# Verifying Pipeline Implementations in OpenMP

Maik Wiesner(✉) and Marie-Christine Jakobs(✉)

Department of Computer Science, Technical University of Darmstadt, Darmstadt, Germany
wiesner@svps.tu-darmstadt.de, jakobs@cs.tu-darmstadt.de

**Abstract.** OpenMP is a popular API for the development of parallel, shared memory programs and allows programmers to easily ready their programs to utilize modern multi-core processors. However, OpenMP-compliant programs do not guarantee that the OpenMP parallelization is functionally equivalent to a sequential execution of the program. Therefore, several approaches analyze OpenMP programs. While some approaches check functional equivalence, they are either general purpose approaches, which ignore the structure of the program and the design pattern applied for parallelization, or they focus on parallelized for-loops. In this paper, we propose a verification approach that aims at pipeline parallelism. To show functional equivalence, our approach mainly computes the dependencies that a sequential execution imposes on the pipeline stages and checks whether these dependencies are incorporated in the OpenMP parallelzation. We implemented our verification approach in a prototype tool and evaluated it on some examples. Our evaluation shows that our approach soundly detects incorrect pipeline implementations.

**Keywords:** OpenMP verification · Functional equivalence · Pipeline parallelism · Parallel design pattern

## 1 Introduction

For several years, the CPU frequency has stayed the same, while the number of cores per CPU is increasing. To take full advantage of today's hardware, we need multi-threaded programs. However, many programs are still not multi-threaded.

OpenMP [17] is an API that allows one to easily transform sequential programs into multi-threaded ones, which are even platform independent. To parallelize a sequential program, one often only needs to insert OpenMP directives.

One problem of OpenMP parallelization is that not all OpenMP-compliant programs are correct [17]. For example, an OpenMP-compliant program may contain data races or deadlocks. Even worse, correctly applying OpenMP is

This work was funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project.

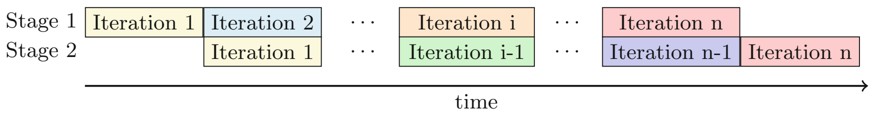


**Fig. 1.** Pipeline parallelism with a two-stage pipeline

```
void foo(int *a, int *b, int N)
{
  #pragma omp parallel
  {
    #pragma omp single
    for(int i=1; i<N;i++)
    {
      #pragma omp task depend(in: a[i-1]) depend(out: b[0:i+1])
      b[i] = a[i-1];
      #pragma omp task depend(in: b[i]) depend(out: a[0:i+1])
      a[i] = b[i]-a[i];
    }
  }
}
```

**Listing 1.1.** An example for a parallelization that uses the pipeline pattern

difficult [5]. Therefore, we need verification methods to check whether a program parallelized with OpenMP is correct, i.e., we want to show that every execution of the parallelized program is *functionally equivalent* to a sequential execution that ignores the OpenMP directives.

Several approaches [2–4,13,14,19,20,23,28] look into correctness threats of OpenMP parallelizations, e.g., data races [2–4,14,23], deadlocks [14,20], etc. However, these approaches do not guarantee functional equivalence. In contrast, equivalence checkers like Pathg [27], CIVL [22], PEQcheck [7], or the approach proposed by Abadi et al. [1] aim at proving functional equivalence. These equivalence checkers are general purpose checkers that ignore how a program is parallelized and, thus, regularly fail to show equivalence.

To overcome this problem, PatEC [6], AutoPar's correctness checker [12] and CIVL's OpenMP simplifier [22] take the kind of parallelization into account. However, they only support parallelizations utilizing data parallelism, e.g., parallelizations of loops whose iterations are independent of each other. In this paper, we propose a verification approach that aims at *pipeline parallelism*.

The pipeline pattern is a parallel design pattern [15,16] that may be used for loops whose iterations depend on each other. The idea of the pipeline pattern is similar to a processor pipeline. The sequential loop iteration is split into a sequence of stages such that each stage consumes data from the previous stage and provides data to the next stage. To exploit parallelism, the execution of consecutive loop iterations are overlapped as shown in Fig. 1.

Our verification approach assumes that the pipeline is implemented as follows. Stages are encapsulated in OpenMP tasks and depend clauses or synchronization directives describe the dependencies between stages. The loop itself is enclosed in a combination of a parallel and single OpenMP directive. Listing 1.1

shows an example for a pipeline with two stages. For demonstration purposes, we assume that a and b do not overlap. Hence, we do not require an in-dependency for a[i] in line 10. Read entry a[i] can only be modified in the same task.

Given a pipeline implementation of that form, our approach first determines the dependencies between tasks. In our example, the first task of iteration $i$ depends on the second task of iteration $i-1$, which declares a dependency on the first $i-1$ elements of array a. Moreover, the second task of iteration $i$ depends on the first task of iteration $i$. In addition, the first (second) task in iteration $i$ depend on all first tasks (second) tasks generated in iterations $j < i$. Next, our approach looks at all read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW) dependencies in the sequential execution that cross stage boundaries. In our example, we have two RAW dependencies.[1] For each of these dependencies, our approach checks whether this dependency is captured by the task dependencies. For write-after-read dependencies, our approach also inspects whether the parallelized program uses data-sharing attributes that allow the read to access the value of the write. In our example, the RAW and WAR dependency is captured by the task dependencies and variable a and b are shared, i.e., writes to a and b are visible to all threads. Finally, our approach examines whether all variables modified by and live at the end of the pipeline use data-sharing attribute shared, which makes the modification visible.

As a proof-of-concept, we implemented our verification procedure in a prototype and evaluated it on several example programs. Our evaluation shows that our implementation soundly detects all incorrect pipeline implementations.

## 2 Using OpenMP to Implement Pipelines

OpenMP [17] is a standard that programmers can use to implement parallel programs for shared-memory systems. Programmers typically only insert OpenMP directives into the code, which the compiler considers to generate the parallel program. In the following, we describe the important features needed to implement a parallel pipeline in OpenMP. We start with the OpenMP directives.

*OpenMP Directives in Parallel Pipelines.* We assume that the parallel pipelines are realized with the following five directives:

**parallel.** Defines a parallel region that multiple threads execute in parallel.
**single.** Defines a region, typically nested inside a parallel region, that is executed by exactly one thread.
**task.** Defines a code region that is executed by an arbitrary thread in parallel with e.g., other tasks.
**barrier.** Introduces an explicit barrier that must be reached by all threads of the enclosing parallel region before any thread can continue.

[1] In iteration $i$, the second task reads memory location b[i] after it is written by the first task. Similarly, the first task reads memory location a[i-1] in iteration $i$ after the second task writes to it in iteration $i-1$.

**taskwait.** Forces the task, `single` region, etc. to wait on the completion of its generated child tasks.[2]

*Data-Sharing Attributes.* The data-sharing attribute defines the visibility of a variable, e.g., whether it is shared among the threads or each thread uses its own local copy, and how information is exchanged between the original variable and its copies. Our analysis supports the following attributes, which primarily occur in parallel pipelines.

**private.** Each thread has a local copy of the variable, which is uninitialized.
**firstprivate.** Similar to private, each thread gets its own local copy. In addition, the variable is initialized to the value of the original variable at the point the directive (`parallel`, `single`, `task`) is encountered.
**shared.** The variable is shared among all threads.

Data-sharing attributes can be specified explicitly by adding data-sharing clauses to the above directives. If not specified explicitly, the data-sharing attribute is determined implicitly via rules. For example, variables declared inside a region are typically private. Variables declared outside the parallel pipeline are shared and for tasks all other variables are normally firstprivate.

*Depend Clauses.* By default, tasks are executed independently, e.g., concurrently or in arbitrary order. However, pipeline parallelism requires a certain partial order on tasks. Depend clauses, which can be added to the task directive, allow one to enforce such order constraints[3]. The general structure of these clauses is

$$\textbf{depend}(\textit{type} : \textit{varList}),$$

where $\textit{type} \in \{\texttt{in}, \texttt{out}, \texttt{inout}\}$ and *varList* denotes a list of variables and array sections as shown in Listing 1.1[4] The dependence type specifies how the mentioned variables are accessed, i.e., read (`in`), written (`out`), or both (`inout`).

Semantically, depend clauses specify a dependency between tasks. A task cannot execute before all tasks that it depends on and that are generated before it finished. The following definition formalizes the dependency between tasks.

**Definition 1.** *Tasks $T_1$ and $T_2$* are dependent *if there exists a depend clause $\texttt{depend}(t_1{:}l_1)$ for $T_1$ and a depend clause $\texttt{depend}(t_2{:}l_2)$ for $T_2$ such that*

- *$(t_1, t_2) \neq (in, in)$ and*
- *there exist $a_1 \in l_1$ and $a_2 \in l_2$ such that $a_1$ and $a_2$ designate the same memory location.*

[2] Note that we currently do not support `taskwait` directives with depend clauses.
[3] In general, the constraints apply to sibling tasks only. Due to the construction of tasks in parallel pipeline implementations that we support, all tasks are siblings.
[4] The OpenMP standard also allows other variants of the depend clause but we stick to these because they are the main ones used when realizing the pipeline pattern. Ignoring other types is sound but can lead to false positives.

In our verification approach, we only look at task constructs (task directive plus associated code region), not tasks generated for task constructs. Thus, we may only introduce dependencies between task constructs if a dependency always exists for all respective tasks generated by the pipeline. This property is fulfilled for all dependencies built on scalar variables or array subscripts that are loop invariant, i.e., whose value does not change during pipeline execution. Also, task construct $T_1$ depends on task construct $T_2$ if every task of $T_2$ depends on every previously constructed task of $T_2$ and each task generated for $T_1$ depends on the last task generated for $T_2$. The following definition captures the first two cases and demonstrates the latter for pipelines with incrementing loops, i.e., loops that only change the loop counter at the end of each loop iteration and that change increments the loop counter by one. Supporting further loops, e.g., decrementing loop or loops with another step size, or supporting multi-dimensional arrays is rather straightforward.

**Definition 2.** *Task construct $T_1$ depends on task construct $T_2$ if there exists a depend clause* ***depend***$(t_1{:}l_1)$ *for $T_1$ and a depend clause* ***depend***$(t_2{:}l_2)$ *for $T_2$ such that*

– $(t_1, t_2) \neq (in, in)$ *and*
– *there exist $a_1 \in l_1$ and $a_2 \in l_2$ such that either*
  1. *$a_1 = a_2$ and $a_1, a_2$ are scalar variables,*
  2. *$a_1 = a_2$, and $a_1, a_2$ are array subscripts, and the subscript expressions are loop invariant, or*
  3. *the pipeline uses an incrementing loop with loop counter $i$ and there exists an array $a$ such that either*
     (a) *$a_2 = a[0 : i + 1], a_1 = a[i]$ and $T_2$ occurs after $T_1$ in the loop body,*
     (b) *$a_2 = a[0 : i + 2], a_1 = a[i]$ and $T_2$ occurs after $T_1$ in the loop body,*
     (c) *$a_2 = a[0 : i+1], a_1 = a[i-1]$ and $T_2$ occurs before $T_1$ in the loop body.*

*Pipeline Structure.* Pipelines can be realized in different ways in OpenMP. We assume that the pipeline is structured as shown in Listing 1.2. This is a common structure for a pipeline implementation and it is e.g., used by the auto-parallelization tool DiscoPoP [10]. As shown in Listing 1.2, the pipeline is implemented in a `parallel` region. Inside the `parallel` region, a `single` region

```
#pragma omp parallel
{ .
  ⋮ //Declarations
#pragma omp single
    {     .
          ⋮ //Declarations
        for /*or while*/ (...) {

            //Tasks, statements, barriers
        }
    }
}
```

**Listing 1.2.** General structure of a pipeline implementation

**Algorithm 1:** Verification algorithm

```
Input: program - source code of program with pipeline to verify
1 dependGraph := buildDependencyGraph(program)
2 potentialViolations := checkRWDependencies(dependGraph)
3 violation := checkPotentialViolations(potentialViolations)
4 if violation = ⊥ then
5     violation := checkRemainingDependencies(dependGraph)
6 return witness
```

constructs the pipeline stages in a loop[5]. Thereby, each instance of a pipeline stage becomes a task, which must not include tasks itself, i.e. task constructs must not be nested. Furthermore, we allow declarations of temporary variables at the beginning of the `parallel` or `single` region. In addition to task constructs, the loop body may contain statements, which prepare the different stages, and barriers (`barrier` or `taskwait`) to further order tasks. Task constructs, statements, and barriers are sequentially composed, especially, task constructs and statements must not contain task constructs or barriers.

## 3 Verifying Correctness of Pipeline Implementations

The goal of our verification is to determine whether a code segment parallelized with the pipeline pattern behaves functionally equivalent to its sequential execution. Our verification algorithm shown in Algorithm 1 consists of four steps. First, it constructs a task dependency graph (Sec. 3.1) that represents the specified constraints on the execution order of the tasks. Then, it inspects which of the RAW and WAR dependencies in the sequential execution that cross task boundaries are reflected in the task dependency graph (Sec. 3.2). Read-write conflicts (i.e., RAW or WAR dependencies) that are not represented in the task dependency graph may be eliminated with barriers or proper data-sharing attributes, e.g., the variable can be private in both tasks. The third step checks this. Finally, the last step (Sec. 3.3) analyzes write-after-write dependencies and ensures that the tasks get the correct input values and make their output available. Algorithm 1 will return $\perp$ if it can prove that the pipeline pattern is correctly implemented.[6] Otherwise, it outputs a read-write or write-write conflict on variable $v$, which may threaten functional equivalence.

### 3.1 Constructing Task Dependency Graphs

A task dependency graph provides information about execution constraints, especially order constraints, on tasks. A vertex of the graph represents a task construct (task directive plus associated code region) or a statement that is not part

[5] Currently, we support *for* and *while* loops.

[6] Note that Algorithm 1 assumes, but does not check that the checked code segment follows the pipeline structure described in the previous section. Therefore, its result is only reliable for those segments.

of a task construct but occurs in the pipeline's single region. In the following, we use $V_T := \{t_1, \dots, t_n\}$ to denote the set of task constructs and $V_S := \{s_1, \dots, s_m\}$ to denote the statements. An edge $(v_1, v_2) \in (V_T \cup V_S) \times (V_T \cup V_S)$ describes a dependency between $v_1$ and $v_2$, e.g., if $v_1 \in V_T$, then a task generated for task construct $v_2$ (statement $v_2$) must be executed after all previously generated tasks for task construct $v_1$ finished. Note that our task dependency graph does not include dependencies from $V_S \times V_S$ because the statements in $V_S$ are executed by a single thread, which executes them in the same order as in the sequential execution. Next, let us discuss how to compute the edges.

*Dependency Edges from Depend Clauses.* First, let us consider dependencies between task constructs that origin from depend clauses. Remember that our depend definition (Def. 2) captures these dependencies. In the task dependency graph, these order constraints are represented by the depend edges.

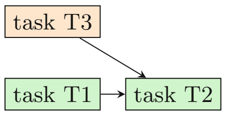
$$E_{\text{depend}} := \{(t_i, t_j) \in V_T \times V_T \mid t_j \text{ depends on } t_i\} \tag{1}$$

*Dependency Edges from Barriers.* Next to depend clauses, also barriers (`barrier` or `taskwait` directives) introduce dependencies. For example, a barrier ensures that no two tasks of the same task construct can execute in parallel. There exists a self-dependency for all task constructs if the set of barriers $B$ is non-empty.

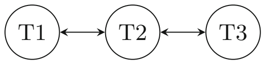
$$E_{\text{self}} := \begin{cases} \varnothing & \text{if } B = \varnothing \\ \{(v, v) \in V_T \times V_T\} & \text{otherwise} \end{cases} \tag{2}$$

```
//T1
#pragma omp task depend(out:a)
{...}

//T2
#pragma omp task depend(in:a) depend(out:b)
{...}

//#pragma omp barrier

//T3
#pragma omp task depend(in:b)
{...}
```

(a) Loop body of pipeline implementation



(b) Task dependencies



(c) Dependency edges from depend clauses

**Fig. 2.** Demonstrating unsoundness of transitive dependency edges

Similarly, we use barriers to add some of the transitive edges from $E_{\text{depend}}$. Since our graph only considers task constructs and cannot distinguish tasks generated in different iterations of the enclosing loop, not all transitive edges from $E_{\text{depend}}$ can be considered without becoming unsound.[7] For example, consider

[7] In contrast, leaving out some of those edges only makes our approach imprecise.

the pipeline implementation sketched in Fig. 2a. Figure 2b shows task T3 constructed for task construct T3 in iteration i, tasks T1 and T2 constructed for task constructs T1 and T2 in iteration i+1, and their dependencies. In addition, Fig. 2c shows the task dependencies $E_{\text{depend}}$.[8] We observe that the transitive edge $(T3, T1)$ is not present in Fig. 2b, although T3 is generated before T1. Thus, edge $(T3, T1)$ must not be added to the task dependency graph. When adding transitive edges, we use that the barrier ensures that at most one task per task construct exists at any point in time and a task belongs to either iteration i (task constructs after the barrier) or iteration i+1 (task constructs before the barrier). Thus, we can safely add transitive dependency edges between task constructs that occur both either before or after the barrier. The transitive edges introduced by a barrier are:

$$E^b_{\text{trans}} := \left((V^b_{\text{before}} \times V^b_{\text{before}}) \cap E_{\text{depend}}\right)^+ \cup \left((V^b_{\text{after}} \times V^b_{\text{after}}) \cap E_{\text{depend}}\right)^+, \tag{3}$$

where $V^b_{\text{before}} \subseteq V$ and $V^b_{\text{after}} \subseteq V$ denote the sets of task constructs before and after barrier $b$ respectively.

In addition, a barrier introduces dependencies between tasks from $V_T$ and statements from $V_S$. A barrier $b$ enforces the single thread to wait until all tasks preceding the barrier finished. Therefore, statements after the barrier ($S^b_{\text{after}}$) depend on the task constructs before the barrier ($V^b_{\text{before}}$). Furthermore, when the barrier was passed new tasks for task constructs before the barrier are only constructed in the next loop iteration, i.e., all statements after the barrier have been executed. Thus, there also exists a dependency between the statements after the barrier and the task constructs before the barrier. These dependencies are captured by the following set of edges.

$$E^b_{\text{stmts}} := \left(V^b_{\text{before}} \times S^b_{\text{after}}\right) \cup \left(S^b_{\text{after}} \times V^b_{\text{after}}\right) \tag{4}$$

So far, we considered dependencies caused by a single barrier. When using multiple barriers, the parallel program either runs tasks between two barriers in parallel, which are generated in the same iteration, or we execute tasks from before the first barrier and tasks after the last barrier concurrently. Even if we cannot distinguish between tasks of different iterations, we know that all tasks before a barrier and all tasks after a barrier depend on each other except when they occur before the first and after the last barrier.[9]

Now, let $B = \{b_1, \ldots, b_n\}$ be the list of barriers in the pipeline implementation such that the barriers in the list are ordered in source code order. Furthermore, let $V^{b_i}_{\text{before}}$ be the statements and task constructs that occur in the pipeline before barrier $b_i$ and $V^{b_i}_{\text{after}}$ those occurring after barrier $b_i$. We use these sets to

[8] Note that Fig. 2b does not contain dependencies between tasks T2 and T1 and tasks T2 and T3 because $T3$ and $T1$ are generated before $T1$.

[9] The same holds for pairs of tasks and statements.

**Algorithm 2:** checkRWEdges($G = (V, E)$)

```
  Input: G - depend graph, where $V = V_T \cup V_S$
1 $potentialViolations := \varnothing$
2 foreach $v \in Vars$ do
3     $T_R :=$ readIn$(V, v)$        ▷ returns tasks in which $v$ is read
4     $T_W :=$ writtenIn$(V, v)$     ▷ returns tasks in which $v$ is written
5     foreach $(t_r, t_w) \in$ getDeps$(T_R, T_W) \setminus (V_S \times V_S)$ do
6         if $(t_r, t_w) \notin E$ then
7             $potentialViolations := potentialViolations \cup (t_r, t_w, v)$
8 return $potentialViolations$
```

define the dependencies discussed above.

$$E^B_{\text{barrier}} := \bigcup_{i=1}^{|B|} \left\{ (v_b, v_a), (v_a, v_b) \mid v_b \in V^{b_i}_{\text{before}} \setminus V^{b_1}_{\text{before}} \wedge v_a \in \times V^{b_i}_{\text{after}} \setminus V^{b_n}_{\text{after}} \right\} \tag{5}$$

Summing up, the dependency edges from a set of barriers $B$ are:

$$E_{\text{barrier}} := E_{\text{self}} \cup E^B_{\text{barrier}} \cup \bigcup_{b \in B} \left( E^b_{\text{trans}} \cup E^b_{\text{stmts}} \right) \tag{6}$$

Now, we have everything at hand to define the task dependency graph.

**Definition 3.** *A* task dependency graph *is a directed graph*

$$G = (V_T \cup V_S, E_{depend} \cup E_{barrier}).$$

### 3.2 Inspecting RAW and WAR Dependencies

In this check, we inspect if the read-after-write and write-after-read dependencies of the sequential execution are respected by the parallel pipeline, i.e., the dependencies occur in the task dependency graph, or they are safely removed from the parallel pipeline. First, we use Algorithm 2 to check which of the dependencies are present in the task dependency graph. Thereafter, we call Algorithm 3 to check whether all dependencies not present in the task dependency graph are safely removed by the pipeline implementation.

To compare the program dependencies with the task dependency graph, Algorithm 2 iterates over the variables. For each variable, it first computes which component, i.e., task construct or statement in the pipeline, reads and which writes the variable. Then, it calls the method `getDeps` to compute all pairs $(t_r, t_w)$ of reading and writing components that have a RAW or WAR conflict on variable $v$. Note that this is sufficient and we do not need to distinguish between RAW and WAR dependencies because OpenMP and our task dependency graph do not distinguish them.[10] Next, Algorithm 2 checks whether all

[10] While one can reflect RAW and WAR dependencies with OpenMP depend clauses, a RAW depend specification can prevent a WAR dependency and vice versa.

**Algorithm 3:** checkPotentialViolations(*potentialViolations*)

```
1  foreach $(t_r, t_w, v) \in potentialViolations$ do
2      $d_r$ := getDataSharingAttribute($t_r$, $v$)
3      $d_w$ := getDataSharingAtrribute($t_w$, $v$)
4      if $d_r$ = firstprivate $\wedge (d_w \neq$ shared $\vee$
5          $d_w =$ shared $\wedge$ $G$.existsBarrierBetween($t_r$, $t_w$)) then
6          continue;
7      if firstWrite($t_r$, $v$) < firstRead($t_r$, $v$)
8        $\wedge$ ($d_r \neq$ shared $\vee$ $d_w \neq$ shared) then
9          continue;
10     return $(t_r, t_w, v)$
```

those pairs are reflected in the task dependency graph. However, it excludes all pairs $(t_r, t_w)$ from $V_S \times V_S$ because they are executed in sequential order. If the task dependency graph does not contain a corresponding edge $(t_r, t_w)$ and the dependency is not an intra-task dependency, a potential dependency violation is found and stored in the set of `potentialViolations`.

In a second step, Algorithm 3 checks the potential violations. Under certain conditions a parallel pipeline is still correctly implemented although it misses a dependency. In general, the variable must not be shared, i.e., at least one of the components uses a thread-local copy to prevent data races, and the read access must still return the same value as a sequential execution. Currently, we support two cases. First, the read variable is allowed to be `firstprivate` if the written variable is either not `shared` or is `shared` and there exists a barrier between the read and write accesses that ensures that the read variable is initialized with the correct value. Second, if a component always writes to variable $v$ before it reads variable $v$, the read does not depend on other components and could be performed on a different copy without altering the behavior. Therefore, a dependency can be missing if the component always writes to variable $v$ before it reads variable $v$ and variable $v$ is (first)private. Note that Algorithm 3 only checks that the read-write conflicts are eliminated, but does not check whether the elimination affects the functional behavior. The latter is considered by the next algorithm.

## 3.3 Checking WAW Dependencies and I/O Availability

In this check, we examine whether all WAW dependencies in the sequential execution are handled appropriately, whether all read accesses in the pipeline see the same value as a sequential execution, and whether the computation result is available after the pipeline execution. To inspect the WAW dependencies, we inspect all variables $v$ and check that all pairs of pipeline components (task constructs or statements from $V_S$) that write to $v$ are either ordered or at least one component uses a thread-local copy of the variable. Thus, we ensure that writes to the same variable cannot interfere.

**Algorithm 4:** checkRemainingDependencies($G = (V, E)$)

```
Input: G - depend graph
// check write-write dependencies
foreach $(t_{w_1}, t_{w_2}) \in ((T_W \times T_W) \setminus (V_S \times V_S))$ do
    if getDataSharingAttribute($t_{w_1}, v$) = shared ∧
      getDataSharingAttribute($t_{w_2}, v$) = shared ∧ $(t_{w_1}, t_{w_2}) \notin E$ then
        return $(t_{w_1}, t_{w_2}, v)$

//check reads
foreach $t_r \in T_R$ do
    $d_r$ := getDataSharingAttribute($t_r, v$)
    if $d_r$ = private ∧ firstWrite($t_r$, $v$) > firstRead($t_r$, $v$) then
        return $(t_r, -, v)$

//check output availability
foreach $v \in$ GetLiveVars do
    foreach $t_w \in T_W$ do
        $d_w$ :=GetDataSharingAttribute($t_w, v$)
        if $d_w \neq$ shared then
            return $(-, t_w, v)$

foreach $v \in$ Vars do
    $T_R$ := readIn($V, v$)              ▷ returns tasks in which $v$ is read
    $T_W$ := writtenIn($V, v$)           ▷ returns tasks in which $v$ is written
    foreach $(t_r, t_w) \in T_R \times T_W$ do
        $d_r$ := getDataSharingAttribute($t_r, v$)
        $d_w$ := getDataSharingAttribute($t_w, v$)
        if firstWrite($t_r$, $v$) < firstRead($t_r$, $v$) then
            continue;
        if $d_r$ = firstprivate ∧ $d_w$ = shared
          ∧ !G.existsBarrierBeforeOrAfter($t_r, t_w$) then
            return $(t_r, t_w, v)$

return $\perp$
```

After we checked the RAW, WAR, and WAW dependencies, we know that reads and writes are ordered as in the sequential execution or they are performed on local copies. To show functional equivalence between the parallel pipeline and the sequential execution, it remains to be shown that a variable read returns the same value in both cases and we get the same values when reading variables after the pipeline. To ensure that the correct value can be read in the pipeline, the algorithm checks that read variables are only `private` if they are defined in a node (task construct or statement) before they are read. To allow that a write can be propagated to a read in or after the pipeline, we check that all variables written in the pipeline that are live have data-sharing attribute `shared`, i.e., we can access the modified value. Finally, we need to check whether we read the correct value in the pipeline. We already know that for each read-write pair, there either exists a dependency edge, i.e., the read and write cannot occur concurrently, or Algorithm 3 already checked that the correct values are read. In addition, a read variable can only be `private` if it is defined in the node before it is read. The only reason why an incorrect value might be read are `firstprivate` variables, which are initialized at task creation. `Firstprivate` variables are unproblematic if the initialization value is irrelevant (i.e., they are defined before read) or a barrier between reading and writing node ensures that during task generation the same value as in a sequential execution is used for initialization.

## 3.4 Handling of Loop Header

To include the statements of the loop header into our analysis we consider them to be part of the loop body. The condition test is the first statement of the body while increment statements are placed right at the end. All these statements are executed by the single thread and can be treated the same way as regular statements placed outside of tasks.

## 3.5 Implementation

We implemented the algorithms in a prototype tool to check pipeline parallelizations of C programs. Our prototype builds on the ROSE compiler framework [18]. Next, we describe how we implemented the predicates used in the algorithms.

`readIn/writtenIn`. We use ROSE's def-use analysis to determine the variable usages and definitions (AST nodes) in the loop. Based on the position of the AST node in the code, we identify the corresponding node (task construct or statement from $V_S$) in the pipeline. Arrays are treated like scalar variables, i.e. array indices are ignored.

`getDeps(`$T_R, T_W$`)`. Based on the nodes reading or writing a variable $v$, our prototype compute a coarse, but fast overapproximation of the read-write dependencies on variable $v$ in the sequential program, namely the Cartesian product $T_R \times T_W$ of nodes reading and writing variable $v$.

`getDataSharingAttribute(`$t, v$`)`. We try to determine the data-sharing attribute based on the corresponding OpenMP declarative and fallback to the rules if it is not explicitly specified.

`firstRead(`$t, v$`)/firstWrite(`$t, v$`)`. To compute these predicates, we take all reads into account, but only consider write accesses that occur on every execution path, i.e., that are not part of a branch or loop body. Since our algorithms only check whether there always exists a write to $v$ before any read of $v$, this approximation is sound, but imprecise. However, the approximation allows us to use source code lines in the implementation of the predicates. More concretely, the predicates return the source line number of the first read access of $v$ in $t$ and the first write access of $v$ in $t$ that is considered. In case there is no read and write access respectively, the source line number of the end of $t$ is returned.

`existsBarrierBetween(`$t_r, t_w$`)`. We use code lines to decide this predicate. Note that this is only valid because we assume that tasks, barriers, and statements (from $V_S$) are sequentially composed and must not be nested.
Let $t.\mathbf{loc}$ and $b.\mathbf{loc}$ be the source code line of the beginning of a node $t \in V_T \cup V_S$ and a barrier $b$. To determine the truth value of the predicate, our implementation checks the following formula by iterating over all barriers $b$.

$$\exists b \in B : \min(t_r.\mathbf{loc}, t_w.\mathbf{loc}) < b.\mathbf{loc} < \max(t_r.\mathbf{loc}, t_w.\mathbf{loc})$$

`existsBarrierBeforeOrAfter(`$t_r, t_w$`)`. Similar to `existsBarrierBetween`, we use code lines and iterate over the barriers $b$ to decide the following formula.

$$\exists b \in B : b.\mathbf{loc} < t_r.\mathbf{loc} \vee b.\mathbf{loc} > t_w.\mathbf{loc}$$

`getLiveVars`. Our implementation returns all modified variables.[11]

So far, our prototype realizes a restricted implementation of the verification technique, which was sufficient for our initial evaluation. For example, the implementation is limited to scalar variables and arrays and, as already mentioned, arrays are handled like scalar variables, i.e. so we do not distinguish different indices. However, the algorithm is not limited to these data types. Adding basic struct support is simple. One could handle struct accesses similar to arrays. To soundly support pointers one however requires a points-to analysis. Using a more fine grained notion of variables, e.g., on the basis of memory locations, allows one to differentiate between different array elements.

Also, recursive function calls are currently not supported. They might violate the assumptions that tasks are not nested. Since nested task are not siblings of all other tasks, a depend clause of a non-sibling tasks has no effect. To support recursion, we, therefore, need to analyze which tasks are siblings and consider this when determining the dependency edges from depend clauses.

Furthermore, the implementations of `readIn`, `writtenIn`, `firstRead`, and `firstWrite` are intra-procedural. In our context, the current implementation of `firstWrite` is sound. To soundly support function calls in `readIn` and `writtenIn`, we could e.g., assume that called functions read and write all global variables and passed parameters. For `firstRead`, we could also assume that called functions read all global variables and associate those reads with the called functions.

In addition, the precision of the prototype can be further improved by using more precise implementations of the above predicates. For example, the `getDeps` predicate could take the control-flow into account. The `getLiveVars` can be refined by applying ROSE's live variable analysis. Furthermore, one can use definition-use chains to improve the predicates `firstRead` and `firstWrite`.

[11] Although read-only variables are excluded, this is sufficient because Algorithm 4 only checks live and modified variables.

**Table 1.** Evaluation results showing for each example, the expected and reported result, the size of the task dependency graph, and the number of barriers $B$

| Task | Expected result | Reported result | $\|V_S\| + \|V_T\|$ | $\|E\|$ | $\|B\|$ |
|---|---|---|---|---|---|
| DRB072-taskdep1-orig-no.c | ✓ | ✓ | 0+2 | 3 | 0 |
| DRB072-taskdep2-orig-no.c | ✓ | ✓ | 0+2 | 4 | 0 |
| DRB072-taskdep3-orig-no.c | ✓ | ✓ | 0+3 | 5 | 0 |
| DRB120-barrier-orig-no.c | ✓ | ✓ | 2+0 | 4 | 1 |
| DRB131-taskdep4-orig-yes-omp45.c | ✓ | ✗ | 4+3 | 16 | 1 |
| DRB132-taskdep4-orig-no-omp45.c | ✗ | ✗ | 4+3 | 25 | 1 |
| DRB133-taskdep5-orig-no-omp45.c | ✗ | ✗ | 4+3 | 29 | 1 |
| DRB134-taskdep5-orig-yes-omp45.c | ✓ | ✗ | 4+3 | 20 | 1 |
| DRB135-taskdep-mutexinoutset-orig-no.c | ✗ | ✗ | 0+6 | 17 | 0 |
| DRB136-taskdep-mutexinoutset-orig-yes.c | ✓ | ✗ | 0+6 | 9 | 0 |
| DRB165-taskdep4-orig-yes-omp50.c | ✓ | ✗ | 2+3 | 8 | 1 |
| DRB166-taskdep4-orig-no-omp50.c | ✗ | ✗ | 2+3 | 15 | 1 |
| DRB167-taskdep4-orig-no-omp50.c | ✗ | ✗ | 2+3 | 19 | 1 |
| DRB168-taskdep5-orig-yes-omp50.c | ✓ | ✗ | 2+3 | 12 | 1 |
| eos-mbpt-hf-interpolate/pipeline_1:27.c | ✓ | ✗ | 0+2 | 3 | 0 |
| Kastors/strassen-task-dep.c | ✓ | ✓ | 0+19 | 305 | 8 |
| Kastors/strassen-task.c | ✓ | ✓ | 0+19 | 361 | 1 |

## 4 Evaluation

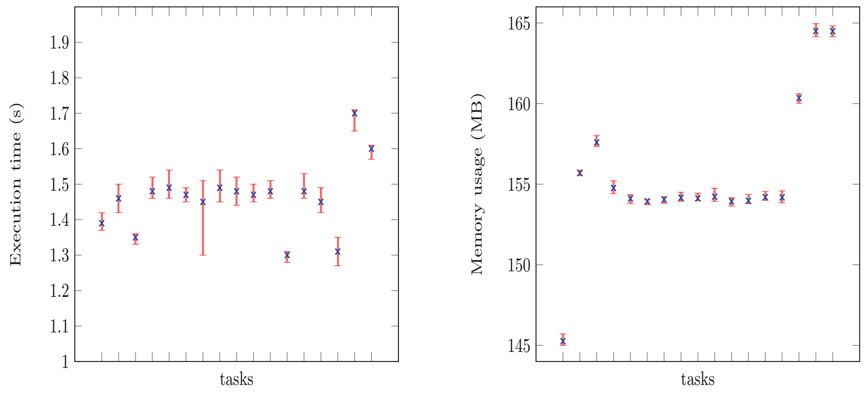
Our goal is to demonstrate the applicability of our verification approach. Note that we could not compare our approach to the closely related approach of Royuela et al. [19] because we failed to find out how to run their analysis.

**Benchmark Tasks.** We looked at the DataRaceBench [11] and KASTORS benchmark [25] and selected all examples that contain task parallelism and use `depend` clauses. Our selection results in 14 examples from the DataRaceBench and two from the KASTORS benchmark. To get a syntactical pipeline implementation, we added a loop to the tasks. In addition, we consider one potential pipeline implementation suggested by the auto-parallelizer DiscoPoP [10] (`eos-mbpt-hf-interpolate/pipeline_1:27.c`). In total, we use 17 examples.

**Environmental Setup.** We run our experiments 5-times on an Ubuntu 18.04 machine with an Intel Core i7 CPU and 32 GB of RAM.

### 4.1 Experimental Results

*RQ 1: Is our algorithm sound and does it detect correct pipelines?* Table 1 shows for each of the 17 tasks the expected result, the reported result, the number of nodes and edges in the task dependency graph as well as the number of barrier statements in the pipeline implementation. Looking at Table 1, we observe that our algorithm rejects all incorrect results. Thus, it is sound on out examples. Also, it detects 50% of the correct pipeline implementations, but rejects the



(a) Execution time (b) Memory usage

**Fig. 3.** Per task, average, maximum and minimum execution time (left) and average, maximum and minimum memory usage (right) of five executions

other half. The main reason for rejection is that the algorithm fails to detect that inter-iteration depenendencies can be ignored for loops with one iteration.

*RQ 2: How does our algorithm scale?* To analyze the scalability, we consider the size of the task dependency graph, which includes the number of tasks ($|V_T|$) and statements ($|V_S|$), as well as the number of barriers in the program. The last three columns of Table 1 provide these numbers. While our set of tasks is too small to come to a definite conclusion, we can observe that the number of edges increases superlinear with the number of tasks and statements (the nodes of the task dependency graph). Furthermore, the last two rows show two parallelizations of the same sequential program. One uses barriers and the other depend clauses. We observe that for the two examples, barriers are more efficient.

*RQ 3: How efficient is our algorithm?* To evaluate the efficiency of our algorithm, we consider the execution time and memory usage shown in Fig. 3a and 3b. For each example, the two plots show the average of five executions together with the maximum and minimum value. The examples are ordered as in Table 1.

We observe that the execution times and memory usage are rather low and often similar. The reason is that for our examples the parsing dominates the execution time and memory usage. Furthermore, we observe that for the analysis of the larger KASTORS tasks (last two examples) we require significantly more time and memory. We also see that the first example requires significantly less memory than the others because it does not call library functions. Another outlier is the DiscoPoP example (third last). The task itself contains more code than most of the other tasks, which leads to larger memory usage. However, the pipeline section itself is rather small. Thus, the analysis time is short.

## 5 Related Work

Several static and dynamic analyses [2–4,9,13,14,19–21,23,28] for OpenMP programs exist. We focus on the static analyses [3,4,13,19,20,23,28]. Some of the static analyses specialize on specific properties like concurrent access [13,19,28], the absence of data races [4,23,26] or deadlocks [20]. In contrast, we examine behavioral equivalence between the parallel and sequential execution.

Equivalence checkers like Pathg [27], CIVL [22], PEQCHECK [7], or the approach of Abadi et al. [1] examine equivalence, but these checkers are general purpose checkers, which ignore applied parallel design patterns and the structure of the parallelization. In contrast, PatEC [6], AutoPar's correctness checker [12], CIVL's OpenMP simplifier [22], or ompVerify [3] analyze equivalence of sequential and parallelized for loops, but they do not support pipeline implementations.

The approach closest to ours is the one by Royuela et al. [19]. Like our approach, they focus on tasks and analyze depend and data-sharing clauses. While we construct a task dependency graph, they represent the task dependencies in an extended control-flow graph. Furthermore, we explicitly check equivalence while Royuela et al. [19] only look for common parallelization mistakes.

Finally, we would like to mention that there exist approaches [8,24] that verify whether a sequential loop optimization that aims at a better utilization of the processor pipeline is correct.

## 6 Conclusion

While the CPU frequency remains static, the number of cores per CPU increases. To speed up a program, one must execute it on multiple CPU cores.

OpenMP is a widely used API that programmers utilize to transform their programs into multi-threaded programs. To this end, the programmers typically only insert OpenMP directives. Unfortunately, not all OpenMP-compliant programs are correct. Thus, programmers should analyze whether an OpenMP parallelization is behavior preserving.

In this paper, we propose an automatic technique to support a programmer with this analysis task. Our technique utilizes that programmers often consider parallel design patterns when parallelizing programs. More concretely, we develop a specific technique to verify that a parallelization that applies the pipeline pattern is behavior preserving. To ensure that the behavior is preserved, our technique aims to check that a read access to a variable returns the same value in the sequential and parallelized program. Therefore, it analyzes whether the dependencies on variables accesses that exist in the sequential program are properly considered in the parallelization and that data-sharing attributes do not prevent reading the proper values.

To test our technique, we implemented it in a prototype tool and evaluated it on 17 examples. Our technique overapproximates, and, thus, our implementation failed to detect all correct parallelizations, i.e., it is not complete. While completeness is desirable, soundness is important. In our evaluation, our implementation behaved soundly and successfully detected all incorrect parallelizations.

## References

1. Abadi, M., Keidar-Barner, S., Pidan, D., Veksler, T.: Verifying parallel code after refactoring using equivalence checking. Int. J. Parallel Program. **47**(1), 59–73 (2018). https://doi.org/10.1007/s10766-017-0548-4
2. Atzeni, S., et al.: ARCHER: effectively spotting data races in large OpenMP applications. In: Proceedings IPDPS, pp. 53–62. IEEE (2016). https://doi.org/10.1109/IPDPS.2016.68
3. Basupalli, V., et al.: ompVerify: polyhedral analysis for the OpenMP programmer. In: Proceedings IWOMP, pp. 37–53. LNCS 6665, Springer (2011). https://doi.org/10.1007/978-3-642-21487-5_4
4. Bora, U., Das, S., Kukreja, P., Joshi, S., Upadrasta, R., Rajopadhye, S.: LLOV: a fast static data-race checker for openMP programs. TACO **17**(4) (2020). https://doi.org/10.1145/3418597
5. Goncalves, R., Amaris, M., Okada, T.K., Bruel, P., Goldman, A.: OpenMP is not as easy as it appears. In: Proceedings HICSS, pp. 5742–5751. IEEE (2016). https://doi.org/10.1109/HICSS.2016.710
6. Jakobs, M.: PatEC: pattern-based equivalence checking. In: Laarman, A., Sokolova, A. (eds.) SPIN 2021, LNCS, vol. 12864, pp. 120–139 (2021). https://doi.org/10.1007/978-3-030-84629-9_7
7. Jakobs, M.C.: PEQcheck: localized and context-aware checking of functional equivalence. In: Proceedings FormaliSE, pp. 130–140. IEEE (2021), https://doi.org/10.1109/FormaliSE52586.2021.00019
8. Leviathan, R., Pnueli, A.: Validating software pipelining optimizations. In: Proceedings CASES, pp. 280–287. ACM (2002). https://doi.org/10.1145/581630.581676
9. Li, J., Hei, D., Yan, L.: Correctness analysis based on testing and checking for OpenMP Programs. In: Proceedings ChinaGrid, pp. 210–215. IEEE (2009). https://doi.org/10.1109/ChinaGrid.2009.12
10. Li, Z., Atre, R., Huda, Z.U., Jannesari, A., Wolf, F.: Unveiling parallelization opportunities in sequential programs. J. Syst. Softw. 282–295 (2016). https://doi.org/10.1016/j.jss.2016.03.045
11. Liao, C., Lin, P.H., Asplund, J., Schordan, M., Karlin, I.: DataRaceBench: a benchmark suite for systematic evaluation of data race detection tools. Proc. SC. ACM (2017). https://doi.org/10.1145/3126908.3126958
12. Liao, C., Quinlan, D.J., Willcock, J., Panas, T.: Extending automatic parallelization to optimize high-level abstractions for multicore. In: Proceedings IWOMP, pp. 28–41. LNCS 5568, Springer (2009). https://doi.org/10.1007/978-3-642-02303-3_3
13. Lin, Y.: Static Nonconcurrency analysis of OpenMP programs. In: Proceedings IWOMP, pp. 36–50. LNCS 4315, Springer (2005). https://doi.org/10.1007/978-3-540-68555-5_4
14. Ma, H., Diersen, S., Wang, L., Liao, C., Quinlan, D., Yang, Z.: Symbolic analysis of concurrency errors in OpenMP programs. In: Proceedings ICPP. pp. 510–516. IEEE (2013). https://doi.org/10.1109/ICPP.2013.63
15. Mattson, T.G., Sanders, B.A., Massingill, B.L.: Patterns for parallel programming. Addison-Wesley Professional (2013)
16. McCool, M., Reinders, J., Robison, A.: Structured Parallel Programming: Patterns for Efficient Computation. Morgan Kaufmann Publishers Inc., Burlington (2012)
17. OpenMP: OpenMP application programming interface (version 5.1). Technical report OpenMP Architecture Review Board (2020). https://www.openmp.org/specifications/

18. Quinlan, D., Liao, C.: The ROSE source-to-source compiler infrastructure. In: Cetus users and compiler infrastructure workshop, in conjunction with PACT, vol. 2011, p. 1. Citeseer (2011)
19. Royuela, S., Ferrer, R., Caballero, D., Martorell, X.: Compiler analysis for OpenMP tasks correctness. In: Proceedings CF, pp. 11–19. ACM (2015). https://doi.org/10.1145/2742854.2742882
20. Saillard, E., Carribault, P., Barthou, D.: Static Validation of barriers and work-sharing constructs in openmp applications. In: Proceedings IWOMP, pp. 73–86. LNCS 8766, Springer (2014). https://doi.org/10.1007/978-3-319-11454-5_6
21. Salamanca, J., Mattos, L., Araujo, G.: Loop-carried dependence verification in OpenMP. In: Proceedin IWOMP, pp. 87–102. LNCS 8766, Springer (2014). https://doi.org/10.1007/978-3-319-11454-5_7
22. Siegel, S.F., et al.: CIVL: the concurrency intermediate verification language. In: Proceedings SC, pp. 61:1–61:12. ACM (2015). https://doi.org/10.1145/2807591.2807635
23. Swain, B., Li, Y., Liu, P., Laguna, I., Georgakoudis, G., Huang, J.: OMPRacer: a scalable and precise static race detector for OpenMP programs. In: Proceedings SC. IEEE (2020). https://doi.org/10.1109/SC41405.2020.00058
24. Tristan, J., Leroy, X.: A simple, verified validator for software pipelining. In: Proceedings POPL, pp. 83–92. ACM (2010). https://doi.org/10.1145/1706299.1706311
25. Virouleau, P., Brunet, P., Broquedis, F., Furmento, N., Thibault, S., Aumage, O., Gautier, T.: Evaluation of OpenMP dependent tasks with the KASTORS benchmark suite. In: Proceedings IWOMP, pp. 16–29. LNCS 8766, Springer (2014). https://doi.org/10.1007/978-3-319-11454-5_2
26. Ye, F., Schordan, M., Liao, C., Lin, P., Karlin, I., Sarkar, V.: Using polyhedral analysis to verify openmp applications are data race free. In: Proceedings CORRECTNESS@SC, pp. 42–50. IEEE (2018). https://doi.org/10.1109/Correctness.2018.00010
27. Yu, F., Yang, S., Wang, F., Chen, G., Chan, C.: Symbolic consistency checking of openmp parallel programs. In: Proceedings LCTES, pp. 139–148. ACM (2012). https://doi.org/10.1145/2248418.2248438
28. Zhang, Y., Duesterwald, E., Gao, G.R.: Concurrency analysis for shared memory programs with textually unaligned barriers. In: Proceedings LCPC, pp. 95–109. LNCS 5234, Springer (2007). https://doi.org/10.1007/978-3-540-85261-2_7