L; Perm(b[k],1/2);
context  \forall*  int k; 0 <= k && k < L; Perm(c[k],1);
context  \forall*  int k; 0 <= k && k < L; Perm(d[k],1);
ensures  \forall   int k; 0 <= k && k < L; c[k]==a[k]+b[k] && d[k]==a[k]*b[k];@*/
```

Iteration Contract 1 (IC_1) of loop L1:

```
/*@ context Perm(c[i],1) ** Perm(a[i],1/4);
ensures c[i]==a[i];  @*/
```

Iteration Contract 2 (IC_2) of loop L2:

```
/*@ context Perm(c[i],1) ** Perm(b[i],1/4);
ensures c[i]==\old(c[i])+b[i];  @*/
```

Iteration Contract 3 (IC_3) of loop L3:

```
/*@ context Perm(d[i],1) ** Perm(a[i],1/4) ** Perm(b[i],1/4);
ensures d[i]==a[i]*b[i];  @*/
```

Fig. 8. Required contracts for verification of the running OpenMP example

program contract for the outermost OpenMP parallel region), (2) encoding of the specified OpenMP program into its PPL counterpart (carrying along the original OpenMP specifications), (3) checking the PPL program against its specifications. Steps two and three have been implemented as part of the VerCors toolset [6,23]. The details of the encoding algorithm is discussed in Section 5.2.

Fig. 8 shows the required contracts for the example discussed in Section 2.1. There are four specifications. The first one is the program contract which is attached to the outermost parallel block. The others are the iteration contracts of the loops L1, L2 and L3. The *requires* and *ensures* keywords indicate pre and post-conditions of each contract and the *context* keyword is a shorthand for both requiring and ensuring the same predicate. We use $**$ and $\backslash\mathsf{forall}*$ to denote separating conjunction $\star$ and universal separating conjunction $\bigstar_{i\in I}$ receptively. Before verification, we encode the example into the following PPL program $\mathcal{P}$:

$$
\begin{array}{l}
\texttt{/*@ Program Contract @*/} \\
\mathcal{P}\left\{\begin{array}{l}
\Big(\underbrace{\texttt{Par(L) /*@IC}_1\texttt{@*/ c[i]=a[i];}}_{\mathsf{B}_1} \oplus \underbrace{\texttt{Par(L) /*@IC}_2\texttt{@*/ c[i]=c[i]+b[i];}}_{\mathsf{B}_2}\Big) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \| \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \overbrace{\qquad\qquad\qquad\qquad}^{\mathsf{B}_3} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{Par(L) /*@IC}_3\texttt{@*/ d[i]=a[i]*b[i];}
\end{array}\right.
\end{array}
$$

Program $\mathcal{P}$ contains three parallel basic blocks B_1, B_2 and B_3 and is verified by discharging two proof obligations: (1) ensures that all heap accesses of all incomparable iteration pairs (i.e. all iteration pairs except the identical iterations of B_1 and B_2) are non-conflicting which implies that the fusion of B_1 and B_2 and parallel composition of $\mathsf{B}_1 \oplus \mathsf{B}_2$ and B_3 are memory safe (2) consists of first proving that each parallel basic block by itself satisfies its iteration contract $\forall \mathsf{b} \in \{1,2,3\}.\{\bigstar_{i\in[0..L)}\mathsf{IC_b}(i)\}\mathsf{B_b}\ \{\bigstar_{i\in[0..L)}\mathsf{IC_b}(i)\}$, and second proving the cor-

OMP ::= #pragma omp parallel [*clause*]* {Job+} |
Job ::= #pragma omp for [*clause*]* {for-loop {SpecS}} |
#pragma omp simd [*clause*]* {for-loop {SpecS}} |
#pragma omp for simd [*clause*]* {for-loop {SpecS}} |
#pragma omp sections [*clause*]* {Section+} |
#pragma omp single {SpecS |OMP}
Section ::= #pragma omp section {SpecS |OMP} |
SpecS ::= a list of sequential statements with a contract
clause ::= allowed OpenMP clause

Fig. 9. OpenMP Core Grammar

rectness of the b-linearised variant of $\mathcal{P}$ against its program contract $\{RC_{\mathcal{P}} \star P_{\mathcal{P}}\}$ $\mathsf{B}_1 \fatsemi \mathsf{B}_2 \fatsemi \mathsf{B}_3\{RC_{\mathcal{P}} \star Q_{\mathcal{P}}\}$.

We have implemented a slightly more general variant of PPL in the tool that supports variable declarations and method calls. To check the first proof obligation in the tool we quantify over pairs of blocks which allows the number of iterations in each block to be a parameter rather than a fixed number.

5.1 Captured subset of OpenMP

We define a core grammar which captures a commonly used subset of OpenMP [3]. This defines also the OpenMP programs that can be encoded into PPL and then verified using our approach. Fig. 9 presents the OMP grammar which supports the OpenMP annotations: omp parallel, omp for, omp simd, omp for simd, omp sections, and omp single. An OMP program is a finite and non-empty list of Jobs enclosed by omp parallel. The body of omp for, omp simd, and omp for simd, is a for-loop. The body of omp single is either an OMP program or it is a sequential code block SpecS. The omp sections block is a finite list of omp section sub-blocks where the body of each omp section is either an OMP program or it is a sequential code block SpecS.

5.2 OpenMP to PPL Encoding

This section discusses the encoding of OpenMP programs into PPL. The encoding algorithm is presented in Fig. 10 and accepts OpenMP programs that conform to the core grammar in Fig. 9. The algorithm consists of a recursive *translate* step and a *compose* step. The translation step recursively encodes all OMP Jobs into their equivalent PPL code blocks without caring about how they will be composed. Later, the compose step conjoins the translated code blocks together to build a PPL program. The translation step is a map, which applies the function m to the list of input tuples and returns a list of output tuples. Depending on the case, m might itself use another map function sec. The input tuples are in the form (A, C) where A is an OpenMP annotation and C is a code block written in C. The tuple represents an annotated code block in OMP programs. The output tuples are in the form $(P, [A])$ where P is a PPL program and $[A]$ is a list of OpenMP annotations.

The compose step takes as its input a list of tuples in the form $(P, [A])$ (the output of the translate step); then it inserts appropriate PPL composition operators between adjacent tuples in the list provided certain conditions hold. To properly bind tuples to the composition operators, the operators are inserted in three individual passes; one pass for each composition operator, based on the binding precedence of the operators from high to low as follows: $\fatsemi < || < \oplus$.

Operator insertion is done by the function bundle (lines 32-36). In each pass bundle consumes the input list recursively; Each recursive call takes the two first tuples of the list and inserts a composition operator if the tuples satisfy the conditions of the composition operator; otherwise, it moves one tuple forward and starts the same process again.

For each composition operator the conditions are different. The conditions for parallel and fusion compositions are checked by the functions fusiable and par_able. The fusion composition is inserted between two consecutive tuples $(P_i, [A_i])$ and $(P_j, [A_j])$ where both $[A_i]$ and $[A_j]$ are omp for annotations, the clauses of both annotations include schedule(static), and the clauses of $[A_i]$ include nowait. The parallel composition is inserted between any two tuples in the program where the clauses of the first tuple include nowait. Otherwise, the sequential composition is inserted. The final outcome is a single merged tuple $(P, [A])$ where P is the result of the encoding and $[A]$ can be eliminated.

To verify generated PPL programs, our approach requires a program contract and an iteration contract for each SpecS block in the OpenMP program from which all the required PPL contracts can be gained. The only exception is omp simd for which the contract of while-loop should also be given by the user.

6 Related Work

Botincan et al. propose a proof-directed parallelization synthesis which takes as input a sequential program with a proof in separation logic and outputs a parallelized counterpart by inserting barrier synchronizations [10,11]. Hurlin uses a proof-rewriting method to parallelize a sequential program's proof [16]. Compared to them, we prove the correctness of parallelization by reducing the parallel proof to a b-linearised proof. Moreover, our approach allows verification of sophisticated block compositions, which enables reasoning about state-of-the-art parallel programming languages (e.g. OpenMP) while their work remains rather theoretical.

Raychev et al. use abstract interpretation to make a non-deterministic program (obtained by naive parallelization of a sequential program) deterministic by inserting barriers [19]. This technique over-approximates the possible program behaviours which ends up in a determinization whose behaviour is implied by a set of rules which decide between feasible schedules rather than the behaviour of the original sequential program. Unlike them, we do not generate any parallel program. Instead we prove that parallelization annotations can safely be applied and the parallelized program is functionally correct and exhibits the same behaviour as its sequential counterpart. Barthe et al. synthesize SIMD code given pre and postconditions for loop kernels in C++ STL or C# BCL [4]. We

```
Def For as for(i..N*M){SpecS(i)}
Def Par as Par(N∗M){SpecS(tid)}
Def WhileVec as
while(i ∈ [0,N))
  Vec(tid ∈ [0,M)) SpecS(i∗M+tid)
Def ParVec as
Par(tid1 ∈ [0,N))
  Vec(tid2 ∈ [0,M)) SpecS(tid1∗M+tid2)

encode p = compose (translate p)
translate  xs = map m xs
m(omp for clause*,For)
  =(Par,omp for clause*)
m(omp simd simdlen(M) clause*,For)
  =(WhileVec,omp simd clause*)
m(omp for simd simdlen(M) clause*,For)
  =(ParVec,omp for simd clause*)
m(omp sections clause*, xs)
  = (fold ||  (map sec xs),clause*)
m(omp single clause*,x)
  = (map sec x,clause*)
sec(omp parallel clause*, Job+)
  = encode Job+
sec(SpecS) = Par(1){SpecS}

par_able (P1,A1) (P2,A2) = nowait(A1)
fusiable (P1,A1) (P2,A2) =
  omp_for(A1) ∧  omp_for(A2) ∧
  sched_static (A1) ∧ sched_static(A2) ∧
  nowait(A1)

bundle _ _ [x] = [x]
bundle op cond x:y: ys
  = x : r , if !(cond x y)
  = (op x (head r)) : (tail  r), else
  where r = bundle op cond (y:ys)
compose ys =
   let  p1 = bundle ⊕ fusiable ys in
   let  p2 = bundle || par_able  p1 in
   fold ⨟ p2
```

Fig. 10. Encoding of a commonly used subset of OpenMP programs into PPL programs

alternatively enable verification of SIMD loops, by encoding them into vectorized basic blocks. Moreover, we address the parallel or sequential composition of those loops with other forms of parallelized blocks.

Dodds et al. introduce a higher-order variant of Concurrent Abstract Predicates (CAP) to support modular verification of synchronization constructs for deterministic parallelism [14]. While their proofs make explicit use of nested region assertions and higher-order protocols, they do not address the semantic difficulties introduced by these features which make their reasoning unsound.

7 Conclusion and Future Work

We have presented the PPL language which captures the main forms of deterministic parallel programming. Then, we proposed a verification technique to reason about data race freedom and functional correctness of PPL programs. We illustrated the practical applicability of our technique by discussing how a commonly used subset of OpenMP can be encoded into PPL and then verified.

As future work, we plan to look into adapting annotation generation techniques to automatically generate iteration contracts, including both resource formulas and functional properties. This will lead to fully automatic verification of deterministic parallel programs. Moreover, our technique can be extended to address a larger subset of OpenMP programs by supporting more complex OpenMP patterns for scheduling iterations and *omp task* constructs. We also plan to identify the subset of atomic operations that can be combined with our technique that allows verification of the widely-used reduction operations.

References

1. LLNL OpenMP Benchmarks. Last accessed Nov. 28, 2016. https://asc.llnl.gov/CORAL-benchmarks/.
2. OpenMP Architecture Review Board, OpenMP API Specification for Parallel Programming. Last accessed Nov. 28, 2016. http://openmp.org/wp/.
3. A. Aviram and B. Ford. Deterministic OpenMP for Race-free Parallelism. In *HotPar'11*, pages 4–4, Berkeley, CA, USA, 2011.
4. G. Barthe, J. M. Crespo, S. Gulwani, C. Kunz, and M. Marron. From Relational Verification to SIMD Loop Synthesis. In *ACM SIGPLAN Notices*, volume 48, pages 123–134, 2013.
5. M. J. Berger, M. J. Aftosmis, D. D. Marshall, and S. M. Murman. Performance of a New CFD Flow Solver Using a Hybrid Programming Paradigm. *J. Parallel Distrib. Comput.*, 65(4):414–423, Apr. 2005.
6. S. Blom and M. Huisman. The VerCors tool for Verification of Concurrent Programs. In *FM*, volume 8442 of *LNCS*, pages 127–131. Springer, 2014.
7. S. Blom, M. Huisman, and M. Mihelčić. Specification and Verification of GPGPU Programs. *Science of Computer Programming*, pages 376–388, 2014.
8. S. C. C. Blom, S. Darabi, and M. Huisman. Verification of Loop Parallelisations. In *FASE 2015*, volume 9033 of *LNCS*, pages 202–217. Springer, 2015.
9. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270, 2005.
10. M. Botincan, M. Dodds, and S. Jagannathan. Resource-sensitive Synchronization Inference by Abduction. In *POPL*, pages 309–322, 2012.
11. M. Botinčan, M. Dodds, and S. Jagannathan. Proof-Directed Parallelization Synthesis by Separation Logic. *ACM Trans. Program. Lang. Syst.*, 35:1–60, 2013.
12. J. Boyland. Checking interference with fractional permissions. In *Static Analysis Symposium*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.
13. S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization. IISWC 2009*, pages 44–54, 2009.
14. M. Dodds, S. Jagannathan, and M. J. Parkinson. Modular Reasoning for Deterministic Parallelism. In *ACM SIGPLAN Notices*, pages 259–270, 2011.
15. C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
16. C. Hurlin. Automatic Parallelization and Optimization of Programs by Proof Rewriting. In *SAS*, pages 52–68. Springer, 2009.
17. H.-Q. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. 1999.
18. P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.
19. V. Raychev, M. Vechev, and E. Yahav. Automatic Synthesis of Deterministic Concurrency. In *SAS*, pages 283–303. Springer, 2013.
20. J. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.
21. J. Smans, B. Jacobs, and F. Piessens. Implicit Dynamic Frames. *ACM Trans. Program. Lang. Syst.*, 34(1):2:1–2:58, 2012.
22. H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *POPL*, pages 97–108. ACM, 2007.
23. VerCors project homepage, Sep. 28, 2016. http://www.utwente.nl/vercors/.