# Verification of Loop Parallelisations

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

University of Twente, the Netherlands

**Abstract.** Writing correct parallel programs becomes more and more difficult as the complexity and heterogeneity of processors increase. This issue is addressed by parallelising compilers. Various compiler directives can be used to tell these compilers where to parallelise. This paper addresses the correctness of such compiler directives for loop parallelisation. Specifically, we propose a technique based on separation logic to verify whether a loop can be parallelised. Our approach requires each loop iteration to be specified with the locations that are read and written in this iteration. If the specifications are correct, they can be used to draw conclusions about loop (in)dependences. Moreover, they also reveal where synchronisation is needed in the parallelised program. The loop iteration specifications can be verified using permission-based separation logic and seamlessly integrate with functional behaviour specifications. We formally prove the correctness of our approach and we discuss automated tool support for our technique. Additionally, we also discuss how the loop iteration contracts can be compiled into specifications for the code coming out of the parallelising compiler.

## 1 Introduction

Parallelising compilers aim to detect loops that can be executed in parallel. However, this detection is not perfect. Therefore developers can typically also add compiler directives to declare that a loop is parallel. Any loop annotated with such a compiler directive is assumed to be parallel by the compiler.

This paper discusses how to verify that loops that are declared parallel by a developer can indeed safely be parallelised. This is achieved by adding specifications to the program that when verified guarantee that the program can be parallelised without changing its behaviour. Our specifications stem from permission-based separation logic [5,6], an extension of Hoare logic. This has the advantage that we can easily combine the specifications related to non-functional properties such as data race freedom with functional correctness properties.

Concretely, for each loop body we add an *iteration contract*, which specifies the iteration's resources, i.e., the variables read and written by one iteration of the loop. We prove that if the iteration contract can be proven correct without any further annotations, the iterations are independent and the loop is parallelisable. If a loop has dependences, we can add additional annotations that capture these dependences. These annotations specify how resources are transferred to another iteration of the loop. We then identify a class of annotation patterns

for which we can prove that the loop can be vectorised because they capture forward dependences. Finally, we also discuss how the verified iteration contract (including possibly functional property specifications) can be translated into a verifiable contract for the parallelised or vector program, written as a kernel.

Our approach is motivated by our work on the CARP project[1]. As part of this project the PENCIL language has been developed [1]. It is a high-level programming language designed to ease the programming of many-core processors such as GPUs. Its core is a subset of sequential C, imposing strong limitations on pointer-arithmetic. However, it should be noted that our approach also is applicable to other programming languages or libraries that have a similar parallel loop construct, such as OpenMP [7], parallel_for in C++ TBB [21] and Parallel.For in .NET TPL [13].

The main contributions of our paper are the following:

- a specification technique, using iteration contracts and dedicated transfer annotations that can capture loop dependences;
- a soundness proof that loops respecting specific patterns of iteration contracts can be either parallelised or vectorised; and
- compilation of iteration contracts to kernel contracts for the parallelised or vectorised program.

An earlier paper sketching the idea of iteration contracts to capture dependences appeared in PLACES 2014 [3]. However, the current paper additionally proves soundness of the approach, and defines specification compilation.

The remainder of this paper is organised as follows. After some background information, Section 3 explains how iteration contracts precisely capture dependences. Soundness of the approach is proven in Section 4. Then Section 5 discusses tool support for iteration contracts, and Section 6 discusses compilation of specifications. Finally, we conclude with related and future work.

## 2 Background

We first provide some background on data dependence and separation logic.

*Loop Dependences.* Given a loop, there exists a *loop-carried dependence* from statement $S_{src}$ to statement $S_{sink}$ in the loop body if there exist two iterations $i$ and $j$ of that loop, such that: (1) $i < j$, and (2) instance $i$ of $S_{src}$ and instance $j$ of $S_{sink}$ access the same memory location, and (3) at least one of these accesses is a write. The *distance* of a dependence is defined as the difference between $j$ and $i$. We distinguish between *forward* and *backward* loop-carried dependences. When $S_{src}$ syntactically appears before $S_{sink}$ (or if they are the same statement) there is a *forward loop-carried dependence*. When $S_{sink}$ syntactically appears before $S_{src}$ there is a *backward loop-carried dependence*.

*Example 1 (Loop-Carried Dependence).* The examples below show two different types of loop-carried dependence. In (a) the loop has a *forward loop-carried dependence*, where $L_1$ is the source and $L_2$ is the sink, as illustrated by considering

[1] See `http://www.carpproject.eu/`.

iteration 1 and 2 of the loop. In general, the $i^{\text{th}}$ element of the array $a$ is shared between iteration $i$ and $i-1$. In (b) the loop has a *backward loop-carried dependence*, because the sink of the dependence ($L_1$) appears before the source ($L_2$).

| | | | |
|---|---|---|---|
| | **for**(**int** i=0;i<N;i++){ | iteration = 1 | iteration = 2 |
| (a) | $L_1$: a[i] = b[i] + 1; | $L_1$: a[1] = b[1] + 1; | $L_1$: a[2] = b[2] + 1; |
| | $L_2$: **if**(i>0) c[i]=a[i−1]+2;} | $L_2$: c[1] = a[0] + 2; | $L_2$: c[2] = a[1] + 2; |
| | **for**(**int** i=0;i<N;i++){ | iteration = 1 | iteration = 2 |
| (b) | $L_1$: a[i] = b[i] + 1; | $L_1$: a[1] = b[1] + 1; | $L_1$: a[2] = b[2] + 1; |
| | $L_2$: **if**(i<N−1) c[i]=a[i+1]+2;} | $L_2$: c[1] = a[2] + 2; | $L_2$: c[2] = a[3] + 2; |

The distinction between forward and backward dependences is important. Independent parallel execution of a loop with dependences is in general unsafe, because it may change the result. However, for loops with forward dependences only, parallelisation is possible if appropriate synchronisation is inserted. This is called *vectorised execution*.

*Separation Logic.* Our approach to reason about loop (in)dependences uses permission-based separation logic to specify which variables are read and written by a loop iteration. Separation logic [17] is an extension of Hoare logic [11], originally proposed to reason about pointer programs. Separation logic is also suited to reason modularly about concurrent programs [15]: two threads working on disjoint parts of the heap do not interfere and thus can be verified in isolation.

The basis of our work is a separation logic for C [22], extended with fractional permissions [6,5] to denote the right to either read from or write to a location. Any fraction in the interval $(0, 1)$ denotes a *read permission*, while 1 denotes a *write permission*. Permissions can be split and combined, but soundness of the logic prevents the sum of the permissions for a location over all threads to exceed 1. This guarantees that if permission specifications can be verified, the program is free of data races. In earlier work, we have shown that this logic is suitable to reason about kernel programs [4].

We write $\mathbf{Perm}(e, \pi)$ to denote that a thread holds an access right $\pi$ to the location denoted by expression $e$. Permissions are combined using *separating conjunction* ($*$), which is the resource-sensitive extension of *conjunction*. For example $\mathbf{Perm}(x, 1/2) * \mathbf{Perm}(y, 1/2)$ indicates that a thread holds read permissions to access locations $x$ and $y$, and these permissions are disjoint. If a thread holds $\mathbf{Perm}(x, 1/2) * \mathbf{Perm}(x, 1/2)$, this can be merged into a write permission $\mathbf{Perm}(x, 1)$.

## 3 Dependence Specifications

The classical way to specify the behaviour of a loop is by means of an invariant that has to be preserved by every iteration of the loop. However, loop invariants offer no insight into possible parallel executions of the loop. Instead we consider every iteration of the loop in isolation. First, we introduce our way of specifying them. Then, we propose a way of verifying our annotations.

```
for(int i=0; i < N; i++) /*@
      requires Perm(a[i],1) ** Perm(b[i],1/2);
      ensures  Perm(a[i],1) ** Perm(b[i],1/2);
 @*/ { a[i]= 2 * b[i];       }
```

**Listing 1:** Iteration contract for an independent loop

### 3.1 Iteration Contracts

Each iteration is specified by its *iteration contract*, such that the precondition of the iteration contract specifies the resources that a particular iteration needs, and the postcondition specifies the resources that become available after the execution of the iteration. In other words, we treat each iteration as a specified block [10]. For convenience, we present our technique on non-nested for-loops with $K$ statements that are executed during $N$ iterations.[2] Each statement $S_k$ labelled by $L_k$ consists of an atomic instruction $I_k$, which is executed if a guard $g_k$ is true, i.e., we consider loops of the following form:

**for**(**int** j=0; j < $N$; j++){ *body*($j$) }

where *body*($j$) $\equiv$ $L_1$: **if**($g_1$) $I_1$; ... $L_K$: **if**($g_K$) $I_K$;

There are two extra restrictions. First, the iteration variable $j$ cannot be assigned anywhere in the loop body. Second, the guards must be expressions that are constant with respect to the execution of the loop body, i.e., they may not contain any variable that is assigned within the iteration.

Listing 1 shows an example of an *independent loop* with its iteration contract. This contract requires that at the start of iteration $i$, permission to write a[i] is available, as well as permissions to read b[i]. Further, the contract ensures that these permissions are returned at the end of iteration $i$. The iteration contract implicitly requires that the *separating conjunction of all iteration preconditions* holds before the first iteration of the loop, and that the *separating conjunction of all iteration postconditions* holds after the last iteration of the loop. For example, the contract in Listing 1 implicitly specifies that upon entering the loop, permission to write the first $N$ elements of a must be available, as well as permission to read the first $N$ elements of b.

To specify *dependent loops*, we need to specify what happens when the computations have to *synchronise* due to a dependence. During such a synchronisation, permissions should 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 two annotations: **send** and **recv**:

//@ $L_S$: **send** $\phi$ **to** $L_R$, $d$;
//@ $L_R$: **recv** $\psi$ **from** $L_S$, $d$;

A **send** specifies that at label $L_S$ the permissions and properties denoted by formula $\phi$ 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 **recv** specifies that permissions and properties as specified by formula $\psi$ are received.

[2] Our technique can be generalized to nested loops as well.

```
(a)  for(int i=0; i < N; i++) /*@
       requires Perm(a[i],1) ** Perm(b[i],1/2) **  Perm(c[i],1);
       ensures  Perm(b[i],1/2) ** Perm(a[i],1/2) ** Perm(c[i],1);
       ensures  i>0 ==> Perm(a[i−1],1/2);
       ensures  i==N−1 ==> Perm(a[i],1/2);  @*/
     {  a[i]=b[i]+1;
        //@  L1:if (i< N−1) send Perm(a[i],1/2) to L2,1;
        //@  L2:if (i>0)  recv Perm(a[i−1],1/2) from L1,1;
        if (i>0) c[i]=a[i−1]+2; }

(b)  for(int i=0; i < N; i++)  /*@
       requires Perm(a[i],1/2) ** Perm(b[i],1/2) ** Perm(c[i],1);
       requires i==0 ==> Perm(a[i],1/2);
       requires i < N−1 ==> Perm(a[i+1],1/2);
       ensures  Perm(a[i],1) ** Perm(b[i],1/2) ** Perm(c[i],1);   @*/
     {  //@ L1:if (i>0) recv Perm(a[i],1/2) from L2,1;
        a[i]=b[i]+1;
        if (i < N−1) c[i]=a[i+1]+2;
        //@ L2:if (i < N−1)  send Perm(a[i+1],1/2) to L1,1; }
```

**Listing 2:** Iteration contracts for loops with loop-carried dependences

The **send** and **recv** annotations can be used to specify loops with both forward and backward loop-carried dependences. Listing 2, shows specified instances of the code in Example 1.

We discuss the annotations for part (a) in some detail. Each iteration $i$ starts with a write permission on a[i] and c[i], and a read permission ($\frac{1}{2}$) on b[i]. The first statement is a write to a[i], which needs write permission. The value written is computed from b[i], for which a read permission is needed. The second statement reads a[i−1], which is not allowed unless read permission is available. This statement is not executed in the first iteration, because of the condition $i > 0$. For all subsequent iterations, permission must be transferred. Hence a **send** annotation is specified before the second assignment that transfers a read permission on a[i] to the next iteration (and in addition, keeps a read permission itself). The postcondition of the iteration contract reflects this: it ensures that the original permission on c[i] is released, as well as the read permission on a[i], which was not sent, and also the read permission on a[i−1], which was received. Finally, since the last iteration cannot transfer a read permission on a[i], the iteration contract's postcondition also specifies that the last iteration returns this non-transferred read permission on a[i].

The **send** annotations indicate an order in which the iterations have to be executed, and thus how the loop can be parallelised. Any execution that respects this order yields the same behaviour as the sequential execution of the loop. For the forward dependence example, this means that it can be vectorised, i.e. we add appropriate synchronisation to the parallel program to ensure permissions can be transferred as specified. However, for the backward dependence example, only sequential execution respects the ordering.

### 3.2 Verification of Iteration Contracts

To prove the correctness of an iteration contract, we propose appropriate program logic rules. As mentioned above, an iteration contract implicitly gives rise to a contract for the loop. The following rule says that if the iteration contract is correct for any execution of the loop body then this contract is true:

$$\frac{\{P(j)\}\ body(j)\ \{Q(j)\} \quad \forall j.j \in [0 \cdots N)}{\{\bigstar_{j=0}^{N-1}\ P(j)\}\ \textbf{for}(\textbf{int}\ \text{j=0;j<N;j++})\{\ body(j)\ \}\ \{\bigstar_{j=0}^{N-1}\ Q(j)\}}$$

Note that this rule for a loop with an iteration contract is a special case of the rule for parallel execution, which allows arbitrary blocks of code to be executed in parallel (see e.g. [15]).

The rules for the **send** and **recv** are similar in spirit to the rules that are typically used for permission transfer upon lock releasing and acquiring, see e.g. [9]. In particular, **send** is used to give up resources that the **recv** acquires. This is captured by the following two proof rules:

$$\frac{}{\{P\}\ \textbf{send}\ P\ \textbf{to}\ L, d\ \{\mathsf{true}\}} \qquad \frac{}{\{\mathsf{true}\}\ \textbf{recv}\ P\ \textbf{from}\ L, d\ \{P\}} \tag{1}$$

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

(i) if $S_r$ is the statement $\textbf{if}(g_r(j))\ \textbf{recv}\ \psi(j)\ \textbf{from}\ L_s,\ d$; then $S_s$ is the statement $\textbf{if}(g_s(j))\ \textbf{send}\ \ \phi(j)\ \textbf{to}\ L_r,\ d$;

(ii) if the **recv** is enabled in iteration $j$, then $d$ iterations earlier, the **send** should be enabled, i.e.,

$$\forall j \in [0, \cdots, N).g_r(j) \implies j \geq d \wedge g_s(j-d) \tag{2}$$

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

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

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

## 4 Soundness of the Approach

Next, we show that a correct iteration contract capturing a loop independence or a forward loop-carried dependence indeed implies that a loop can be parallelised or vectorised, while preserving the behaviour of the sequential loop.

To construct the proof, we first define the semantics of the three loop execution paradigms: sequential, vectorised, and parallel. We also define the instrumented semantics for a loop specified with an iteration contract. Next, to prove the soundness of our approach we show that the instrumented semantics of an independent loop is equivalent to the parallel execution of the loop, while the instrumented semantics of a loop with a forward dependence is an extension of the vectorised execution of the loop. Functional equivalence of two semantics is shown by transforming the computations in one semantics into the computations in the other semantics by swapping adjacent independent execution steps.

### 4.1 Semantics of Loop Executions

To keep our formalisation tractable, we split the loop semantics into two layers. The upper layer determines which sequences of atomic statements, called *computations*, a loop can have. The lower layer defines the effect of each atomic statement, and we do not discuss this further here, as this is standard.

As above, we develop our formalisation for non-nested loops with $K$ guarded statements. We instantiate the loop body for each iteration of the loop, so we have $(L_i^j\colon \textbf{if}(g_i^j)\ I_i^j;)$ as the instantiation of the $i^{th}$ statement in the $j^{th}$ iteration of the loop. We refer to this instance of statements as $S_i^j$. 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 $g_i^j$ holds, otherwise it behaves as a skip. If we execute iterations one by one in sequential order and we preserve the program order of the loop body, we will have a sequence of statement instances starting from $S_0^0$ and ending at $S_K^{N-1}$. Intuitively this is the semantics of the sequential execution of the loop.

**Definition 1.** *A* computation *$c$ is a finite sequence $t_1, t_2, ..., t_m$ of statement instances such that $t_1$ is executed first, then $t_2$ is executed and so on until the last statement $t_m$.*

To define the set of computations describing the parallel and vectorised semantics of a loop, we define auxiliary operators *concatenation* and *interleaving*. We define two versions of concatenation, plain concatenation ($++$) and *synchronised concatenation* ($\#$), which prevents data races between statements by inserting a barrier $b$ that acts as a memory fence:

$$\begin{aligned} C_1 \mathbin{++} C_2 &= \{c_1 \cdot c_2 \mid c_1 \in C_1, c_2 \in C_2\} \\ C_1 \mathbin{\#} C_2 &= \{c_1 \cdot b \cdot c_2 \mid c_1 \in C_1, c_2 \in C_2\} \end{aligned}$$

We lift concatenation to multiple sets as follows:

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

Next, interleaving defines how to weave several computations into a single computation. This is parametrised by a *happens-before* order $<$, in order not to violate restrictions imposed by the program semantics. To define the interleaving operator ($\mathsf{Interleave}_<$), we use an auxiliary operator that denotes interleaving with a fixed first step: ($\mathsf{Interleave}^i_<$):

$$\begin{aligned} &\mathsf{Interleave}_<^{i=1..N} c_i = \mathsf{Interleave}_<(c_1, \cdots, c_n) = \textstyle\bigcup_{i=1}^{n} \mathsf{Interleave}^i_<(c_1, \cdots, c_n) \\ &\mathsf{Interleave}^i_<(\epsilon, \cdots, \epsilon) = \{\epsilon\} \\ &\mathsf{Interleave}^i_<(c_1, \cdots \epsilon \cdots, c_n) = \emptyset \\ &\mathsf{Interleave}^i_<(c_1, \cdots s_i\, c_i \cdots, c_n) = \emptyset \quad \text{, if } \exists j \neq i, s \in c_j.s < s_i \\ &\mathsf{Interleave}^i_<(c_1, \cdots s_i\, c_i \cdots, c_n) = \{s_i \cdot s \mid s \in \mathsf{Interleave}_<(c_1, \cdots c_i \cdots, c_n)\} \text{ , otherwise} \end{aligned}$$

where $\epsilon$ is the empty computation. We use two happens-before orders: *program order* ($\mathsf{PO}$), which maintains the order of statements executed by the same thread

and *specification order* ($\mathsf{SO}$), which extends program order by also enforcing that for every matching pair of **send** and **recv**, the **send** statement happens-before the **recv** statement. Both orders maintain the order between a barrier and the statements preceding and following it.

Now we are ready to define the semantics of the different loop executions. *Sequential execution* simply executes all steps sequentially, *parallel execution* allows any interleaving that preserves program order within the loop body and *vectorised execution* executes multiple iterations in lock-step.

**Definition 2.** *Suppose we have a loop LP in standard form.*

- *Its* sequential execution semantics *is* $[\![LP]\!]^{Seq} = \mathsf{Concat}_{j=0}^{N-1}\mathsf{Concat}_{i=1}^{K}[\![S_i^j]\!]$
- *Its* parallel execution semantics *is* $[\![LP]\!]^{Par} = \mathsf{Interleave}_{\mathsf{PO}}^{j=0..N-1}\mathsf{Concat}_{i=1}^{K}[\![S_i^j]\!]$
- *Its* vectorised execution semantics *for vector length* $V$ *is*

$$[\![LP]\!]^{Vec(V)} = \mathsf{Concat}_{v=0}^{(N/V)-1}\mathsf{SyncConcat}_{i=1}^{K}\left(\mathsf{Interleave}_{\emptyset}^{j=vV..vV+V-1}[\![S_i^j]\!]\right)$$

We define the *instrumented semantics* to capture the behaviour of $LP$ in the presence of its specifications. This semantics contains all possible computations respecting the specification order ($\mathsf{SO}$). It is formalised by parametrising the interleaving operator with $\mathsf{SO}$.

**Definition 3.** *The instrumented semantics of* LP *is*

$$[\![LP]\!]^{Spec} = \mathsf{Interleave}_{\mathsf{SO}}^{j=0..N-1}\mathsf{Concat}_{i=1}^{K}[\![S_i^j]\!]$$

### 4.2 Correctness of Parallel Loops

In the previous section, we defined the semantics of parallelised and vectorised executions in terms of possible traces of atomic steps. This section proves, under certain conditions, that each of those traces is safe and yields the same result as sequential execution, where safe means that the execution of the trace is data race free. Equivalence is established by considering traces modulo reordering independent steps and while ignoring steps that make no modifications. First, we formally define these notions. Then we present our correctness theorems.

To determine if two steps are independent and/or can cause a data race, we need to know for every atomic step ($t$) which locations in memory it writes ($\mathsf{write}(t)$), which locations in memory it reads ($\mathsf{read}(t)$) and by which thread it is executed. We define the set of accessed locations as $\mathsf{access}(t) = \mathsf{write}(t) \cup \mathsf{read}(t)$. Now we define a *data race* in a trace as a pair of statements that both access a location, where at least one access is a write, and are not ordered by the happens-before relation:

**Definition 4.** *A computation contains a* data race *with respect to an happens-before order* $<$*, if it contains two steps* $s$ *and* $t$*, such that* $\mathsf{write}(s) \cap \mathsf{access}(t) \neq \emptyset \wedge \neg(s < t \vee t > s)$.

To reason about different execution orders, equivalence of executions is defined in terms of swapping the order of steps which are not in the happens-before relation. The following proposition states that this does not change the end result of a data race free computation.

**Proposition 1.** *In a data race free computation, swapping two adjacent statements which are unordered in the happens-before relation does not change the behaviour of that computation.*

*Proof.* Because the statements are unordered and the computation is data race free, the set of locations written by one of the actions cannot affect the set of locations accessed by the other. Hence neither step can see the effect of the other. □

The traces in the different semantics do not just differ by their order, but also by steps that are used to enforce synchronisation. To compare the functional result of two threads, we only look at the steps in those traces that actually modify locations that are relevant to the program semantics.

**Definition 5.** *Given two computations $c_1$ and $c_2$. The computations $c_1$ and $c_2$ are* functionally equivalent *if $\mathsf{mods}(c_1) = \mathsf{mods}(c_2)$, where*

$$\mathsf{mods}(c) = \begin{cases} \epsilon & , \text{ if } c = \epsilon \\ \mathsf{mods}(c') & , \text{ if } c = t \cdot c' \wedge \mathsf{write}(t) = \emptyset \\ t \cdot \mathsf{mods}(c'), & \text{if } c = t \cdot c' \wedge \mathsf{write}(t) \neq \emptyset \end{cases}$$

The correctness of the various loop semantics depends on the correctness of the instrumented semantics:

**Theorem 1.** *Given a loop with a valid specification.*

1. *All computations in $[\![LP]\!]^{Spec}$ are data race free.*
2. *All computations in $[\![LP]\!]^{Spec}$ and $[\![LP]\!]^{Seq}$ are functionally equivalent.*

*Proof.* 1. Because there is a valid specification, all invariants of separation logic hold. In particular, for every location the sum over all threads of the permissions held for that location cannot exceed 1.
Suppose that a statement $s$ occurs before $t$ where one writes a location $l$ and the other accesses it and they are not ordered by happens-before ($s \not< t$). If $s$ needs a fraction $p$ permission on $l$ then we can trace which threads hold the permission when $t$ is executed. It cannot be the thread that executes $t$, because that implies $s < t$. The fraction $q$ held for $t$ and $p$ are thus held at the same time. Because $p = 1$ or $q = 1$, we have $p + q > 1$. This contradicts the invariant.

2. We prove that every computation in $[\![LP]\!]^{Spec}$ is functionally equivalent to the single computation $[\![LP]\!]^{Seq}$, by showing that any computation can be reordered until it is the sequential computation using Prop. 1.
Assume that the first $n$ steps of the given computation are in the same order

as the sequential computation. 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 computation is in happens-before order, all of the steps that have to happen before $t_{n+1}$ are already included in the prefix. Hence, step $t_{n+1}$ is independent of all of the steps after the prefix and before itself in the given sequence and can be swapped with them one-by-one until it is the next step. We then repeat until the whole sequence matches. □

The correctness of parallelisation of independent loops is an immediate corollary of this theorem.

**Corollary 1.** *Given a loop with a valid specification, that does not make use of* **send** *or* **recv**.

1. *All computations in* $[\![LP]\!]^{Par}$ *are data race free.*
2. *All computations in* $[\![LP]\!]^{Par}$ *and* $[\![LP]\!]^{Seq}$ *are functionally equivalent.*

*Proof.* If the specification does not make use of **send** or **recv** then program order coincides with specification order and the result follows from Theorem 1. □

This proof is straightforward because in this case, the program order and synchronisation order coincide, thus the set of parallel executions is equivalent to the set of instrumented executions. However, if the specifications use **send** and **recv** then some parallel execution order may contain data races. But if the **send** occurs before the matching **recv** in the loop then vectorisation is possible.

**Theorem 2.** *Given a loop with a valid specification, such that every* **send** *occurs before the matching* **recv** *in the body, and* $V$ *that divides* $N$.

1. *All computations in* $[\![LP]\!]^{Vec(V)}$ *are data race free.*
2. *All computations in* $[\![LP]\!]^{Vec(V)}$ *and* $[\![LP]\!]^{Seq}$ *are functionally equivalent.*

*Proof.* Because every **send** occurs before the matching **recv**, every computation that may occur in $[\![LP]\!]^{Vec(V)}$ can also occur in $[\![LP]\!]^{Spec}$. That is, we can construct a specification order sequence in which the computational steps occur in the same order and in which the happens-before relation on the vectorised sequence are more restrictive than those in the specification order sequence. Hence all vectorised sequences are data race free because all specification order sequences are data race free (Theorem 1). Moreover, every vectorised computation is functionally equivalent to a specification order sequence and thus functionally equivalent to $[\![LP]\!]^{Seq}$ (Theorem 1). □

## 5 Tool Support

After discussing the soundness of our approach, we now turn to tool support as provided by the VerCors tool set. The VerCors tool set was originally developed to reason about multi-threaded Java programs, but it has been extended to support verification of OpenCL kernels [4] and parallel loops. The tool set leverages

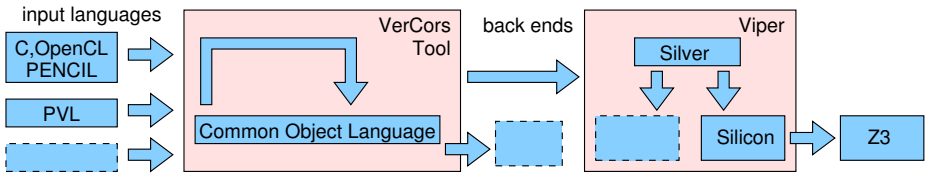


**Fig. 1.** VerCors tool set overall architecture

existing verification technology: it encodes programs via several program transformation steps into Silver programs [12]. Silver is an intermediate language for separation logic-like specifications, used by the Viper project [12,23]. Verification of the encoded program uses the Silver verification framework. Figure 1 sketches the overall architecture of the tool set, where dashed boxes are other front-ends and back-ends that are not relevant for this paper.

**Encoding into Silver.** For the verification of parallel loops, we only use the encoding into Silver, using the Silicon verifier. We describe this encoding below.

To verify our iteration contracts using the Silicon verifier, we encode the behaviour of parallel loops and the **send**/**recv** annotations as method contracts. The idea is that every loop annotated with an iteration contract is encoded by a call to the method loop_main, whose contract encodes the application of the Hoare Logic rule for parallel loops, 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)); @*/
loop_main(**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 loop variable, containing the loop body, and specified by the iteration contract.

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

Within the body there may be **send** and **recv** statements.

//@ $L_s$: **if** ($g_s(j)$) { **send** $\phi(j)$ **to** $L_r$, $d$;}
//@ $L_r$: **if** ($g_r(j)$) { **recv** $\psi(j)$ **from** $L_s$, $d$;}

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

//@ $L_s$: **if** ($g_s(j)$) { send_s_to_r(j,N,free($\phi(j)$);}
//@ $L_r$: **if** ($g_r(j)$) { recv_s_to_r(j,N,free($\psi(j)$);}

where **requires** $\phi(j)$; **ensures** $\psi(j)$;
send_s_to_r(**int** j,**int** N,free($S$))); recv_s_to_r(**int** j,**int** N,free($S$)));

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

**Verification Examples.** In Section 3, we illustrated our approach by specifying loops with different data dependencies in Listings 1, and 2. These examples are

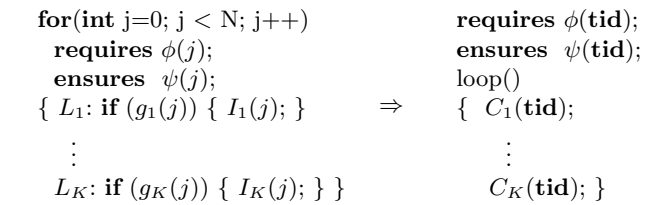


**Fig. 2.** Vectorisation of a loop with a forward loop-carried dependence

verified automatically by the tool. Moreover, the tool is also able to verify the functional correctness of loops. For example, to verify the functional correctness of the program in Listing 2(a), we could add the following specifications:

```
requires  b[i]==i;
ensures   a[i]==i+1 ** b[i]==i ** (i>0 ==> c[i]==i+2);
```

to its iteration contract. To make this verify, the property a[i]==i+1 has to be added to the **send** resource formula and a[i−1]==i has to be added to the **recv** resource formula. To see the fully annotated examples, we refer to `http://www.utwente.nl/vercors`.

## 6 Compiling Iteration Contracts to Kernel Specifications

Above, we discussed verification of loop parallelisability in high-level sequential programs. Typically, we want to be sure that when we parallelise the program, the resulting low-level parallel code is still correct. To support this, we define how a specification of the original program can be translated into a specification of the low-level code. In particular, this section shows how iteration contracts are translated into OpenCL kernel specifications [4], such that if the code is compiled using a basic parallelising compiler, without further optimisations, the compiled code is correct w.r.t. the compiled specification.

*Independent loops* Given an independent loop, the basic compilation to kernel code is simple: create a kernel with as many threads as there are loop iterations and each kernel thread executes one iteration. Moreover, the iteration contract can be used as the thread contract for each parallel thread in the kernel directly. The size of the work-group can be chosen at will, because no barriers are used.

*Forward loop-carried dependences* If the loop has forward dependences then the kernel must mimic the vectorised execution of the loop. Consider the specified loop on the left of Figure 2, for simplicity, we assume that both the number of threads and the size of the working group are $N$. Basic vectorisation results in the kernel on the right of Figure 2, where:

– if $I_k(j)$ is a **send** statement then it is ignored: $C_k(j) \equiv \{\}$

```
kernel Ref {
  global int[tcount] a,b,c;

  requires Perm(a[tid],1) ** Perm(b[tid],1/2) ** Perm(c[tid],1) ** b[tid]==tid;
  ensures  Perm(a[tid],1/2) ** Perm(b[tid],1/2) ** Perm(c[tid],1);
  ensures  tid>0 ==> Perm(a[tid−1],1/2);
  ensures  tid==tcount−1==>Perm(a[tid],1/2);
  ensures  a[tid]==tid+1 ** b[tid]==tid ** (tid>0==>c[tid]==tid+2);

  void main(){
    a[tid]=b[tid]+1;
    barrier(global){
      requires tid<tcount−1 ==> Perm(a[tid],1/2) ** a[tid]==tid+1;
      ensures  (tid>0==>Perm(a[tid−1],1/2)) ** (tid>0==>a[tid−1]==tid);}
    if (tid>0)   c[tid]=a[tid−1]+2;
  }
}
```

**Listing 3:** Kernel implementing the loop with forward dependence

- if $I_k(j)$ is a **recv** statement with a matching **send** statement at $L_i$, then it is replaced by a barrier $C_k(j) \equiv$

  $$\textbf{barrier}()\{\textbf{requires } g_i(j) \texttt{==>} \phi_S(j); \textbf{ ensures } \ g_k(j) \texttt{==>} \phi_R(j);\}$$

  where the barrier contract specifies how the permissions are exchanged at the barrier (cf. [4]).
- if $I_k(j)$ is any other statement then it is copied: $C_k(j) \equiv \textbf{if } (g_k(j))\{\ I_k(j);\ \}$

Listing 3 shows the kernel that is derived in this way from the forward dependence example in Listing 2(a).

## 7 Related Work

**Verification of high-level parallel constructs.** Recently, almost all major programming languages have been augmented by high-level parallelisation constructs. Verification of these high-level constructs has been investigated in different works. Salamanca et al. [19] present an integration of a runtime loop-carried dependence checker in OpenMP. Compared to their approach, we propose a static approach to detect loop-carried dependences, that is valid for all possible executions.

Radoi et al. [16] employ the restricted thread structure of parallel loops to specialise a set of static data race detection techniques and make them practical for the verification of Java 8 loop-parallelism mechanism [20]. In comparison, their method cannot distinguish between vectorised and parallel loop executions, while our approach propose different specification patterns for each of these executions. Also, they use a specialised data race techniques for Java 8 collections, while we investigate the problem in a more general sense.

Barthe et al. [2] propose a new program synthesis technique which produces SIMD code for a given innermost loop. They exploit the *relational verification*

approach to prove functional equivalence of the generated SIMD code and the original sequential code, while we employ permission-based separation logic to prove such an equivalence for both vectorised and parallel loop executions.

**Automated loop verification.** Gedell et al. [8] employ automated first-order reasoning in order to deal with parallelisable loops instead of interactive proof techniques, such as induction. They transform a loop into a universally quantified update of state changes by the loop body. The extraction of quantified state update for a particular loop iteration is intuitively similar to the idea of iteration contracts in our method. Their technique only works for parallelisable loops where there is no loop-carried dependence, while our iteration contracts idea addresses both dependent and independent loops.

**Parallelising compilers.** From parallelising compilers perspective, our approach can complement the current static dependence analysis techniques. Specifically, in case of *input-dependent semantics* where static analysis cannot decide whether a loop is independent or not [14,18].

## 8 Conclusion and Future Work

This paper proposes how to verify compiler directives about loop parallelisation. Each loop is specified by its iteration contract and in the presence of loop-carried dependence, additional **send**/**recv** annotations are added to the iteration specifications to indicate how the iterations synchronise with each other. We prove that loops without **send**/**recv** annotations are parallelisable, and for a specific pattern of **send**/**recv** annotations the loop is vectorisable. As an additional result, we propose how the high-level iteration contracts can be compiled into low-level kernel contracts.

In addition to the verification of compiler directives, our approach can be employed to detect possible loop parallelisations even where (in)dependence cannot be determined from static analysis of program text.

The method described is modular in the sense that it allows us to treat any parallel loop as a statement, thus nested loops can be dealt with simply by giving them their own iteration contract. Alternatively one iteration contract can be used for several nested loops.

As future work we plan to investigate how the verifier and the parallelising compiler can support each other. We believe this support can work in both ways. First of all, the parallelising compiler can use verified annotations to know about dependences without analysing the code itself. Conversely, if the compiler performs an analysis then it could emit its findings as a specification template for the code, from which a complete specification can be constructed. This might extend to a set of techniques for automatic generation of iteration contracts.

**Acknowledgement** This work is supported by the ERC 258405 VerCors project and by the EU FP7 STREP 287767 project CARP.

## References


1. R. Baghdadi, A. Cohen, S. Guelton, S. Verdoolaege, J. Inoue, T. Grosser, G. Kouveli, A. Kravets, A. Lokhmotov, C. Nugteren, F. Waters, and A. F. Donaldson. PENCIL: Towards a Platform-Neutral Compute Intermediate Language for DSLs. *CoRR*, abs/1302.5586, 2013.
2. G. Barthe, J. M. Crespo, S. Gulwani, C. Kunz, and M. Marron. From relational verification to SIMD loop synthesis. In *PPoPP*, pages 123–134, 2013.
3. S. Blom, S. Darabi, and M. Huisman. Verifying parallel loops with separation logic. In *PLACES*, pages 47–53, 2014.
4. S. Blom, M. Huisman, and M. Mihelčić. Specification and verification of GPGPU programs. *Science of Computer Programming*, 2014.
5. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In J. Palsberg and M. Abadi, editors, *POPL*, pages 259–270. ACM, 2005.
6. J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis Symposium*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.
7. L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.
8. T. Gedell and R. Hähnle. Automating verification of loops by parallelization. In *LPAR*, pages 332–346, 2006.
9. C. Haack, M. Huisman, and C. Hurlin. Reasoning about Java's reentrant locks. In G. Ramalingam, editor, *Asian Programming Languages and Systems Symposium*, volume 5356 of *LNCS*, pages 171–187. Springer, 2008.
10. E. Hehner. Specified blocks. In B. Meyer and J. Woodcock, editors, *VSTTE*, volume 4171 of *Lecture Notes in Computer Science*, pages 384–391. Springer, 2005.
11. C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
12. U. Juhasz, I. T. Kassios, P. Müller, M. Novacek, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. Technical report, ETH Zurich, 2014.
13. Microsoft TPL. http://msdn.microsoft.com/enus/library/dd460717.aspx.
14. C. E. Oancea and L. Rauchwerger. Logical inference techniques for loop parallelization. *SIGPLAN Not.*, 47(6):509–520, 2012.
15. P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.
16. C. Radoi and D. Dig. Practical static race detection for java parallel loops. In *ISSTA*, pages 178–190, 2013.
17. J. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.
18. S. Rus, M. Pennings, and L. Rauchwerger. Sensitivity analysis for automatic parallelization on multi-cores. In *Proceedings of the 21st Annual International Conference on Supercomputing*, ICS, pages 263–273. ACM, 2007.
19. J. Salamanca, L. Mattos, and G. Araujo. Loop-carried dependence verification in openmp. In *IWOMP*, pages 87–102, 2014.
20. State of the Lambda: Libraries Edition. `http://cr.openjdk.java.net/~briangoetz/lambda/lambda-libraries-final.html`.
21. Threading Building Blocks. `http://threadingbuildingblocks.org`.
22. H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *POPL*, pages 97–108. ACM, 2007.
23. Viper project website. `http://www.pm.inf.ethz.ch/research/viper`.