International Journal on Software Tools for Technology Transfer (2021) 23:741–763
https://doi.org/10.1007/s10009-020-00601-z

GENERAL

Special Issue: MeTRID

Correct program parallelisations

S. Blom[1] · S. Darabi[2] · M. Huisman[3] · M. Safari[3]

Accepted: 10 December 2020 / Published online: 14 February 2021

Abstract
A commonly used approach to develop deterministic 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 reason about such compiler directives, in particular to show that they do not change the behaviour of the program. Moreover, the verification technique is tool-supported and can be combined with proving functional correctness of the program. To develop our verification technique, we propose a simple intermediate representation (syntax and semantics) that captures the main forms of deterministic parallel programs. This language distinguishes three kinds of basic blocks: parallel, vectorised and sequential blocks, which can be composed using three different composition operators: sequential, parallel and fusion composition. We show how a widely used subset of OpenMP can be encoded into this intermediate representation. Our verification technique builds on the notion of iteration contract to specify the behaviour of basic blocks; we show that if iteration contracts are manually specified for single blocks, then that is sufficient to automatically reason about data race freedom of the composed program. Moreover, we also show that it is sufficient to establish functional correctness on a linearised version of the original program to conclude functional correctness of the parallel program. Finally, we exemplify our approach on an example OpenMP program, and we discuss how tool support is provided.

Keywords Software verification · Deterministic parallel programming · Parallelisation

1 Introduction

A common approach to handle the complexity of parallel programming is to write a sequential program augmented with parallelisation compiler directives that indicate which part of the code might be parallelised. A parallelising compiler consumes the annotated sequential program and automatically generates a parallel version. This parallel programming approach is often called *deterministic parallel programming*, as the parallelisation 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, for example, OpenMP [20], and is often used for financial and scientific applications (see e.g. [4,11,17,21]).

Although it is relatively easy to write parallel programs in this way, careless use of compiler directives can easily introduce data races[1] and consequently non-deterministic program behaviour. This paper proposes a tool-supported static verification technique to prove that parallelisation as indicated by the compiler directives does not introduce such non-determinism. Our technique is not fully automatic: the user has to add some additional annotations, and verification of these annotations gives the guarantee that program behaviour is not changed by the compiler directives. Moreover, we also show that it is sufficient to prove functional correctness on a sequential version of the program, in order

✉ M. Safari
m.safari@utwente.nl

S. Blom
sblom@betterbe.com

S. Darabi
saeed.darabi@gmail.com

M. Huisman
m.huisman@utwente.nl

1 BetterBe, Enschede, The Netherlands

2 ASML Veldhoven, Veldhoven, The Netherlands

3 University of Twente, Enschede, The Netherlands

[1] A data race is a situation when two or more threads may access the same memory location simultaneously where at least one of them is a write.

to conclude functional correctness of the parallel program. We develop a verification technique to reason about data race freedom and functional correctness on an intermediate representation language, called PPL (for Parallel Programming Language), which captures the core features of deterministic parallel programming. We then show that a commonly used subset of a deterministic programming language such as OpenMP can be encoded into this intermediate representation, and thus, our verification technique allows us to reason about the correctness of compiler directives in OpenMP. The verification technique is implemented as part of our program verifier VerCors. That means, if we (manually) annotate an OpenMP program with specifications, data race freedom and functional correctness can be verified automatically. We illustrate this approach on some characteristic examples.

In essence, our intermediate representation language PPL is defined in terms of the composition of code blocks. We identify three kinds of *basic blocks*: a *parallel block*, a *vectorised 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 that users specify each basic block by an iteration contract that describes which memory locations are read and written by a thread. We introduce these contracts and present verification rules for basic blocks. Moreover, the program itself can be specified by a global contract. To verify the global contract, we show that the block compositions are memory safe (i.e. data race free) by proving that for all 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 parallelised program is functionally correct.

The main contributions of this paper are the following:

- An intermediate representation language PPL that captures the core features of deterministic parallel programming, with a suitable operational semantics.
- An algorithm that encodes a commonly used subset of OpenMP into its PPL intermediate representation.
- A tool-supported verification approach for reasoning about data race freedom and functional correctness of OpenMP programs by using the encoding of OpenMP into PPL.

This paper is an extended version of our paper presented at NFM 2017 [12]. In addition, it contains (1) a rephrasing of the verification rules for parallel and vectorised loops, presented at FASE 2015 [5] in the setting of PPL, i.e. rephrasing them for basic blocks, and (2) an algorithm that encodes a commonly used subset of OpenMP into PPL.

This paper is organised as follows. After some background information on OpenMP and our program specification language, Sect. 3 introduces our intermediate representation language PPL, presenting syntax and semantics. Then, Sect. 4 shows how OpenMP programs are encoded into PPL. Section 5 presents the verification rules for basic blocks, while Sect. 6 presents the verification rules for block compositions. Section 7 provides more information on how the tool support is provided, while Sect. 8 uses our technique on an OpenMP program. Finally, Sect. 9 presents related work, and Sect. 10 concludes the paper and discusses future work.

2 Background

This section provides some background information on the OpenMP compiler directives and briefly introduces syntax and semantics of our program specification language.

2.1 OpenMP

As mentioned above, in this paper we consider a frequently used subset of OpenMP constructs, using only the following pragmas: omp parallel, omp for, omp simd, omp for simd, omp sections, and omp single, as well as all allowed clauses. We illustrate these OpenMP features by means of examples. For full details on OpenMP, we refer to [20]. Later, Sect. 4 shows how programs in this subset are encoded into our core parallel programming language, and Sect. 8 shows how to verify that these programs can safely be parallelised, after the user has added the necessary program contracts.

Example 1 Figure 1 presents a sequential C program augmented by OpenMP compiler directives (called *pragmas*). The pivotal parallelisation annotation in OpenMP is *omp parallel* which denotes a parallelisable 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.

This example shows a parallel region with three for-loops L1, L2, and L3. The loops are marked as *omp for* meaning that they are parallelisable (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 the *fusion* of 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 the implicit barrier at

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

Fig. 1 OpenMP example

the end of *omp for* is eliminated. 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 the *private* clause) when they are passed to a parallel region.

Since OpenMP 4.0, support for the single instruction multiple data (SIMD) execution model has been added to the OpenMP standard. The SIMD execution model is a well-known technique to speed up vector arithmetics, specifically in scientific applications.

Example 2 Figure 2 presents an OpenMP example to illustrate this. The first loop uses the *omp simd* annotation to vectorise the for-loop L1, which partitions the iterations of the loop into smaller chunks, where the size of each chunk is equal to the vectorisation size given by the extra clause *simdlen* (i.e. M in this example). The loop execution is defined as the sequential execution of chunks, where each chunk is executed in a vectorised fashion.

The second for-loop (L2) shows the other form of OpenMP vectorisation using the *omp for simd* annotation. In this case, the loop execution is defined similarly, however the iteration chunks are executed in parallel rather than sequentially. Figure 3 visualises the execution of these loops.

Example 3 Figure 4 presents how the parallel execution of two parallel regions is defined in OpenMP. The example consists of three parallel regions: P_1 in lines 4–11, P_2 in lines 14–23 and P_3 in lines 26–29. Similar to the previous examples, the behaviour of each thread is defined by further OpenMP compiler directives. We use the *omp sections*

```
1  /*@ Program Contract (PC) @*/
2  void adm(int L,int M,int a[],int b[], int c[]){
3     #pragma omp parallel {
4        #pragma omp simd simdlen(M)
5        for(int i=0;i<L;i++) //Loop L1
6         /*@ Iteration  Contract 1 (IC1) @*/
7         { c[i]=a[i]*b[i]; }
8        #pragma omp for simd simdlen(M)
9        for(int i=0;i<L;i++) //Loop L2
10        /*@ Iteration  Contract 2 (IC2) @*/
11        { c[i]=c[i]*b[i]; }
12    }
13 }
```

Fig. 2 Vectorised loops in OpenMP

annotation, which defines the blocks of the code (marked by *omp section*) which are executed in parallel. For example, two threads are forked upon entering the parallel region P_1, one executes the method add and the other one executes the method mul. Note that the bodies of the methods are also parallel regions. Therefore, the threads executing the add and mul methods fork more threads upon entering the parallel region P_2 and P_3. The parallel region P_2 is a fusion and the parallel region P_3 is a single parallel loop where *omp parallel for* is a shorthand for an *omp parallel* with a single *omp for*.

Example 4 Figure 5 shows an OpenMP program using incorrect compiler directives, which results in data races. As there is a data dependence between the two loops, we need a barrier between them when we parallelise the loops. However the clause *schedule(static) nowait* explicitly removes the barrier, which results in an erroneous parallelisation. Using our approach, as a user has to specify iteration contracts for the two loops, we can detect that parallelisation of this program would lead to data races.

2.2 Program specifications: syntax and semantics

Our program specification language is based on permission-based separation logic, combined with the look-and-feel of the java modeling language (JML) [18]. In this way, we exploit the expressiveness and readability of JML, while using the power of separation logic to support thread-modular reasoning. We briefly explain the syntax and semantics of the permission-based separation logic formulas and how they extend the standard JML-program annotations in first-order logic.

Syntax Threads hold permissions to access memory locations. Permissions are encoded by fractional values, as introduced by Boyland [9]: 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

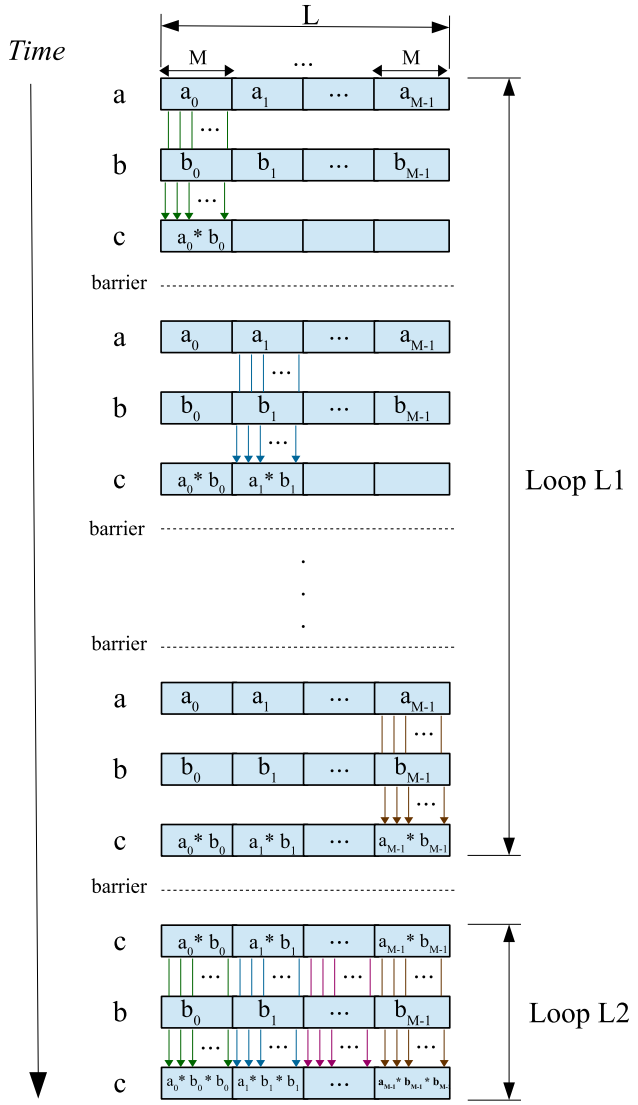


Fig. 3 Thread execution of the program in Fig. 2

```
1  /*@ Program Contract (PC) @*/
2  void main(){
3     int  a[N], b[N], c[N], d[N], e[N];
4     #pragma omp parallel {
5        #pragma omp sections {
6           #pragma omp section
7              add(N, a, b, c, d);
8           #pragma omp section
9              mul(N, a, b, e);
10       }
11    }
12 }
13 void add(int L,int a[],int b[], int c[], int d[]){
14    #pragma omp parallel {
15       #pragma omp for schedule(static) nowait
16       for(int i=0;i<L;i++) //Loop L1
17        /*@ Iteration  Contract 1 (IC1) @*/
18        { c[i]=a[i]; }
19       #pragma omp for schedule(static)
20       for(int i=0;i<L;i++) //Loop L2
21        /*@ Iteration  Contract 2 (IC2) @*/
22        { d[i]=c[i]+b[i]; }
23    }
24 }
25 void mul(int L,int a[],int b[], int c[]){
26    #pragma omp parallel for
27       for(int i=0; i<L; i++) //Loop L3
28        /*@ Iteration  Contract 3 (IC3)  @*/
29        { d[i]=a[i]*b[i]; }
30 }
```

Fig. 4 Parallel regions in OpenMP

```
1  void adm(int L,int M,int a[],int b[], int c[], int d[]){
2     #pragma omp parallel {
3        #pragma omp for schedule(static) nowait
4        for(int i=0;i<L;i++) //Loop L1
5         { c[i]=a[i]*b[i]; }
6        #pragma omp for
7        for(int i=0;i<L;i++) //Loop L2
8         { d[i]=c[i+1]; }
9     }
10 }
```

Fig. 5 A simple OpenMP program that has a data race

of the logic ensures that for every memory location the total sum of permissions over all threads to access this location does not exceed 1. This guarantees that if the permission specifications can be verified, the program is data-race-free. The set of permissions that a thread holds are typically called its *resources*.

Formulas F in our program specification language are built from first-order logic formulas b, permission predicates $\mathsf{Perm}(e_1, e_2)$, 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:

$$F ::= b \mid \mathsf{Perm}(e_1, e_2) \mid b ? F : F \mid F \star F \mid \bigstar_{i \in I} F(i)$$
$$b ::= \textbf{true} \mid \textbf{false} \mid e_1 == e_2 \mid e_1 \leq e_2 \mid \neg b \mid b_1 \wedge b_2 \mid \ldots$$
$$e ::= v \mid n \mid [e] \mid e_1 + e_2 \mid e_1 - e_2 \mid \ldots$$

where b is a side-effect free Boolean expression, e is a side-effect free arithmetic expression, [.] is a unary dereferencing operator—thus $[e]$ returns the value stored in the address e in shared memory—v ranges over variables and n ranges over numerals. We assume the first argument of the $\mathsf{Perm}(e_1, e_2)$ predicate is always an address and the second argument is a fraction. For convenience, we often use the keyword *read* instead of an explicit fraction to specify an arbitrary read

$$\frac{\sigma, h, \pi\,[b\rangle\,\mathsf{true}}{\sigma, h, \pi\,[b\rangle\,\pi_0}\,[\mathbf{Boolean}]$$

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

$$\frac{\sigma, h, \pi\,[b\rangle\,\mathsf{true} \qquad \sigma, h, \pi\,[F_1\rangle\,\pi'}{\sigma, h, \pi\,[b?F_1 : F_2\rangle\,\pi'}\,[\mathbf{Cond\ 1}] \qquad \frac{\sigma, h, \pi\,[b\rangle\,\mathsf{false} \qquad \sigma, h, \pi\,[F_2\rangle\,\pi'}{\sigma, h, \pi\,[b?F_1 : F_2\rangle\,\pi'}\,[\mathbf{Cond\ 2}]$$

$$\frac{\sigma, h, \pi\,[F_1\rangle\,\pi' \qquad \sigma, h, \pi + \pi'\,[F_2\rangle\,\pi''}{\sigma, h, \pi\,[F_1 \star F_2\rangle\,\pi' + \pi''}\,[\mathbf{SepConj}]$$

$$\frac{\forall i \in I : \sigma, h, \pi\,[F(i)\rangle\,\pi_i \qquad \pi + \Sigma_{i\in I}\pi_i \leq 1}{\sigma, h, \pi\,[\bigstar_{i\in I} F(i)\rangle\,\Sigma_{i\in I}\pi_i}\,[\mathbf{USepConj}]$$

Fig. 6 Semantics of formulas in permission-based separation logic

permission, and the keyword *write* instead of 1 to denote a write permission.

We use the array notation $a[e]$ as syntactic sugar for $[a+e]$ where a is a variable containing the base address of the array a and e is the subscript expression; together they point to the address $a + e$ in shared memory.

Semantics Our semantics mixes concepts of implicit dynamic frames [25] and separation logic with fractional permissions, which makes it different from the traditional separation logic semantics and more aligned towards the way separation logic is implemented using traditional first order logic tooling. For further reading on the relationship between separation logic and implicit dynamic frames, we refer to the work of Parkinson and Summers [22].

To define the semantics of formulas, we assume the existence of the following domains: Loc, the set of memory locations, VarName, the set of variable names, Val, the set of all values, including memory locations, and Frac, the set of fractions ([0, 1]).

We define *memory* as a map from locations to values $h : \mathsf{Loc} \rightarrow \mathsf{Val}$. A *memory mask* is a map from locations to fractions $\pi : \mathsf{Loc} \rightarrow \mathsf{Frac}$ with unit element $\pi_0 : l \mapsto 0$ with respect to the point-wise addition of heap masks. A *store* is a function from variable names to values: $\sigma : \mathsf{VarName} \rightarrow \mathsf{Val}$.

Formulas can access the memory directly; the fractional permissions to access the memory are provided by the Perm predicate. A strict form of self-framing is enforced, meaning that the Boolean formulas expressing the functional properties in pre- and postconditions and invariants should be framed by sufficient resources (i.e. there should be sufficient access permissions for the memory locations that are accessed by the Boolean formula, in order to evaluate this formula).

The semantics of an *expression* e depends on a store σ, a memory h, and a memory mask π and yields a value: $\sigma, h, \pi\,[e\rangle\,v$. The store σ and the memory h are used to determine the value v, and the memory mask π is used to determine if the expression is correctly framed, i.e. sufficient access permissions are available. For example, the rule for array access is:

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

where $\sigma(a)$ is the initial address of array a in the memory and i is the array index that is the result of evaluating of index expression e. Apart from the check for correct framing as explained above, the evaluation of expressions is standard and we do not explain it any further.

The semantics of a *formula* F, given in Fig. 6, depends on a store, a memory, and a memory mask and yields a memory mask: $\sigma, h, \pi\,[F\rangle\,\pi'$. The given mask π denotes the permissions by which the formula F is framed. The yielded mask π' denotes the additional permissions provided by the formula. Thus, a Boolean expression is valid if it is true and yields no additional permissions, (rule **Boolean**), while evaluating a $\mathsf{Perm}(e_1, e_2)$ predicate yields additional permissions to the location, provided the expressions e_1 and e_2 are properly framed (rule **Permission**). Note that evaluation of expression e_1 results in a location l, while evaluation of expression e_2 results in a fraction f. The rule checks that the permissions already held on location l plus the additional fraction f does not exceed 1. The rules for evaluation of a conditional formula are standard (rules **Cond 1** and **Cond 2**). We overload standard addition $+$, summation Σ, and comparison operators to be, respectively, used as pointwise addition, summation and comparison over the memory masks. These operators are used in the rules **SepConj** and **USepConj**. In the rule **SepConj**, each formula F_1 and F_2 yields a separate memory mask, π' and π'', respectively, where the final memory mask is calculated by pointwise addition of two memory masks, $\pi' + \pi''$. The rule checks if F_1 is framed by π and F_2 is framed by $\pi + \pi'$. Note that since F_2 is framed by $\pi + \pi'$, this implicitly guarantees that the permissions per location never exceed 1. Finally, the rule **USepConj** extends the similar evaluation by quantifying over a set of formulas conjoined by the universal separating conjunction operator. Again, rule **USepConj** checks that the permission fractions on any location in the memory cannot exceed 1.

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

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

Example 5 Figure 7 presents an example of how we annotate a sequential program using our specification language. The formulas in the annotations are interpreted using the semantics as defined in Fig. 6. The program logic rules are

```
1  /*@ context_everywhere a != NULL && b != NULL && c != NULL && size > 0;
2      context_everywhere \length(a) == size && \length(b) == size && \length(c) == size;
3      context  (\forall*  int k; 0 <= k && k < size; Perm(a[k], read));
4      context  (\forall*  int k; 0 <= k && k < size; Perm(b[k], read));
5      context  (\forall*  int k; 0 <= k && k < size; Perm(c[k], write));
6      ensures  (\forall   int k; 0 <= k && k < size; c[k] == a[k]+b[k]);
7  @*/
8  void add(int a[], int b[], int c[], int size) {
9    int i = 0;
10
11   /*@ loop_invariant  (\forall*  int k; 0 <= k && k < size; Perm(a[k], read));
12       loop_invariant  (\forall*  int k; 0 <= k && k < size; Perm(b[k], read));
13       loop_invariant  (\forall*  int k; 0 <= k && k < size; Perm(c[k], write));
14       loop_invariant  (\forall   int k; 0 <= k && k < i; c[k] == a[k]+b[k]);
15   @*/
16   for(i; i < size; i++)
17     {c[i] = a[i] + b[i];}
18 }
```

Fig. 7 An example of an annotated sequential program

the basic proof rules from separation logic (an extension of Hoare logic).

This sequential program has a loop (lines 11–17) that adds the corresponding elements of two arrays (named a and b) and stores it in a different array (named c) in line 17. Annotations are provided to give a function specification (lines 1–7) and a loop invariants (lines 12–16). Note that *\forall** indicates universal separating conjunction, $\bigstar_{i \in I}$, over permission predicates and *\forall* denotes standard universal conjunction over logical predicates. Preconditions and postconditions, using keywords *requires* and *ensures* (lines 3–6), should hold at the beginning and the end of the function, respectively. We use the keyword *context* to abbreviate both *requires* and *ensures* clause. This is convenient to have, because permission pre- and postconditions are often the same. The keyword *context_everywhere* is used to specify an invariant property (lines 1–2) that must hold throughout the function. As pre- and postcondition, we have read permissions over all elements in arrays a and b (lines 3–4) and write permissions over all elements in array c (line 5). The loop invariants specifies the permissions that are used in the loop (lines 12–14). Further the loop invariant specifies that when iteration i starts, we have added the elements from a and b from the beginning up to location $i-1$ (line 15). Therefore, at the end of the loop (and the function), we have added all elements (specified as a postcondition in line 6).

Parallel Programming Language:

$$
\begin{array}{lll}
\mathcal{P} & ::= & \mathcal{P} \,||\, \mathcal{P} \mid \mathcal{P} \oplus \mathcal{P} \mid \mathcal{P} \fatsemi \mathcal{P} \mid \\
 & & \quad \mathsf{Par(N)\ S} \mid \mathsf{S} \\
\mathsf{S} & ::= & s\mathsf{;S} \mid \mathsf{skip} \\
s & ::= & \mathsf{assg} \mid \ \mathsf{if}\ (b)\ \{\mathsf{S}\}\ \mathsf{else}\ \{\mathsf{S}\} \mid \ \mathsf{while}\ (b)\ \{\mathsf{S}\} \mid \mathsf{Vec(N)\ V} \\
\mathsf{V} & ::= & b \Rightarrow \mathsf{assg;V} \mid \ \mathsf{skip} \\
\mathsf{assg} & ::= & v := e \mid \ gv := e \mid \ v := \mathsf{mem}(e) \mid \ \mathsf{mem}(e) := v \mid \\
 & & \quad [e] := e \\
b & ::= & b \wedge b \mid \ \neg b \mid \ e = e \mid \ e \leq e \mid \ \ldots \\
e & ::= & v \mid \ gv \mid \ N \mid \ e + e \mid \ e \text{ - } e \mid \ [e\,] \mid \ \ldots \\
v & ::= & \text{thread local variables} \\
gv & ::= & \text{program global variables} \\
N & ::= & \text{integer constants}
\end{array}
$$

Fig. 8 Abstract syntax for parallel programming language

3 Syntax and semantics of deterministic parallelism

As mentioned before, we define our verification technique over an intermediate representation language that captures precisely the main features of deterministic parallelism. This section presents the abstract syntax and semantics of PPL, our *Parallel Programming Language*. In Sect. 4, we show how an important fragment of OpenMP can be encoded into this intermediate representation language.

3.1 Syntax

Figure 8 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* $\fatsemi$.

The entry block of the program is the outermost block. *Basic blocks* are:

- a *parallel* block Par (N) S;
- a *vectorised* block Vec (N) 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 *parallelisation level*, S is a sequence of statements and V is a sequence of guarded assignments $b \Rightarrow$ assg.

In the grammar, we define a vectorised block at a different level than the other basic blocks, because this allows us to define the semantics in a more convenient way, while it does not prevent us from writing programs such as the parallel or fusion composition of a parallel and a vectorised block.

We assume a restricted syntax for fusion composition such that its operands are parallel basic blocks with the same parallelisation levels. This is checked by an extra well-formedness condition over PPL programs. Each basic block has a local read-only variable tid $\in$ [0..N) called *thread identifier*, where N is the block's parallelisation level. We (ab)use the term *iteration* to refer to the computations of a single thread in a basic block. So a parallel or vectorised block with parallelisation 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 ($v :=$ mem(e)). A thread may update the heap by writing the value of one of its local variables to it (mem(e):= v). For the arrays, we use notation a[e] as syntactic sugar for [a+e] where a is a variable containing the base address of the array a and e is the subscript expression.

Example 6 Figure 9, line 1 and 2, contains a PPL expression that captures the program in lines 4–13. In this example, the two basic blocks are composed using (||). Figure 10 shows another example of a PPL expression and its corresponding OpenMP program where the basic parallel and vectorised blocks are composed sequentially (lines 1–3). Note that tid_1 refers to the thread identifier of the parallel block, while tid_2 refers to the thread identifier of the vectorised block.

```
1  Par(N)(b[tid] = a[tid] + 1; ) ||
2  Par(N)(c[tid] = a[tid] * 1; )
3
4  void adm(int N,int a[],int b[], int c[]){
5     #pragma omp parallel {
6        #pragma omp for schedule(static) nowait
7        for(int i =0;i<N;i++) //Loop L1
8         { b[i]=a[i]+1; }
9        #pragma omp for
10       for(int i =0; i<N; i++) //Loop L2
11        { c[i]=a[i]*2;  }
12    }
13 }
```

Fig. 9 PPL of an OpenMP program

```
1  Par(N)(c[tid] = a[tid] + b[tid]; ) ;
2  Par(N/M)Vec(M)
3      (c[tid1 * M + tid2] = c[tid1 * M + tid2] * 2; )
4
5  void adm(int N,int M,int a[],int b[], int c[]){
6     #pragma omp parallel {
7        #pragma omp for
8        for(int i =0;i<N;i++) //Loop L1
9         { c[i]=a[i]+b[i];  }
10       #pragma omp for simd simdlen(M)
11       for(int i =0;i<N;i++) //Loop L2
12        { c[i]=c[i]*2;  }
13    }
14 }
```

Fig. 10 PPL of another OpenMP program

3.2 Semantics

The behaviour of PPL programs is described using a small step operational semantics. For a convenient and understandable definition, the operational semantics is defined in several layers, as defined below. Throughout, we assume existence of the finite domains:

- VarName, the set of variable names,
- Val, the set of all values, which includes the memory locations,
- Loc, the set of memory locations, and
- [0..N) for thread identifiers.

We write $+\!\!+$ to concatenate two statement sequences (S$+\!\!+$S). *Program State* To define the program state, we use the following definitions.

$h \in \mathsf{SharedMem} \triangleq \mathsf{Loc} \to \mathsf{Val}$
(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)

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 addresses of arrays).

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 vectorised or a sequential basic block should be executed, and a terminated block state Done.

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

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

The Init state consists of a block statement $\mathcal{P}$. The ParC state consists of two block states, while the SeqC state contains a block state and a block statement $\mathcal{P}$; 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 $\mathbb{LS} \in [0..\mathsf{N}) \to \mathsf{LocalState}$, which maps each thread to its local state, to model the parallel execution of the threads. There are three kinds of local states: a vectorised state Vec, a sequential state Seq, and a terminated sequential state Done.

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

$\mathsf{Vec}(\Sigma, \mathsf{E}, \mathsf{V}, \sigma, \mathsf{S})\mid$	vectorised 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 vectorised 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 vectorised blocks may appear inside a sequential block, keeping σ and S allows continuation of the sequential basic block after termination of the vectorised block. To model vectorised execution, the state contains an auxiliary set $\mathsf{E} \subseteq [0..\mathsf{N})$ that models which threads have already executed the current instruction. Only when E equals [0..N), the next instruction is ready to be executed. Finally, the Seq block state consists of private memory σ and a statement S.

To simplify our notation, each thread receives a copy of the program store as part of its private memory when it initialises. This is captured in rules **Init Par** and **Init Seq** (Fig. 11), where the local store γ is passed as an argument to the Seq block state.

Operational Semantics The operational semantics is defined as a transition relation between program states: $\to_p \subseteq (\mathsf{BlockState} \times \mathsf{Store} \times \mathsf{SharedMem}) \times (\mathsf{BlockState} \times \mathsf{Store} \times \mathsf{SharedMem})$, (Fig. 11), and using an auxiliary transition relation between thread local states: $\to_s \subseteq (\mathsf{LocalState} \times \mathsf{SharedMem}) \times (\mathsf{LocalState} \times \mathsf{SharedMem})$, (Fig. 12), and then a standard transition relation: $\to_{assg} \subseteq (\mathsf{PrivateMem} \times \mathsf{S} \times \mathsf{SharedMem}) \times (\mathsf{PrivateMem} \times \mathsf{SharedMem})$ to evaluate assignments (Fig. 13). 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 $b \in B$:

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

As mentioned, the main transition relation between program states is defined in Fig. 11. Program execution starts in a program state $(\mathsf{Init}(\mathcal{P}), \gamma, h)$ where $\mathcal{P}$ is the program's entry block. Depending on the form of $\mathcal{P}$, a transition is made into an appropriate block state, leaving the heap unchanged (see rules **Init ParC**, **Init SeqC**, **Init Fuse**, **Init Par** and **Init Seq**).

The evaluation of a ParC state non-deterministically evaluates one of its block states (i.e. EB_1 or EB_2), until both blocks are done (rule **ParC Done**).

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, evaluation of the subsequent block is initialised.

Rule **Lift Seq** captures that evaluation of a thread local state is defined in terms of the local thread execution (as defined in Fig. 12). When the local thread state is fully evaluated, this results in a terminated block state (rule **Local Done**).

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, modelled by the mapping $\mathbb{LS}$. A thread in the parallel block terminates if there are no more statements to be executed and a parallel block terminates if all threads executing the block are terminated.

The evaluation of sequential basic block's statements as defined in Fig. 12 is standard except when it contains a vec-

$$\frac{}{\mathsf{Init}(\mathcal{P}_1||\mathcal{P}_2),\gamma,h \rightarrow_p \mathsf{ParC}(\mathsf{Init}(\mathcal{P}_1),\mathsf{Init}(\mathcal{P}_2)),\gamma,h}\ [\textbf{Init ParC}]$$

$$\frac{}{\mathsf{Init}(\mathcal{P}_1;\mathcal{P}_2),\gamma,h \rightarrow_p \mathsf{SeqC}(\mathsf{Init}(\mathcal{P}_1),\mathcal{P}_2),\gamma,h}\ [\textbf{Init SeqC}]$$

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

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

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

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

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

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

Fig. 11 Operational semantics for program execution

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

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

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

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

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

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

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

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

Fig. 12 Operational semantics for thread execution

Fig. 13 Operational semantics for assignments

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

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

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. 14 OpenMP core grammar

torised basic block. A sequential basic block terminates if there is no instruction left to be executed (**Seq Done**). The execution of a vectorised block (defined by the rules **Init Vec**, **Vec Step1**, **Vec Step2**, **Vec Sync** and **Vec Done** in Fig. 12) 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 $b \Rightarrow \mathsf{assg}$). When a thread executes a vector instruction, its thread 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. 13 is standard and does not require further discussion.

4 Encoding OpenMP into PPL

In order to show that PPL indeed captures the core of deterministic parallel programming languages, this section shows how a widely used subset of OpenMP can be encoded into PPL.

4.1 Subset of OpenMP

Figure 14 defines a grammar which captures a commonly used subset of OpenMP [2]. This grammar defines the OpenMP programs that can be encoded into PPL (and thus can be verified using the verification technique presented below).

Our grammar supports the following OpenMP annotations: omp parallel, omp for, omp simd, omp for simd, omp sections, and omp single. Every 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 a program in our OpenMP subset 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 a program in our OpenMP subset or it is a sequential code block SpecS. For our translation, the relevant clauses are simdlen(M), schedule static, and nowait, all other clauses are ignored.

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

encode p = compose (translate p)

(* translate  ::  [OpenMPAnnot x CProg] =>
                    [PPLProg x [OpenMPAnnot]] *)
translate  xs = map match xs

match(omp for clause*, For) = (Par, omp for clause*)
match(omp simd simdlen(M) clause*, For) =
  (WhileVec, omp simd clause*)
match(omp for simd simdlen(M) clause*, For) =
  (ParVec, omp for simd clause*)
match(omp sections clause*, xs) =
  (fold  ||  (map sec xs),  clause*)
match(omp single clause*, x) = (sec x, clause*)

sec(omp parallel clause*, Job+) = encode Job+
sec(SpecS) = Par(1){SpecS}

(* compose :: [PPLProg x  [OpenMPAnnot]] =>
                    (PPLProg x [OpenMPAnnot])*)
compose ys =
        let  p1 = bundle ⊕ fusible ys in
        let  p2 = bundle ||  par_able  p1 in
        fold  ; p2

par_able  (P1,A1) (P2,A2) = nowait(A1)

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

bundle op cond [x]  = [x]
bundle op cond x:y: ys  =
  let  r  = bundle op cond (y: ys)  in
  if  !( cond x y)
  then x :  r
  else  ((op (fst  x)  (fst (head r))),  snd(head r)) :  tail  r
```

Fig. 15 Translation of a commonly used subset of OpenMP programs into PPL programs

4.2 OpenMP to PPL encoding

This section discusses the encoding of OpenMP programs that can be derived from the grammar in Fig. 14 into PPL.

The encoding algorithm is presented in Fig. 15 in a functional programming-like style.

Line 2 to 7 of the algorithm define some syntactic macros of several program patterns, to improve readability of the algorithm. Note that in the macro *ParVec*, tid_1 refers to the thread identifier of the parallel block, while tid_2 refers to the thread identifier of the vectorised block. The algorithm consists of two steps: a recursive *translate* step, and a *compose* step. The translation step recursively encodes all 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 match to the list of input jobs and returns a list of equivalent PPL code block. The input jobs are encoded in the form (A, C) where A is an OpenMP annotation and C is a code block written in C. The translation returns a list of the form $(P, [A])$, where P is the PPL program corresponding to the C code, and $[A]$ are the OpenMP annotations that are needed to decide how to combine this PPL block with the other code blocks. Notice that the resulting PPL program is not necessarily a single basic block. The function match works as follows:

- an OpenMP for annotation for a for-loop is translated into a parallel block;
- an OpenMP simd annotation for a for-loop is translated into a loop of vectorised statements (taking into account the simdlen(M) argument);
- an OpenMP for simd annotation for a for-loop is translated into a parallel composition of several vectorised statements (taking into account the simdlen(M) argument);
- an OpenMP sections annotation is translated into the parallel composition of the individual statements; and
- an OpenMP single annotation encodes the statements in the single block recursively.

The match function uses the function sec which recursively calls match on nested parallel blocks. A sequence of sequential statements with a contract is encoded as a parallel block with a single thread. Notice that in these cases, any nested OpenMP clauses are passed on; therefore, the match function returns a pair of a PPL program and 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 program blocks 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: $\oplus > || > \fatsemi$.

Operator insertion is done by the function bundle (lines 40–44). 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. Notice that ultimately the head of the list x is composed with the head of the recursive call, rather than with the second element of the list. This is okay, because the composition to be applied is determined locally, and not affected by the compositions of the other blocks.

For each composition operator, the conditions are different. The conditions for parallel and fusion compositions are checked by the functions fusible and par_able. As explained in Sect. 2, fusion of two parallel loops L1 and L2 means that the corresponding iterations of L1 and L2 are executed by the same thread without waiting. Therefore, fusion composition is inserted between two consecutive tuples $(P_i, [A_i])$ and $(P_j, [A_j])$ if:

- both $[A_i]$ and $[A_j]$ are single-element lists containing an omp for annotation,
- the clauses of both annotations include schedule(static), and
- the clauses of $[A_i]$ include nowait.[2]

The parallel composition is inserted between any two tuples in the program where the clauses of the first tuple include a 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.

4.3 Example translations

To illustrate the encoding, we discuss the translation of two small OpenMP programs into PPL.

Example 7 To translate the OpenMP program in Fig. 1 (in Sect. 2.1), we first apply the *translate* function to it:

```
1  B1 = translate(omp for schedule(static) nowait,
2                 for(int i=0;i<L;i++){c[i]=a[i];})
3     = (Par(L) (c[tid]=a[tid];),
         [omp for schedule(static) nowait])
```

[2] Note that this condition is independent of whether A_i is actually an omp for annotation.

```
B2 = translate(omp for schedule(static) nowait,
                 for(int i=0;i<L;i++){c[i]=c[i]+b[i];})
   = (Par(L) (c[tid]=c[tid]+b[tid];),
      [omp for schedule(static) nowait])
```

```
B3 = translate(omp for,
                 for(int i=0; i<L; i++){d[i]=a[i]*b[i];})
   = (Par(L) (d[tid]=a[tid]*b[tid];), [omp for])
```

Next, applying the compose function results in the following PPL program:

$$\text{compose}([B_1, B_2, B_3]) = (B_1 \oplus B_2) \,||\, B_3 =$$

$$\big(\underbrace{\text{Par(L) (c[tid]=a[tid];)}}_{B_1} \oplus \underbrace{\text{Par(L) (c[tid]=c[tid]+b[tid];)}}_{B_2}\big) \,||\, \underbrace{\text{Par(L) (d[tid]=a[tid]*b[tid];)}}_{B_3}$$

Example 8 As another example, we translate the OpenMP program in Fig. 2 (in Sect. 2.1) into PPL. First we use the translate function:

```
B1 = translate(omp simd simdlen(M),
                 for(int i=0;i<L;i++){c[i]=a[i]*b[i];})
   = (while(i ∈ [0,L/M)) Vec(M)
      (c[i*M+tid]=a[i*M+tid]*b[i*M+tid];) , [omp simd])
```

```
B2 = translate(omp for simd simdlen(M),
                 for(int i=0;i<L;i++){c[i]=a[i]*b[i];})
   = (Par(L/M) Vec(M)
      (c[tid1*M+tid2]= a[tid1*M+tid2]*b[tid1*M+tid2];) ,
       [omp for simd])
```

Using the compose function on the list with these two pairs results in the following PPL program:

```
compose([B1, B2]) = (B1 ⨟ B2) =
(while(i ∈ [0,L/M)) Vec(M)
      (c[i*M+tid]=a[i*M+tid]*b[i*M+tid];))
                    ⨟
(Par(L/M) Vec(M)
     (c[tid1*M+tid2]=a[tid1*M+tid2]*b[tid1*M+tid2];))
```

5 Verification of basic blocks

The first step of our verification technique deals with the verification of basic blocks. As mentioned above, there are three types of basic blocks: a sequential block, a vectorised block and a parallel block.

$$\frac{\forall i \in [0..\mathsf{N}).\{rc(i) \star P(i)\}\mathsf{S}(i)\{rc(i) \star Q(i)\}}{\{\bigstar_{i=0}^{\mathsf{N}-1}\ rc(i) \star P(i)\}\mathsf{Par}(\mathsf{N})\ \mathsf{S}\{\bigstar_{i=0}^{\mathsf{N}-1}\ rc(i) \star Q(i)\}}\ [\mathbf{ParBlock}]$$

Fig. 16 Proof rule for the verification of a parallel block

For each basic block, we specify an *iteration contract*, which is a contract for each thread executing in the block. Thus, for a sequential block, the iteration contract coincides with a standard block contract (as there is only one thread executing the block), while for parallel and vectorised blocks, the iteration contract specifies the behaviour of one single thread executed in parallel or in lock-step, respectively. We call this an iteration contract, as it corresponds to the specification of a single iteration of a parallelisable or vectorisable block.

5.1 Iteration contracts

An iteration contract consists of: a resource contract *rc(i)*, and a functional contract *fc(i)*, where i is the block's iteration variable. A resource contract indicates the permissions to access memory locations and a functional contract is related to values in the memory locations. Both the resource contract and the functional contract consist of a precondition and a postcondition. We use $P(i)$ to denote the functional precondition, and $Q(i)$ to denote the functional postcondition. In case the resource pre- and postcondition are the same, we simply write $rc(i)$; otherwise, we distinguish them by $rc_{\mathsf{pre}}(i)$ and $rc_{\mathsf{post}}(i)$.

Example 9 Consider the PPL program in Example 7. An iteration contract for basic block B_1 would be:

```
/*@ requires Perm(c[tid], write) ** Perm(a[tid], read);
    ensures Perm(c[tid], write) ** Perm(a[tid], read);
    ensures c[tid]==a[tid];
@*/
```

where the first two lines show a resource contract and the last line indicates a functional contract. Note that ** is the ASCII-notation for $\star$.

5.2 Verification rules for basic blocks

As mentioned above, a sequential block is executed by a single thread, thus its iteration contract coincides with its block contract, and no special verification rule is needed.

Parallel basic blocks are verified by the rule **ParBlock** presented in Fig. 16, where $\mathsf{S}(i)$ is the body of the i^{th} iteration of the parallel basic block. This rule states that if each single thread respects its iteration contract, the contract for the basic block is composed by the universal separating conjunction

of the iteration contract's precondition and postcondition, respectively. As the threads execute completely independently, there is no permission transfer, and the resource pre- and postcondition coincide. Notice further that soundness of this rule implies that all threads in a parallel block must be independent, because otherwise the universal separating conjunction would not be satisfiable.

For vectorised blocks, the **ParBlock** rule can be used in case there are no inter-iteration data dependencies. If there are inter-iteration data-dependencies, we need to provide extra annotations that indicate how permissions are transferred inside the vectorised block. In a vectorised block, implicitly all threads synchronise between every instruction. During such a synchronisation, permissions may be transferred from the iteration containing the source of a dependence to the iteration containing the sink of that dependence. To specify such a transfer we introduce send and receive ghost statements.[3] Remember that according to the PPL grammar, the body of a vectorised block is a sequence of guarded assignments $b \Rightarrow \mathsf{assg}$. A guard $b_s(i)$ denotes the guard of statement s in iteration i.

$$\begin{aligned} &\text{//@ } L_s\text{: if}(b_s(i))\,\{\,\text{send } \phi(i) \text{ to } L_r,\ d;\ \}\\ &\text{//@ } L_r\text{: if}(b_r(i))\,\{\,\text{receive } \psi(i) \text{ from } L_s,\ d;\ \}\end{aligned}$$

A send annotation specifies that at label L_s, if a guard $b_s(i)$ is true, the permissions and properties denoted by formula ϕ are transferred to the statement labelled L_r in iteration $i+d$, where i is the current iteration and d is the distance of dependence. A receive annotation specifies that the permissions and properties denoted by formula ψ are received by the current iteration from iteration $i-d$. These annotations always come in pairs. In practice, the information provided by either the send or receive annotation is sufficient to infer the other. Therefore, to reduce the annotation overhead, optionally only one of them has to be provided by the developer. However, by providing them both, we make the specifications easier to understand.

Example 10 Suppose we have a basic block

```
Vec(N)(x[tid + 1] = tid; a[tid] = x[tid] + 3;)
```

where N − 1 == x.length. We can verify that this block annotated with send and receive respects the following iteration contract:

```
/*@ requires N − 1 == x.length;
    requires Perm(x[tid + 1], write) ** Perm(a[tid], write);
    requires tid == 0 ==> Perm(x[tid], write);
    ensures Perm(x[tid], write) ** Perm(a[tid], write);
    ensures tid == N − 1 ==> Perm(x[tid+ 1], write);
    ensures tid > 0 ==> a[tid] = tid − 1 + 3;
@*/
```

[3] Ghost statements are specification-only statements. They are not part of the program, but are used purely for verification purposes.

$$\frac{\forall i \in [0..\mathsf{N}).\{rc_{\mathsf{pre}}(i) \star P(i)\}\mathsf{S}(i)\{rc_{\mathsf{post}}(i) \star Q(i)\}}{\{\bigstar_{i=0}^{\mathsf{N}-1}\ rc_{\mathsf{pre}}(i) \star P(i)\}\mathsf{Vec(N)}\ \mathsf{V}\{\bigstar_{i=0}^{\mathsf{N}-1}\ rc_{\mathsf{post}}(i) \star Q(i)\}}\ [\mathbf{VecBlock}]$$

Fig. 17 Proof rule for the verification of vectorised block

```
Vec(N)
(
x[tid + 1] = tid;
//@ L1: if(tid < N − 1)
send Perm(x[tid + 1], write) **
x[tid + 1] == tid
to L2, 1;
//@ L2: if(tid > 0)
receive Perm(x[tid], write) **
x[tid] == tid − 1
from L1, 1;
a[i] = x[tid] + 3;
)
```

In order to verify this example, we need a proof rule for vectorised blocks, as well as for the send and receive ghost statements.

The rule for the verification of vectorised blocks is given in Fig. 17. It is similar in spirit to the **ParBlock** rule, but does not require the resource pre- and postcondition to be the same.

The rules for the send and receive ghost statements are similar in spirit to the rules that are typically used for permission transfer upon lock acquiring and release (see e.g. [15]). In particular, send is used to give up resources that the receive acquires. This is captured by the following two proof rules:

$$\begin{array}{c}\dfrac{}{\{P\}\ \mathsf{send}\ P\ \mathsf{to}\ L, d\ \{\mathsf{true}\}}\ [\mathbf{send}]\\[2ex] \dfrac{}{\{\mathsf{true}\}\ \mathsf{receive}\ P\ \mathsf{from}\ L, d\ \{P\}}\ [\mathbf{receive}]\end{array} \tag{1}$$

Receiving permissions and properties that were not sent is unsound. Therefore, send and receive annotations have to be properly matched, meaning that:

(i) send and receive annotations always come in pairs;
(ii) if the receive is enabled in iteration j, then d iterations earlier, the send should be enabled, i.e.,

$$\forall j \in [0..N).b_r(j) \implies j \geq d \wedge b_s(j-d) \tag{2}$$

(iii) the information and resources received should be implied by those sent:

$$\forall j \in [d..N).\phi(j-d) \implies \psi(j) \tag{3}$$

In other words, the rules in Eq. 1 cannot be used unless the syntactic criterion (i) and the proof obligations (ii) and (iii) hold.

5.3 Soundness

This section discusses the soundness of the proof rules **ParBlock** and **VecBlock** above. To show soundness of these rules, we have to show that in order to prove correctness of a parallel or vectorised block, it is sufficient to reason about the body of the block, and to prove independence or inter-iteration data dependence of that body. As always, the interpretation of a Hoare triple $\{P\}S\{Q\}$ is the following: if the precondition P holds in a state s, and if execution of statement S from state s terminates in a state s', then the postcondition Q holds in this state s'. As the proof rules are adapted from the proof rules for parallel and vectorised loops presented in [5], the soundness argument is also similar.

To construct the proof, we define the set of possible *execution traces* of atomic steps over the vectorised and parallel blocks. In addition, we also define the *instrumented sequentialised execution traces* for those blocks, which are the executions (1) if all iterations are executed in order and (2) such that validity of each iteration contract is checked for each separate iteration.

To prove soundness of the rule **ParBlock**, we show that the all execution traces of this statement are equivalent to the instrumented sequentialised execution trace of the parallel block. To prove soundness of the rule **VecBlock**, we show that all execution traces of this statement are equivalent to the instrumented sequentialised execution trace of the vectorised block.

Functional equivalence of the two traces is shown by transforming the computations in one trace into the computations in the other trace by swapping adjacent independent execution steps.

5.3.1 Denotational semantics of blocks

To phrase the soundness proof, we prefer to use a denotational semantics for the parallel and vectorised blocks, where the semantic domain is a set of traces, seen as sequences of instructions. The denotational semantics that is defined in this section is equivalent to the operational semantics as defined in Sect. 3, but the proof is omitted from the paper. We develop our formalisation for non-nested blocks with K guarded statements. We instantiate the block body for each iteration of the block; thus, we have $(L_i^j\colon \mathsf{if}(b_i^j)\ I_i^j;)$ as the instantiation of the i^{th} instruction in the j^{th} iteration of the block. We refer to this instance of statements as S_i^j.

Definition 1 The semantics of a statement instance $[\![S_i^j]\!]$ is defined as the atomic execution of the instruction I_i^j labelled by L_i^j provided its guard condition b_i^j holds; otherwise, it behaves as a skip.

Definition 2 An *execution trace* c is a finite sequence $t_1, t_2, \ldots, t_m$ of statement instances such that t_1 is executed first, then t_2 is executed and so on until the last statement t_m. We write ϵ for an *empty* execution trace.

To characterise the set of execution traces for parallel and vectorised blocks, we define auxiliary operators *concatenation* and *interleaving*.

First, we define two versions of concatenation, *plain concatenation* ($++$) and *synchronised concatenation* (#).

Definition 3 The *plain concatenation* ($++$) operator is defined as $C_1 ++ C_2 = \{c_1 \cdot c_2 \mid c_1 \in C_1 \ \wedge c_2 \in C_2\}$.

Plain concatenation takes two sets of execution traces and creates a new set that concatenates all execution traces in the first set with all execution traces in the second set.

Definition 4 The *synchronised concatenation* (#) operator inserts a barrier b between the execution traces. It is defined as $C_1 \# C_2 = \{c_1 \cdot b \cdot c_2 \mid c_1 \in C_1 \ \wedge c_2 \in C_2\}$.

The intuition here is that the insertion of a barrier b indicates an implicit synchronisation point. When defining the interleaving of traces, the barrier restricts what interleavings are possible.

We lift concatenation to multiple sets as follows:

$$\begin{aligned}\mathsf{Concat}_{i=1}^{N} C_i &= C_1 ++ \cdots ++ C_N\\ \mathsf{SyncConcat}_{i=1}^{N} C_i &= C_1 \# \cdots \# C_N\end{aligned}$$

Next, interleaving defines how to weave several execution traces into a single execution trace. This uses a *happens-before* order $<$, in order not to violate restrictions imposed by the program semantics. This happens-before order $<$ is defined such that it maintains *program order* (PO), i.e. it maintains the order of statements executed by the same thread, and it also maintains synchronisation order (SO), i.e. it maintains the order between a barrier and the statements preceding and following it.

To define the interleaving operator ($\mathsf{Interleave}$), we first define an auxiliary operator $\mathsf{Interleave}^i$ that denotes interleaving with a fixed first statement s of thread i:

$$\begin{aligned}&\mathsf{Interleave}^i(\epsilon, \cdots, \epsilon) = \{\epsilon\}\\ &\mathsf{Interleave}^i(c_1, \cdots, c_{i-1}, \epsilon, c_{i+1}, \cdots, c_N) = \emptyset, \text{ if } \exists c_{j\neq i} \neq \epsilon\\ &\mathsf{Interleave}^i(c_1, \cdots, c_{i-1}, s \cdot c_i{}', c_{i+1}, \cdots, c_N) =\\ &\quad \{s_1 \cdot x \mid x \in \mathsf{Interleave}(c_1, \cdots, c_{i-1}, c_i{}', c_{i+1}, \cdots, c_N) \wedge\\ &\qquad \nexists s' \in x.s' < s\}\end{aligned}$$

If the complete execution trace of thread i has been interleaved, there are two possible cases. If all other threads are also done, then this returns an empty execution trace (as a base case). If any other thread can still take a step, then this call for thread i returns an empty set of interleavings. If thread

i has a non-empty execution trace to interleave, i.e. it is of the form $s_1 \cdot c_{i'}$, then we obtain all interleavings that start with s_1, extended with the (recursive) interleaving of all other execution traces and the remainder of this execution trace $c_{i'}$. Note that this extension is only allowed if it does not violate the happens-before order $<$. Next we define the full interleaving operator, which basically considers all interleavings for all threads.

$$\begin{aligned}&\mathsf{Interleave}^{i=1..N} c_i = \\ &\quad \mathsf{Interleave}(c_1, \cdots, c_N) = \\ &\quad \bigcup_{i=1}^{N} \mathsf{Interleave}^{i}(c_1, \cdots, c_N)\end{aligned}$$

Now we can define the denotational semantics of parallel and vectorised blocks. The semantics of a parallel block is any interleaving of all statement instances that preserve the program order PO. The semantics of a vectorised block is any interleaving of the *synchronised concatenation* of the execution traces of the individual traces, thus with an implicit barrier added after the execution steps of each statement instance. Formally, these are defined as follows.

Definition 5 The denotational semantics of a parallel block is defined as

$$[\![Par(N)S]\!] = \mathsf{Interleave}^{j=1..N}\mathsf{Concat}_{i=1}^{K}[\![S_i^j]\!]$$

Definition 6 The denotational semantics of a vectorised block is defined as

$$[\![Vec(N)S]\!] = \mathsf{Interleave}^{j=1..N}\mathsf{SynchConcat}_{i=1}^{K}[\![S_i^j]\!]$$

Next, we define the *sequentialised execution trace* of a parallel and vectorised block. This is the sequential execution of all iterations in a parallel and vectorised block.

Definition 7 The sequential execution trace of a parallel and vectorised block is

$$\begin{aligned}[\![Par(N)S]\!]^{Seq} &= \mathsf{Concat}_{j=1}^{N}\mathsf{Concat}_{i=1}^{K}[\![S_i^j]\!] \\ [\![Vec(N)S]\!]^{Seq} &= \mathsf{Concat}_{j=1}^{N}\mathsf{SynchConcat}_{i=1}^{K}[\![S_i^j]\!]\end{aligned}$$

Finally, we define the *instrumented sequentialised execution trace* of a parallel and vectorised block. This is the sequential execution of all iterations, where in addition all precondition and postcondition are checked. Below we will show that all parallel and vectorised execution traces are equivalent to this instrumented sequentialised execution trace.

Definition 8 The *instrumented sequentialised execution traces* of a parallel and vectorised block are

$$\begin{aligned}[\![Par(N)S]\!]_{Spec}^{Seq} = \mathsf{Concat}_{j=1}^{N}(&\mathsf{Assert}\ rc(j) \star P(j) \mathbin{+\!\!+} \\ &\mathsf{Concat}_{i=1}^{K}[\![S_i^j]\!] \mathbin{+\!\!+} \\ &\mathsf{Assert}\ rc(j) \star Q(j))\end{aligned}$$

$$\begin{aligned}[\![Vec(N)S]\!]_{Spec}^{Seq} = \mathsf{Concat}_{j=1}^{N}(&\mathsf{Assert}\ rc(j) \star P(j) \mathbin{+\!\!+} \\ &\mathsf{SynchConcat}_{i=1}^{K}[\![S_i^j]\!] \mathbin{+\!\!+} \\ &\mathsf{Assert}\ rc(j) \star Q(j))\end{aligned}$$

where Assert checks the pre- and postcondition before and after each iteration. If the asserted property ϕ holds, Assert ϕ behaves as a skip; otherwise, it aborts (i.e. there is no execution). Note that the sequential execution trace is in happens-before order.

5.3.2 Correctness of parallel blocks

In the previous section, we defined a denotational semantics of parallel and vectorised blocks in terms of possible traces of atomic steps. In addition, we defined the instrumented sequentialised execution of parallel and vectorised blocks. Now, we argue correctness of the rules for parallel and vectorised blocks (Figs. 16 and 17).

We prove that every execution trace in [![Par(N) S]!] is functionally equivalent to the single execution trace [![Par(N) S]!]$_{Spec}^{Seq}$ if all contracts hold, by showing that any execution trace can be reordered until it is the sequential execution order.

Theorem 1 *All execution traces in* [![Par(N) S]!] *and* [![Par(N) S]!]$_{Spec}^{Seq}$ *are functionally equivalent only if all contracts hold.*

Proof sketch 1 Assume that the first n steps of the given execution trace are in the same order as the sequential execution trace. Then, step t_{n+1} in the sequential execution has to be somewhere in the given sequence. Because each sequence contains the same steps and the sequential execution trace is in happens-before order, all the steps that have to happen before t_{n+1} are already included in the prefix. Hence, in the given sequence, all the steps between the end of the prefix and t_{n+1} are independent of step t_{n+1} itself. Therefore, step t_{n+1} can be swapped with all these intermediate steps. We then repeat until the whole sequence matches.

We proved that any legal execution trace of parallel block can be reordered into the sequential one, i.e. [![Par(N) S]!] = $S_0 \star S_1 \star S_2 \star \ldots \star S_N$. Now suppose that in the initial state $P_0 \star P_1 \star \ldots \star P_N$ holds. Since all instructions are independent, after the execution of S_0, Q_0 holds and $P_1 \star P_2 \star \ldots \star P_N$ is preserved. After the execution of S_1, Q_1 holds and $P_2 \star P_3 \star \ldots \star P_N$ is preserved. Moreover, S_1 will not make Q_0 invalid. After the execution of S_2, Q_2 holds and $P_3 \star P_4 \star \ldots \star P_N$ is preserved. In addition, S_2 will not make

$Q_0 \star Q_1$ invalid. By continuing in this way, in the final state of the execution trace $Q_0 \star Q_1 \star \ldots \star Q_N$ holds. Therefore, we can conclude for any legal execution trace in $[\![\text{Par(N) } S]\!]$ starting in the precondition, the postcondition will hold for the final state.

As a corollary of Theorem 1, we can also conclude that all executions in $[\![\text{Par(N) } S]\!]$ are data-race-free. We can apply the same argument for the vectorised blocks, but as the vectorised blocks is defined in terms of SynchConcat, swapping past barriers is never necessary.

Theorem 2 *All execution traces in* $[\![\text{Vec(N) } S]\!]$ *and* $[\![\text{Vec(N) } S]\!]^{Seq}_{Spec}$ *are functionally equivalent.*

Note that the sequentialised instrumented execution trace now also contains send/receive ghost annotations and barriers between each iteration.

6 Verification of block composition

Now that we have seen how correctness of a basic block can be verified in isolation, the next step is to verify their composition. We show how this can be done on the basis of the block iteration contracts only, by proving that all 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.

We first discuss how we can verify programs where the resources in the iteration contracts are constant, i.e. the resource pre- and postconditions are always the same. Next, we sketch how to extend the approach to the case where the resource pre- and postconditions of an iteration contract differ.

6.1 Verification of block composition without resource transfers

As mentioned above, we first assume that each basic block of a program is specified by an iteration contract with constant resources $rc(i)$ for iteration i. Further, we assume that the program is 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 $\mathbb{B}_{\mathcal{P}} = \{b_1, b_2, \ldots, b_n\}$, as the finite set of basic block labels of the program $\mathcal{P}$. For a basic block b with parallelisation level m, we define a finite set of iteration labels $I_b = \{0^b, 1^b, \ldots, (m-1)^b\}$ 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 that are not ordered sequentially, the *incomparable iteration pairs*, $\mathfrak{I}^{\mathcal{P}}_{\perp}$ as:

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

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:

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

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 . 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 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.

Example 11 As an example, the *b-linearisation* of the PPL in Example 7 is as follows:

```
(Par(L) (c[tid]=a[tid];)⨟Par(L) (c[tid]=c[tid]+b[tid];))
                        ⨟
          Par(L) (d[tid]=a[tid]*b[tid];)
```

Fig. 18 Proof rule for b-linearisation reduction of PPL programs

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

Figure 18 presents the rule **b-linearise**. In this 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 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 rules discussed above) and (2) proving the correctness of $blin(\mathcal{P})$ against the program contract.

6.2 Soundness

Now we are ready to show that a PPL program with provably correct iteration contracts and a global contract that is provable in our logic (including the rule **b-linearise**) is indeed data race free and functionally correct w.r.t. its specifications. To show this, we prove (i) soundness of the **b-linearise** rule and (ii) that each verified program is free of data races.

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 normalised w.r.t. program order and second that any normalised execution can be mapped to a b-linearised execution. To formalise this argument, we first define: an *execution*, an *instrumented execution*, and a *normalised 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 9 (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 mas-ks* (*memory 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 a read and 1 is a 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 initialisation state with heap mask π an extra premise should be discharged, which states that there are $n \geq 2$ heap masks $\pi_1, \ldots, \pi_n$, one for each newly initialised 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, Fig. 19 presents the instrumented versions of the rules **Init ParC**, **ParC Done**, **rdsh**, and **wrsh**, where $\rightarrow_{p,i}$ and $\rightarrow_{assg,i}$ denote program and assignment transition relations in the instrumented semantics, respectively. If a transition cannot satisfy its premises, it blocks.

Definition 10 (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).* $\forall(i^b, j^{b'}) \in \mathfrak{I}^{\mathcal{P}}_{\perp}.RC_{\mathcal{P}} \rightarrow rc_b(i) \star rc_{b'}(j)$ *and (2).* $\forall b \in \mathbb{B}_{\mathcal{P}}.\{\bigstar_{i \in [0..N_b)} rc_b(i)\} \mathcal{P}_b \{\bigstar_{i \in [0..N_b)} rc_b(i)\}$ *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 sketch 2 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 vectorised basic blocks are non-conflicting. This implies that for an arbitrary **Init Par** or **Init Vec** transition which initialises a basic

Fig. 19 The instrumented versions of the rules **Init ParC**, **ParC Done**, **rdsh**, and **wrsh**

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

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

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

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

block b, there exists $\pi_1, \ldots, \pi_n$ such that $\Sigma_i^n \pi_i \leq \pi_b$ holds in b's initialisation 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 sketch 3 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 normalised execution is an instrumented execution that respects the program order, which is defined using an auxiliary labelling function $\mathcal{L} : \mathbb{T} \rightarrow \mathbb{B}_{\mathcal{P}}^{all} \times \mathbb{L}$ where $\mathbb{T}$ is the set of all transitions, $\mathbb{L}$ is the set of labels $\{I, C, T\}$, and $\mathbb{B}_{\mathcal{P}}^{all}$ is the set of block labels (including both composite and basic block labels).

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

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

Definition 11 (Normalised Execution). An instrumented execution labelled by $\mathcal{L}$ is *normalised* if the labels of its transitions are in non-decreasing order.

We transform an instrumented execution to a normalised 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 normalised execution such that they both end in the same terminal state.*

Proof sketch 4 Given an instrumented execution $IE = IE_1 : (s_1, t_1) : (s_2, t_2) : IE_2$, if $\mathcal{L}(t_1) > \mathcal{L}(t_2)$, a state s_x exists such that a new instrumented execution $IE' = IE_1 : (s_1, t_2) : (s_x, t_1) : IE_2$ can be constructed by swapping two adjacent transitions t_1 and t_2. As the swap is on an instrumented execution, from Lemma 2 we know that this is data-race-free, thus 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 normalised execution of IE obtained by applying a finite number of such swaps yields the same terminal state as IE. □

Lemma 4 *For each normalised 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 sketch 5 An execution of $blin(\mathcal{P})$ is constructed by applying the map $\mathcal{M} : \mathsf{BlockState} \rightarrow \mathsf{BlockState}$ to each state of a normalised execution. $\mathcal{M}$ is defined as:

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

where $\mathbb{LS}_2^0$ is the initial mapping of thread local states of $\mathcal{P}_2$ 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 12 (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 2.2.

Theorem 3 *The rule* ***b-linearise*** *is sound.*

Proof sketch 6 Assume that (1). $\forall(i^b, j^{b'}) \in \mathfrak{I}_{\perp}^{\mathcal{P}}.RC_{\mathcal{P}} \rightarrow rc_b(i) \star rc_{b'}(j)$ and (2). $\{RC_{\mathcal{P}} \star P_{\mathcal{P}}\}\ blin(\mathcal{P})\{RC_{\mathcal{P}} \star Q_{\mathcal{P}}\}$. From assumption (2) and the soundness of the program logic used to prove it [5], we conclude (3). $\forall b \in \mathbb{B}_{\mathcal{P}}.\ \{\bigstar_{i\in[0..N_b)} rc_b(i)\}\mathcal{P}_b\ \{\bigstar_{i\in[0..N_b)} rc_b(i)\}$. 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 [5], $\{RC_{\mathcal{P}} \star Q_{\mathcal{P}}\}$ 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 sketch 7 Given a program $\mathcal{P}$, with the same reasoning steps mentioned in Theorem 3, 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. □

6.3 Verification of block composition with resource transfers

Next we look at how to adapt this rule in case there are intra-block dependencies; thus, the resource pre- and postconditions of individual iterations are different, and we need send/receive annotations in order to verify the blocks.

This makes the independence check more involved: instead of just checking that the resource contracts for independent iterations are non-conflicting ($\forall(i^b, j^{b'}) \in \mathfrak{I}_{\perp}^{\mathcal{P}}.(RC_{\mathcal{P}} \rightarrow rc_b(i) \star rc_{b'}(j))$), we now need to check the absence of conflicts for all combinations of resource pre- and

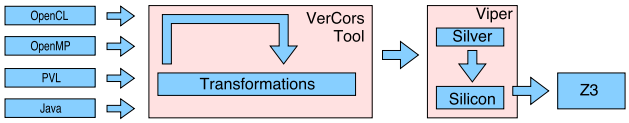


Fig. 20 VerCors tool set overall architecture

postconditions. In case there is only a single resource transfer, we can replace this condition in the rule **b-linearise** by the following condition:

$$\begin{aligned}\forall(i^b, j^{b'}) \in \mathfrak{I}_{\perp}^{\mathcal{P}}.(RC_{\mathcal{P}} \rightarrow\ & rc_{\mathsf{pre},b}(i) \star rc_{\mathsf{pre},b'}(j) \wedge \\ & rc_{\mathsf{pre},b}(i) \star rc_{\mathsf{post},b'}(j) \wedge \\ & rc_{\mathsf{post},b}(i) \star rc_{\mathsf{pre},b'}(j) \wedge \\ & rc_{\mathsf{post},b}(i) \star rc_{\mathsf{post},b'}(j))\end{aligned}$$

This new version of the rule **b-linearise** is sound, because:

1. the check guarantees that the resource precondition of iteration i is disjoint from the resource pre- and postcondition of iteration j;
2. the check also guarantees that the resource postcondition of iteration i is disjoint from the resource pre- and postcondition of iteration j;
3. the resources specified in the resource precondition of iteration i either are send to another iteration (say k) in the same block or they should be part of the resource postcondition of iteration i. The rule guarantees that it will also be checked that the resource pre- and postconditions of iteration k are disjoint from the resource pre- and postconditions of iteration j (because if i and j are independent, then also k and j will be independent.

However, if multiple resource transfers happen within a block, it can happen that at an intermediate point in the block, the thread holds more permissions than it holds at the beginning and the end of the block. To address this, we need to define the *intermediate maximal resource contract* for an intermediate statement S as the universal separating conjunction of the iteration's precondition, and all the resources that are received by all statements that happen-before S. Absence of conflicts is then defined as a check over all intermediate resource contracts. It is future work to define this formally.

7 Tool support

As mentioned above, our verification technique is supported by the VerCors program verifier.[4] This section briefly discusses how our approach is implemented in VerCors.

[4] The tool and a list of case studies and verified examples is available at: https://github.com/utwente-fmt/vercors.

VerCors is a verifier to specify and verify (concurrent and parallel) programs written in a high-level language such as (subsets of) Java, C, OpenCL, OpenMP and PVL, where PVL is VerCors' internal language for prototyping new features. The programs are annotated with pre-/postconditions in permission-based separation logic [1,6]. Then, VerCors encodes annotated programs via several program transformation steps into the intermediate representation language (Silver) of the Viper framework [19,26], and then the encoded program is verified using the Viper technology (Fig. 20).

Using this approach, OpenMP programs are verified with VerCors in the following steps:

1. Specify the OpenMP program (i.e. provide an iteration contract for each block and write the program contract for the outermost OpenMP parallel region.
2. Encode the specified OpenMP program into its PPL counterpart (carrying along the original OpenMP specifications) (as discussed in Sect. 4).
3. Check the PPL program against its specifications, by transforming the PPL program into a Viper program.

Steps 2 and 3 are fully automatic, the user only has to provide the specifications for the OpenMP program. This section provides more details about the encoding of PPL programs into Viper.

7.1 Encoding of basic blocks into viper

To verify our iteration contracts using Viper, we encode the behaviour of the basic blocks and the send/receive annotations as method contracts. The idea is that every block annotated with an iteration contract is encoded by a call to the method basic_block, whose contract encodes the application of the suitable Hoare Logic rule for basic blocks, instantiated for the specific iteration contract.

```
/*@ requires (\ forall*  int j; 0<=j && j<N; pre(j));
    ensures (\forall* int j; 0<=j && j<N; post(j));
@*/
basic_block(int N, free(S));
```

We also need to verify that every iteration respects the iteration contract. This is encoded by a method, parametrised by the thread identifier, containing the basic block's body, and specified by the iteration contract.

```
/*@ requires (0<=j && j<N) ** pre(j);
    ensures post(j);
@*/
block_body(int j,int N, free(S)){body;}
```

Within the body of the basic block there may be send and receive statements.

```
// @ L_s: if (b_s(j)) { send φ(j) to L_r, d;}
// @ L_r: if (b_r(j)) { receive ψ(j) from L_s, d;}
```

The guards are untouched, but the statements are replaced by method calls

```
// @ L_s: if (b_s(j)) { send_s_to_r(j,N,free(φ(j));}
// @ L_r: if (b_r(j)) { receive_r_from_s(j,N,free(ψ(j));}
```

where

```
requires φ(j);
send_s_to_r(int j, int N, free(S));
ensures ψ(j);
receive_r_from_s(int j, int N, free(S));
```

Finally, we need to check that the proof obligations in Eq. 2 and 3 hold.

7.2 Encoding of the b-linearise rule into viper

Finally, for the verification of block composition, we implemented the rule **b-linearize** as part of the encoding into Viper. This means we implemented in VerCors:

- a function to compute the set $\mathfrak{I}_{\perp}^{\mathcal{P}}$, and
- the program transformation $blin$, resulting in a Viper program called blin_program().

This implementation basically follows the formal definition as presented above in Sect. 6.

Next, as part of the Viper encoding, we encode the first proof obligation

$$\forall(i^b, j^{b'}) \in \mathfrak{I}_{\perp}^{\mathcal{P}}.(RC_{\mathcal{P}} \rightarrow rc_b(i) \star rc_{b'}(j))$$

as lemmas for all independent iteration pairs $(i^b, j^{b'})$, i.e. these are encoded as specifications for empty method bodies of the following form:

```
/*@ requires RC_P;
    ensures rc_b(i) ⋆ rc_b'(j);
@*/
indep_iteration (b,b',i,j );
```

Finally, for the b-linearised program, we prove that it satisfies the global method specification.

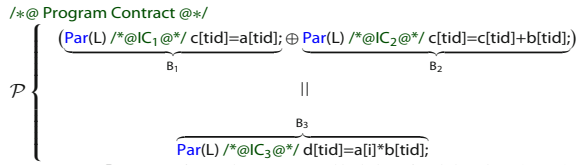
```
/*@ requires {RC_P ⋆ P_P};
    ensures {RC_P ⋆ Q_P}
@*/
blin_program();
```

8 Example: verification of an OpenMP program

To conclude, we show how our verification technique for PPL and the encoding of OpenMP into PPL can be used to verify OpenMP programs. As mentioned above, our approach

requires the user to specify a program contract and an iteration contract for each SpecS block in the OpenMP program, from which all the required PPL contracts can be obtained. We demonstrate this in detail on two of the OpenMP programs presented in Sect. 2.1, which are successfully verified by VerCors.

Figure 21 shows the required contracts for the example discussed in Fig. 1 (in Sect. 2.1). There are four specifications. The first one is the program contract attached to the outermost parallel block. The other contracts are the iteration contracts of the loops L1, L2 and L3, where the *context* keyword is used as a shorthand notation for both requiring and ensuring the same predicate, and \forall$*$ denotes the universal separating conjunction $\bigstar_{i \in I}$. Example 7 already showed how this OpenMP program was encoded into PPL. After adding the annotations in Fig. 21 to the OpenMP program, VerCors generates the following PPL program $\mathcal{P}$:



Program $\mathcal{P}$ contains three parallel basic blocks B_1, B_2 and B_3. The fusion of B_1 and B_2 creates a composite block that is enclosed by the parentheses. Then, the composite block is composed with the basic block B_3 using the parallel composition operator. It is verified by discharging two proof obligations:

1. prove 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, and
2. prove 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 correctness of the b-linearised variant of $\mathcal{P}$ against its program contract $\{RC_{\mathcal{P}} \star P_{\mathcal{P}}\}\ \mathsf{B}_1 \fatsemi \mathsf{B}_2 \fatsemi \mathsf{B}_3 \{RC_{\mathcal{P}} \star Q_{\mathcal{P}}\}$.

Figure 22 illustrates the necessary contract for the other example in Sect. 2.1 (Fig. 2). We have implemented a slightly more general variant of PPL in our VerCors tool, which 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. Our implementation successfully verified the example in 25 seconds.

9 Related work

Botincan et al. propose a proof-directed parallelisation synthesis, which takes as input a sequential program with a proof in separation logic and outputs a parallelised counterpart by inserting barrier synchronisations [7,8]. Hurlin uses a proof-rewriting method to parallelise a sequential program's proof [16]. Compared to them, we prove the correctness of parallelisation 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 nondeterministic program (obtained by naive parallelisation of a sequential program) deterministic by inserting barriers [23]. This technique over-approximates the possible program behaviours which ends up in a determinisation 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 parallelisation annotations can safely be applied and the parallelised program is functionally correct and exhibits the same behaviour as its sequential counterpart.

Barthe et al. synthesise SIMD code given pre- and postconditions for loop kernels in C++ STL or C# BCL [3]. We alternatively enable verification of SIMD loops, by encoding them into vectorised basic blocks. Moreover, we address the parallel or sequential composition of those loops with other forms of parallelised blocks.

Dodds et al. introduce a *higher-order variant* of concurrent abstract predicates (CAP) to support modular verification of synchronisation constructs for deterministic parallelism [13]. 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. As mentioned in the paper, the reasoning is unsound in certain corner cases, which was fixed in an expanded version of their paper using iCAP [14]. Their approach relies on a powerful program logic and focuses much less on automation of the verification process.

Salamanca et al. [24] propose a run-time loop-carried dependence checker as an extension to OpenMP which helps programmers to detect hidden data dependencies in *omp parallel for*. Compared to them, we statically detect any violation of data dependencies without any run-time overhead and we address a larger subset of OpenMP constructs.

Bubel et al. [10] provide a formal trace semantics for data dependences and a program logic to analyse and reason about dependences in imperative programming languages. They benefit from ghost variables to extend the program states to keep track of heap memories. The authors implement their

Program Contract (PC):

```
/*@ context_everywhere a != NULL && b != NULL && c != NULL && d != NULL && L>0;
    context_everywhere \length(a)==L && \length(b)==L && \length(c)==L && \length(d)==L;
    context (\forall* int k; 0 <= k && k < L; Perm(a[k], read));
    context (\forall* int k; 0 <= k && k < L; Perm(b[k], read));
    context (\forall* int k; 0 <= k && k < L; Perm(c[k], write));
    context (\forall* int k; 0 <= k && k < L; Perm(d[k], write));
    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 tid >= 0 && tid < L;
/*@ context Perm(c[tid], write) **
          Perm(a[tid], read);
    ensures c[tid]==a[tid];
@*/
```

Iteration Contract 2 (IC_2) of loop L2:

```
/*@ context tid >= 0 && tid < L;
/*@ context Perm(c[tid], write) ** Perm(b[tid], read);
    ensures c[tid]==\old(c[tid]) +b[tid];
@*/
```

Iteration Contract 3 (IC_3) of loop L3:

```
/*@ context tid >= 0 && tid < L;
/*@ context Perm(d[tid], write) ** Perm(a[tid], read) ** Perm(b[tid], read);
    ensures d[tid]==a[tid]*b[tid];
@*/
```

Fig. 21 Required contracts for verification of the OpenMP example in Fig. 1

Program Contract (PC):

```
/*@ context_everywhere a != NULL && b != NULL && c != NULL && L>0;
    context_everywhere \length(a)==L && \length(b)==L && \length(c)==L;
    context (\forall* int k; 0 <= k && k < L; Perm(a[k], read));
    context (\forall* int k; 0 <= k && k < L; Perm(b[k], read));
    context (\forall* int k; 0 <= k && k < L; Perm(c[k], write));
    ensures (\forall  int k; 0 <= k && k < L; c[k]==a[k]*b[k]*b[k]);
@*/
```

Iteration Contract 1 (IC_1) of loop L1:

```
/*@ context tid >= 0 && tid < L;
/*@ context Perm(c[tid], write);
/*@ context Perm(a[tid], read);
/*@ context Perm(b[tid], read);
    ensures c[tid]==a[tid]*b[tid];
@*/
```

Iteration Contract 2 (IC_2) of loop L2:

```
/*@ context tid >= 0 && tid < L;
/*@ context Perm(c[tid], write) ** Perm(b[tid], read);
    ensures c[tid]==\old(c[tid]) *b[tid];
@*/
```

Fig. 22 Required contracts for verification of the vectorised loops in OpenMP example in Fig. 2

approach in the KeY verifier and show the effectiveness of their approach by experimenting on Java programs. Their approach for loop-free programs is highly automatic, but for programs containing loops, user interaction is required. In comparison with our work, for programs with loops, users need to provide loop invariants, while we only require iteration contracts (which we believe are often easier to specify).

Praun et al. [27] propose an abstract model to capture data dependences. The model represents these dependences as a density metric to predict potential concurrency of programs. This metric categorises the programs into high, medium and low densities. Programs with high density are good candidates for parallelism, while those with low density are not. Programs with medium density requires a scheduler that is aware of the algorithmic dependences. In contrast to our approach, their model abstracts from runtime aspects such as the number of threads and concurrency control and does not prove correctness of parallelised programs. Their work can benefit from our approach to guarantee correctness after discovering dependencies and parallelising the programs.

10 Conclusion and future work

We have presented the PPL language that captures the main forms of deterministic parallel programming, and we have shown how a commonly used subset of OpenMP can be encoded into PPL. Then, we proposed a verification tech-

nique to reason about data race freedom and functional correctness of PPL programs. The verification technique consists of two parts: reasoning about the correctness of basic blocks, and reasoning about the composition of blocks. Finally, we illustrate the technique to verify the correctness of an example OpenMP program.

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. Amighi, A., Haack, C., Huisman, M., Hurlin, C.: Permission-based separation logic for multithreaded Java programs. LMCS **11**(1), (2015)
2. Aviram, A., Ford, B.: Deterministic OpenMP for Race-free Parallelism. In HotPar'11 (2011)
3. Barthe, G., Crespo, J.M., Gulwani, S., Kunz, C., Marron, M.: From relational verification to SIMD loop synthesis. In: PPoPP, pp. 123–134 (2013)
4. Berger, M.J., Aftosmis, M.J., Marshall, D.D., Murman, S.M.: Performance of a new CFD flow solver using a hybrid programming paradigm. J. Parallel Distrib. Comput. **65**(4), 414–423 (2005)
5. Blom, S., Darabi, S., Huisman, M.: Verification of loop parallelisations. In: Egyed, A., Schaefer, I. (eds.) FASE, Volume 9033 of LNCS. Springer, pp. 202–217 (2015)
6. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: POPL, pp. 259–270 (2005)
7. Botincan, M., Dodds, M., Jagannathan, S.: Resource-sensitive synchronization inference by abduction. In: Field, J., Hicks, M. (eds.) Principles of Programming Languages (POPL 2012), pp. 309–322 (2012)
8. Botinčan, M., Dodds, M., Jagannathan, S.: Proof-directed parallelization synthesis by separation logic. ACM Trans. Program. Lang. Syst. **35**, 1–60 (2013)
9. Boyland, J.: Checking interference with fractional permissions. In: SAS, Volume 2694 of LNCS. Springer, pp. 55–72 (2003)
10. Bubel, R., Hähnle, R., Heydari Tabar, A.: A program logic for dependence analysis. In: Ahrendt, W., Tapia Tarifa, S.L. (eds.) Integrated Formal Methods. Springer International Publishing, Cham, pp. 83–100 (2019)
11. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.-H., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In Workload Characterization. IISWC 2009, pp. 44–54 (2009)
12. Darabi, S., Blom, S., Huisman, M.: A verification technique for deterministic parallel programs. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NASA Formal Methods (NFM), Volume 10227 of LNCS, pp. 247–264 (2017)
13. Dodds, M., Jagannathan, S., Parkinson, M.J.: Modular reasoning for deterministic parallelism. In ACM SIGPLAN Notices, pp. 259–270 (2011)
14. Dodds, M., Jagannathan, S., Parkinson, M.J., Svendsen, K., Birkedal, L.: Verifying custom synchronization constructs using higher-order separation logic. ACM Trans. Program. Lang. Syst. **38**(2), 4:1–4:72 (2016)
15. Haack, C., Huisman, M., Hurlin, C.: Reasoning about Java's reentrant locks. In: Ramalingam, G., (ed.) Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9–11, 2008. Proceedings, Volume 5356 of LNCS. Springer, pp. 171–187 (2008)
16. Hurlin, C.: Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic. PhD thesis, Université Nice Sophia Antipolis (2009)
17. Jin, H.-Q., Frumkin, M., Yan, J.: The OpenMP Implementation of NAS Parallel Benchmarks and its Performance (1999)
18. Leavens, G., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D.R., Müller, P., Kiniry, J., Chalin, P.: JML Reference Manual (2007). Dept. of Computer Science, Iowa State University. http://www.jmlspecs.org
19. Müller, P., Schwerhoff, M., Summers, A.: Viper—a verification infrastructure for permission-based reasoning. In VMCAI (2016)
20. OpenMP architecture review board, OpenMP API specification for parallel programming. Last accessed 18 Oct 2016. http://openmp.org/wp/
21. LLNL OpenMP Benchmarks. Last accessed 28 Nov 2016. https://asc.llnl.gov/CORAL-benchmarks/
22. Parkinson, M., Summers, A.: The relationship between separation logic and implicit dynamic frames. In Barthe, G. (ed.) Programming Languages and Systems—20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings, volume 6602 of LNCS. Springer, pp. 439–458 (2011)
23. Raychev, V., Vechev, M., Yahav, E.: Automatic synthesis of deterministic concurrency. In: Static Analysis—20th International Symposium, SAS 2013, Seattle, WA, USA, June 20–22, 2013. Proceedings. Springer, pp. 283–303 (2013)
24. Salamanca, J., Mattos, L., Araujo, G.: Loop-carried dependence verification in OpenMP. In: International Workshop on OpenMP 2014, pp. 87–102 (2014)
25. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames. ACM Trans. Program. Lang. Syst. **34**(1), 2:1–2:58 (2012)
26. Viper project website. http://www.pm.inf.ethz.ch/research/viper
27. von Praun, C., Bordawekar, R., Cascaval, C.: Modeling optimistic concurrency using quantitative dependence analysis. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 185–196 (2008)