Detecting Non-sibling Dependencies in OpenMP Task-Based Applications

Ricardo Bispo Vieira[1(✉)], Antoine Capra[2], Patrick Carribault[3], Julien Jaeger[3], Marc Pérache[3], and Adrien Roussel[3]

[1] Exascale Computing Research Lab, Bruyères-le-Châtel, France
ricardo.bispo-vieira@exascale-computing.eu
[2] Bull/Atos SAS, Les Clayes-sous-Bois, France
antoine.capra@atos.net
[3] CEA, DAM, DIF, 91297 Arpajon, France
{patrick.carribault,julien.jaeger,marc.perache,adrien.roussel}@cea.fr

Abstract. The advent of the multicore era led to the duplication of functional units through an increasing number of cores. To exploit those processors, a shared-memory parallel programming model is one possible direction. Thus, OpenMP is a good candidate to enable different paradigms: data parallelism (including loop-based directives) and control parallelism, through the notion of tasks with dependencies. But this is the programmer responsibility to ensure that data dependencies are complete such as no data races may happen. It might be complex to guarantee that no issue will occur and that all dependencies have been correctly expressed in the context of nested tasks. This paper proposes an algorithm to detect the data dependencies that might be missing on the OpenMP task clauses between tasks that have been generated by different parents. This approach is implemented inside a tool relying on the OMPT interface.

Keywords: OpenMP task · Nested task · OMPT · Data dependency · Data-race

1 Introduction

The advent of multi-core processors occurred more than a decade ago, bringing processors scaling from a few cores to several hundreds. To exploit those functional units, the OpenMP programming model [1] became the *defacto* standard leveraging the programmability and the performance of such systems. Based on compiler directives and the fork-join model, it spawns threads and implies a synchronization *rendez-vous* at the end of parallel regions. Mainly oriented to structured and regular parallelism first, it has been extended with a task programming model to enable efficient use of irregular and nested parallelism. Even if this tasking model has proven to provide good performance, global synchronizations are expensive and may prevent scheduling of upcoming tasks. Therefore, the notion of *data dependency* has been introduced, to provide a lighter

X. Fan et al. (Eds.): IWOMP 2019, LNCS 11718, pp. 231–245, 2019.
https://doi.org/10.1007/978-3-030-28596-8_16

local synchronization between successive dependent tasks. These dependencies can be expressed only between sibling tasks (i.e., created by the same parent task). The execution order of these tasks is given by the creation sequence (`task` directives order in the code) and the `depend` clauses. We call the sets of tasks spawned from the same parent a *dependency domain*.

Combining nested tasks with data dependencies may lead to some issues because of the parallel execution of tasks between dependency domains. Indeed, dependencies only apply between sibling tasks. However, these dependencies are not passed on the next generation of tasks. Hence, two dependency domains issued from sibling tasks with dependencies will not inherit their parent order. In this case, race conditions may occur even if the programmer thinks the dependencies are correctly expressed in the `depend` clauses. Correctly specifying a large number of dependencies across multiple dependency domains implies a non negligible burden to the developer and remains error prone.

In this paper, we aim at detecting such dependency declaration errors. The contribution of this paper is threefold: (1) we develop an algorithm to detect possible data races based on declared task dependencies, (2) we propose new extensions to the OMPT interface for keeping track of the memory scope of dependency variables and, (3) we implement the OMPT extensions in an OpenMP implementation and the algorithm in a tool to effectively detect data races.

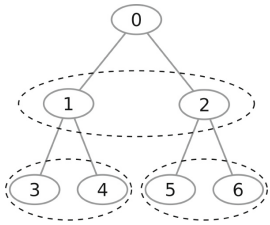
The remaining of the paper is organized as follows: Sect. 2 presents some motivating examples. Related work regarding nested tasks with data dependencies and their correctness is presented in Sect. 3. Then, Sect. 4 explains the main contribution through the dynamic detection of race conditions among data dependencies in non-sibling tasks. Section 5 describes the implementation of our approach while Sect. 6 illustrates our tool output and its overhead on some applications, before concluding in Sect. 7.

2 Motivating Examples

When considering nested tasks, each task in a dependency domain generates its own children tasks, hence its own dependency domain. By representing each task with a vertex, and linking each task to its children with an edge, it results a tree structure. We call such tree a *spawn-tree*. Since dependencies can only induce scheduling constraints inside a dependency domain (i.e., between sibling tasks), there is no ordering between tasks from different domains. Thus, these tasks can run concurrently in any order, even if they are at different levels in the spawn-tree. Indeed, specifying a dependency clause at a given level in the task nest does not propagate it to deeper levels (i.e., to children tasks). This might become tricky as, from the user point of view, dataflow information has been expressed. However, the resulting behavior and scheduling may not be the one expected. We present very simple test cases to illustrate such possible data races with misleading `depend` clauses.

```
1  main(void)
2    variable a,b;
3    #pragma omp parallel {
4      #pragma omp single {
5        #pragma omp task depend(in: a) {
6          #pragma omp task depend(inout: a) {}
7          #pragma omp task depend(inout: a) {}
8        }
9        #pragma omp task depend(out: a) {
10         #pragma omp task depend(inout: a) {}
11         #pragma omp task depend(inout: a) {}
12       }
13     }
14   }
```

Listing (1.1) Nested tasks with dependencies



(a) Single spawn-subtree

Fig. 1. An OpenMP code with nested tasks with dependencies and its corresponding spawn-subtree. Dotted ellipses in the tree are for dependency domains.

Wrongly Expressed Dependencies. Listing 1.1 presents a test case based on nested tasks with data dependencies. The first task (`single` directive - task 0 in the spawn-tree represented in Fig. 1a) spawns two children tasks with dependencies (`task` constructs lines 5 and 9 with `depend(in)` and `depend(out)` clauses - tasks 1 and 2 in the spawn-tree). These tasks belong to the same dependency domain (dotted ellipse around task 1 and 2 in the spawn-tree). Each of these tasks spawns two other children tasks with dependencies (`task` constructs with `depend(inout)` clauses - tasks 3, 4, 5 and 6 in the spawn-tree).

The parents tasks 1 and 2 have serialized dependencies over a, ensuring an order. However, their children don't inherit this dependency. Without any `taskwait` directive at the end of task 1 to ensure that all its children tasks have finished before task 1 ends, all the tasks at the last level of the tree can run concurrently. Moreover, the children of task 1 can run concurrently with task 2. If the variable a is effectively written as suggested in the depend clauses, a race condition on a may happen.

```
1  main(void)
2    variable a,b;
3    #pragma omp parallel {
4      #pragma omp single {
5        #pragma omp task depend(in: a) {
6          #pragma omp task depend(inout: a) {}
7      #pragma omp taskwait
8        }
9        #pragma omp task depend(in: a) {
10         #pragma omp task depend(inout: a) {}
11     #pragma omp taskwait
12       }
13     }
14   }
```

Listing 1.2. Unexpressed/Hidden dependencies

Unexpressed/Hidden Dependencies. Listing 1.2 presents a similar test case, except that parent tasks don't express a data dependency over a. In this case, adding a `taskwait` directive is not enough to order all tasks. Since there is no inferred writing of a between the parent tasks, they can be executed in any order, and even concurrently. Hence, ensuring that all children tasks are finished

does not enforce an order between other tasks at the same tree level, as the tasks from the two lower dependency domains can run concurrently, also causing data races. One possibility to solve this issue is to apply children dependency clauses to the parent tasks. Thus, these dependencies are said to be *unexpressed* (or *hidden*).

Listing 1.3 presents the same behavior: two tasks are spawned inside `parallel` construct. Even without nested tasks, the same problem occurs due to implicit tasks. Indeed, the `parallel` construct first spawns implicit tasks (one per OpenMP thread). Due to these implicit tasks, the explicit task creations (`task` constructs) represents the second level in the spawn-tree. For example, if one considers 2 OpenMP threads, thus two implicit tasks, this listing produces the same spawn-tree as depicted in Fig. 1a. The implicit task level cannot accept `depend` clauses. Hence, the dependencies expressed on the explicit tasks are hidden to the implicit ones, causing the ordering issue as before. The same applies when creating explicit tasks with dependencies in a worksharing-Loop construct.

```
1  variable a;
2  main(void)
3    #pragma omp parallel {
4      #pragma omp task depend(in: a) {}
5      #pragma omp task depend(inout: a) {}
6    }
```

Listing 1.3. Tasks with dependencies in implicit task

3 Related Work

The OpenMP support for tasks with dependencies has shown a growing interest from the community of developers and researchers, in various topics such as scheduling [4], data locality [3] and more generally performance optimization [6,7]. Thus, Perez et al. propose an extension of the OpenMP task directive to apply dependencies between different family lineage of domain dependencies [2]. The new clauses `weakwait`, `weakin`, `weakout`, `weakinout` and `release` refine the dependency relationship in a two-step process starting by applying inner-task dependencies directly to the outer-task successors at a `weakwait` synchronization point. Early processing is possible as the `release` clause indicates that no more dependencies will be expressed on the listed variables. Then, outer tasks with a `weak` dependency clause pass down predecessors dependencies to inner subtasks. The results obtained with these new extensions are coherent with the theoretical study conducted by Dinh et al. [8]. In the nested dataflow model (ND), they showed that modified scheduling algorithms achieve better locality reuse and higher performance on large number of processors. ND is the extension of the nested parallel model (NP) with dependencies, where the `fire` construct completes the `parallel` and `sequential` constructs representing partial dependencies between two dependency domains. They introduced a methodology called DAG rewriting system (DRS) to translate from NP to ND and use it to revisit existing linear algebra algorithms, providing material for the modified scheduling proof. But these approaches do not enable debugging of current

OpenMP task-based applications. For this purpose, data-race detection methods exist, based on either static, dynamic, or post-mortem approaches. Nonetheless, the majority only provides support for tasking model without data dependencies. Some tools support tasks with dependencies. Protze et al. [9] proposed an efficient and scalable OpenMP data-race detection tool called Archer based on a static-dynamic method for large HPC applications: relying on a LLVM compilation pass for static analysis and on ThreadSanitizer for dynamic analysis via code instrumentation and *Happens-before* relation. They annotated the OpenMP runtime to reduce false positives arising from synchronizations points and locking. They defined three detection states resulting from static analysis, race free, certainly racy and potentially racy regions. On top of that information, they extend ThreadSanitizer to take as input a blacklisted set of race-free regions, notably reducing amount of instrumentation at dynamic analysis. In [10], they detailed how they reported OpenMP runtime annotations into OMPT events callbacks, providing a portable data race detection tool with support for tasks with dependencies. Matar et al. [11] conducted a similar study mainly oriented to tasking programming model, relying on ThreadSanitizer and the *Happens-before* relation for dynamic analysis. They proved that their tool, Tasksanitizer, is more efficient at task level to detect determinacy races. However, when combining nested tasks with dependencies, their respective solutions might be related to task scheduling, missing some possible race conditions. Our approach does not instrument every memory access, but it tracks dependency clauses and deals with the hierarchy of tasks, whatever the scheduling of those tasks. It is therefore complementary to methods like Archer and Tasksanitizer.

4 Detecting Dependencies Between Non-sibling Tasks

Section 2 showed how unexpressed dependencies or the absence of `taskwait` directive in descendant tasks may lead to data races, despite the expression of dependencies on some tasks. In this Section, we present our approach to detect such wrong behavior. First, we will describe our approach with our main algorithm to detect potential data races based on the expressed dependencies. Then, since dependencies in OpenMP are passed through variables (i.e., logical memory addresses), we present in a second part how we detect that the `depend` clauses on the same address indeed concern the same variable.

4.1 Main Approach

Our main approach to detect potential data races in nested tasks is based on spawn-tree subgraph and their isolation. Each task t in a dependency domain will generate its own subtree in the spawn-tree. This subtree regroups all the tasks having the task t as an ancestor. All the tasks from the subtree of t should be compared with the subtree spawned from the siblings of t. However, these subtrees may not be compared in one case: if the t subtree is *isolated*.

A subtree is *isolated* if all the tasks in the subtree are enforced to be finished before any subtree from a subsequent sibling is started. Thus, the t subtree is isolated from another t' subtree if, and only if, there is an ordering between t and t', and all tasks in t subtree are done before starting task t' and its own subtree. This isolation can be achieved with several methods. The first method consists in putting a `taskwait` directive after the last task of each level in the t spawn-tree. The second method encapsulates task t in a `taskgroup` construct ending before task t'. A third method inserts a `if(0)` clause on each task of the subtree.

If the subtree is isolated, no tasks from the t subtree may run concurrently with t subsequent sibling tasks. On the other hand, if the t subtree is not isolated from the subtrees of t subsequent siblings, tasks of multiple subtrees may run concurrently. In such case, it is necessary to test each task in all subtrees in a pairwise manner to detect `depend` clauses on same addresses. If this occurs, and the address in the multiple `depend` clauses refers to the same variable, then a data race may occur.

Algorithm 1. Resolve Non Sibling Dependencies

```
1  ResolveNonSiblingDependencies
       inputs: vertex root of the spawn-tree
2      if root.children ≠ ∅ then
3          for v ∈ root.children do
4              DoDectectionConflicts = true
5              for v' ∈ root.children ∖ {v} do
6                  if DependencyPath(v, v') = true then
7                      Synched = CheckSynch(v)
8                      if Synched = true then
9                          DoDectectionConflicts = false
10                 if DoDectectionConflicts = true then
11                     for w ∈ subtree(v)∪v do
12                         for w' ∈ subtree(v')∪v' do
13                             DetectConflicts(w, w')
14             ResolveNonSiblingDependencies(v)
```

Algorithm 1 presents these different steps. We will describe it on a small example. Listing 1.4 presents a task-based Fibonacci kernel extracted from the BOTS benchmarks [14] and modified to express dependencies. In this new program, each invocation of the `fib` function creates three tasks: one for each new invocation of the `fib` function, and a third task to realize the sum of the two sub-results. The two `fib` invocations are independent (`depend(out:x)` and `depend(out:y)` clauses respectively), but the last task depends from the two

```
1  fib(n)
2    int x, y, s;
3    if( n < 2 ) return n;
4    #pragma omp task shared(x) depend(out:x) {
5      x = fib( n - 1 );
6    }
7    #pragma omp task shared(y) depend(out:y) {
8      y = fib( n - 2 );
9    }
10   #pragma omp task shared(s,x,y) depend(in:x,y) {
11     s = x + y;
12   }
13   #pragma omp taskwait
14   return s;
```

Listing 1.4. Task-Based Fibonacci with dependencies

previous tasks (`depend(in:x,y)` clause). The computation of `fib(4)` with this new algorithm produces the spawn-tree displayed in Fig. 2a.

In our algorithm, we study each pair of tasks in each dependency domain, starting with the set of tasks generated by the root of the spawn-tree (l.3 and l.5 in the algorithm). Applied to the `fib(4)` example, we start by studying the tasks `fib(3)` and `fib(2)` at the first level. For each pair, we check if there is an isolation between their subtrees, hence if these tasks are ordered and all descendant tasks of the first task are enforced to be finished before starting the other task. We start by looking if the two tasks are ordered. To do so, for each dependency domain, we build a *Directed Acyclic Graph* (e.g., DAG) representing the complete ordering of tasks, thanks to `depend` and `if` clauses, `taskwait` and `taskgroup` directives. Then detecting if two tasks are ordered in a dependency graph is equivalent to find a path between the two tasks in the DAG (l.6). If there is a path, then the two tasks are ordered.

Figure 2b depicts the DAG generated for the dependency domain formed by the leaf tasks in subtree B. Since the two `fib` invocations are independent, there is no link between them. However, two links come from these tasks towards the third (sum) task, due to the `depend(in)` expressed dependencies. Hence, an order exists between `fib(0)` and `(sum)`, and an order also exists between `fib(1)` and `(sum)`.

We then check if the first subtree is isolated. If so, it is useless to detect potential conflicts between these subtrees (l.7–9). In the example, if the subtrees from `fib(3)` and `fib(2)` are isolated, no data race can happen between `(sum)` and the subtrees. However, it will not prevent data races between the subtrees, as they can be executed in any order, and even concurrently.

If no isolation is detected, we need to compare every pair of tasks in the tested subtrees (l.11–12). We check each `depend` clause from the two tasks to detect potential conflicting memory access (l.13). Once all the current sibling tasks are tested, we do the same procedure with the next level in the spawn-tree.

4.2 Tracking Memory Scope

The OpenMP runtime only uses the address of memory storage to express the dependencies. When detecting conflict with addresses, two cases arise. First, the address always identifies the same variable throughout the program execution. It is the case for global variables. On the other hand, some addresses can be reused throughout the program to store different variables. It is the case for heap and stack addresses, through function calls and return statements or memory allocation/deallocation. To ensure that the detected conflict on addresses passed to `depend` clauses can actually lead to a data race, we have to ensure that the same address relates to the same variable.

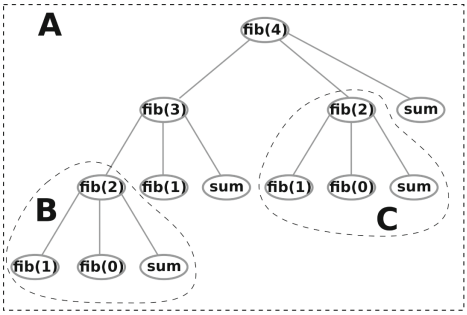
The fibonacci example can illustrate such behavior. With the dataflow based fibonacci algorithm, we can see that the same pattern of tasks may be replicated in the spawn-tree. It is the case for the subtrees **B** and **C** in Fig. 2a.

When the program is running, the following behavior can happen. First, a thread runs the task which is the root node of subtree **B**. This task declares dependencies on stack addresses for variables x, y and s for the children task. Upon completion of the task, stack memory is recycled for the next instructions. If the root node of subtree **C** is then scheduled on this same thread, as it is the same task as the root of subtree **B**, it will map the same variables to the same stack addresses. The executing task will also declares dependencies for variables x, y and s, which happen to have the exact same stack addresses than the dependencies declared for the previous task. However, they are not related, and the reuse of addresses only relates to this specific scheduling. It is necessary to check if the use of the same addresses in multiple `depend` clauses are indeed related to the same variables.

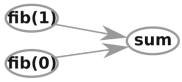
Data scoping is a key element in OpenMP, and more generally in shared memory programming models. It describes if a specific data is shared among threads or is local to a thread. By default, scope attribute is set to shared for threads and implicit tasks, and to firstprivate for explicit tasks. OpenMP provides clauses to modify the scope attribute of data: `shared` exhibits data's memory address to all threads, and `private`, `firstprivate` or `lastprivate` create a thread's local data copy (different memory address). The `firstprivate` clause is a special case, the value of the variable is passed on to the local copy. By this way, if the variable value is an address, it violates the private attribute since all threads having the local data copy can simultaneously access the same memory storage. If the variable is used later in a `depend` clause, it may lead to a concurrent access.

To ensure that the same address in multiple `depend` clauses relates to the same variable, we record the data scoping attributes at task creation. We then study the data scoping path, i.e. the variable's scope attribute at each level between a task and a child task of its lineage. A color c is associated with each tested pair of tasks and each tested address. For the two tested tasks, we go up in the spawn-tree and check at each level if the address is a shared data, or if it was a value passed in a firstprivate clause. In both cases, the link to the checked level is colored with the color c. Once a common ancestor for the two tasks is

found in the spawn tree, we obtain a direct path between the two tasks. If all the links in the path have the same color c, it means that the tested address was passed by the common ancestor down to the two tested tasks, and that the address relates to the same variable. Hence, the tested `depend` clauses may actually cause a data race, and the *DetectConflicts* phase in our algorithm raises an issue.



(a) Spawn-tree instance of fibonacci(4)



(b) DAG generated from the dependency domain formed by the leaf tasks in subtree B.

Fig. 2. Data structures related to the task-based Fibonacci example

To illustrate this coloring search, we focus on x variables from `fib(1)` invocations. To both tasks `fib(1)` from subtrees **B** and **C** for variable x, we use the color c_0. Since the variable is in a `shared` clause for both tasks, the links between these tasks and their parents (respectively roots of subtrees **B** and **C**) are colored with c_0. However, the variable passed in the `shared` clause is a newly created variable and does not come from a previous `shared` clause (or `firstprivate` clause). Hence, the upward links are colored with a new color c_1 (from root node of subtree **B** to `fib(3)` task, and from root node of subtree **C** to `fib(4)` task). For the same reason, the last link from `fib(3)` to `fib(4)` will have a new color c_2. Once this link is colored, we obtain a colored path between the two tested tasks. However the path has multiple colors, hence the two addresses don't relate to the same variable. No potential data race will be raised, even if the `depend` clauses use the same memory address.

4.3 Method Limitations

Our method uses the same information as the OpenMP dependencies mechanism, i.e, the memory address. We do not aim to detect nor instrument actual memory access, but only to check if the dependencies declared in the OpenMP task constructs are coherent. As we are based on the addresses passed in the dependency clauses, our method may miss some data races or report false positive.

The false positive arises when the task constructs declares dependencies on variables which are not used in the task, or in its descendant tasks. In these cases, our method returns a potential data race. However, since the variables are not used in the tasks, it is not a data race. These variables might just have been used to infer ordering between tasks with no actual read or write.

In the same way, if variables are used in a task but do not appear in a `depend` clause, our method will not consider the variable. The same also happens if a variable a variable is used in the task spawning a dependency domain, with tasks declaring dependencies on the same variable. In these cases, out method will not detect the potential data race.

```
1  #pragma omp task depend(in:a)
2    #pragma omp task depend(out:a) {
3      a = some_value; }
4    local = a;
```

Listing 1.5. Nested tasks with potential race conditions

The test case in Listing 1.5 presents such scenario. Based only on the dependency declarations, there is no way to detect when the actual memory access is performed in the parent task, i.e, before or after the child task.

5 Tool Implementation

Our detection method is based on the task spawn-tree and the *DAG* built from dependency clauses information. Building and maintaining such structures requires accessing information from the OpenMP directives and internals in addition to those provided by its API: e.g., when a parallel region starts and ends, when synchronizations occur at multiple levels, be informed of tasks creation and retrieve their dependencies set if any. These information are tightly linked to the OpenMP API and runtime implementation.

The OMPT [12] interface aimed at developing portable performance and analysis tools for OpenMP. Recently released as part of the OpenMP specification, it provides an instrumentation-like portable interface for tool callbacks. A callback is a function that is registered during the tool initialization to be triggered at corresponding events. In addition, OMPT specifies a collection of inquiry functions to probe the OpenMP runtime for internal information.

Our tool can either be used at runtime or post-mortem through the generation of a trace. Both versions use the same information that can be gathered through the set of OMPT callbacks listed below. Implementation has been done inside the MPC framework [15], a hybrid MPI/OpenMP runtime which support the OMPT interface. During the initialization phase, we create the internal structures and the root task of the spawn tree. Then, to instrument all OpenMP tasks in the application, the tool registers the following OMPT callbacks to the OpenMP runtime:

- ompt_callback_parallel_{begin/end}_t: callbacks to register the entry and exit points of parallel regions. We use the *begin* event to retrieve the number of threads inside the parallel team. During the *end* event, we deallocate all nodes

of the spawn tree, if any, and its related dependencies. Only the root task remains as the code returns to the initial task.

- ompt_callback_implicit_task_t: callback triggered during the creation of implicit tasks. We use it to add nodes representing the implicit tasks of the parallel region into the spawn tree.

- ompt_callback_sync_region_t: callback to register region synchronization. Its parameters contain the synchronization type (*i.e.* a barrier, a taskgroup or a taskwait) and the endpoint scope (*i.e.* the beginning or the end of the synchronization). We use it for partitioning the dependency domain at the explicit task level. In the runtime version, the main Algorithm 1 for data race detection is triggered. This allows to reduce memory consumption by only keeping and applying resolution on one instance of the spawn-tree at the time. In post-mortem version, trace generation is performed by dumping local buffers to output files.

- ompt_callback_task_create_t: callback to register the creation of an explicit task. We use this callback to add a task node to our internal spawn tree representation at the creation of an explicit task. Such informations are stored in local buffers in the post-mortem version.

- ompt_callback_dependences_t: callback to register all dependencies specified on a new task. We retrieve the dependencies of the newly created task, and update the dependency *DAG* of the parent task node. Such informations are stored in local buffers in the post-mortem version.

OMPT Extensions. Section 4 highlights that it is necessary to know the data sharing attribute of a dependence to detect data races in the context of nested tasks with dependencies. The current OMPT interface exposes the scope of a parallel region, the spawning sequence of tasks and the dependencies between these tasks. But it lacks a way to provide information about data-sharing attributes at constructs, needed in our method to detect false positives. To do so, we propose the following extensions to the OMPT interfaces.

```
1  typedef void (*ompt_callback_task_create_t) (
2    ompt_data_t *          encountering_task_data,
3    const ompt_frame_t *   encountering_task_frame,
4    ompt_data_t *          new_task_data,
5    int                    flags,
6    int                    has_dependences,
7    size_t                 array_data_attributes_size,
8    void *                 array_data_attributes
9    const void *           codeptr_ra,
10 );
```

Listing 1.6. Extension to ompt_callback_task_create_t

The data sharing attributes of each variable are retrieved at task creation. We extend the callback to also store an array with the data collection inherited from outer scope to the new task (see Listing 1.6). This array contains values for each variable: if a variable is shared, the array contains its address. If a variable is firstprivate, the array contains the variable value.

```
1  typedef struct ompt_dependence_s
2    ompt_data_t             variable;
3    ompt_dependence_type_t  dependence_type;
4    int                     address_location;
5  ) ompt_dependence_t;
```

Listing 1.7. Extension to ompt_task_dependence_t

The location of the address variable used in the `depend` clause is required to eliminate false positive in our data race detection method. We extended the structure exposed in Listing 1.7 to include an *int* value to store this location.

6 Experimental Results

An enumeration of available applications using nested tasks with dependencies lead to a small set of candidates. Upon *ad hoc* test cases based on those presented throughout the whole paper and the modified Fibonacci, the Kastors benchmarks suite [13] provided a suitable candidate. The *Strassen* benchmark is a well-known algorithm for matrix multiplication that achieves lower execution bound than the regular method $O(n^3)$. It recursively splits the matrices and applies the Strassen method in a *divide and conquer* manner, until a specified cutoff is reached where the regular method turns back to be more efficient. We present the output format of our tool and its associated overhead on these benchmarks.

Output Description. The generated output goes along with the approach described as follows: the nodes of the spawn tree are numbered in a breadth-first search manner, therefore, the root has the number 0, each implicit task has the number between [1, ..., N] where N is the number of threads participating to the parallel region, and the explicit tasks have a number between [N, ..., M] where M is the total number of nodes in the spawn tree. Two conflicting nodes n and n' respectively belonging to the subtree where nodes rn and rn' are the roots and with dependencies d and d' on a variable address *addr* generate the following output: $*$ *addr*$<rn, n, d><rn', n', d'>$

```
> OMP_TOOL_LIBRARIES=Ompt_tool.so OMP_NUM_THREADS=2 mpcrun ./testCase3
* 0x2b7730422e70 < 1, 3, in >< 2, 6, out >
* 0x2b7730422e70 < 1, 4, out >< 2, 5, in >
* 0x2b7730422e70 < 1, 4, out >< 2, 6, out >
```

The small example above is the output of our tool for Listing 1.3. The corresponding spawn-tree is depicted in Fig. 1a. Our tool returns three potential data races: task 3 with task 6, task 4 with task 5 and task 4 with task 6. Theses pairs of tasks have dependencies on the same variable, with at least one being a write, so the analysis is true. A data race is possible as implicit tasks may run concurrently. Task 3 and task 5 have both read dependencies on the same variable, hence no order is required. Hence no issue is raised for this pair.

We also applied the Archer and the Tasksanitizer tools on this example. Tasksanitizer correctly unveils a data race, whereas for Archer, the analysis being

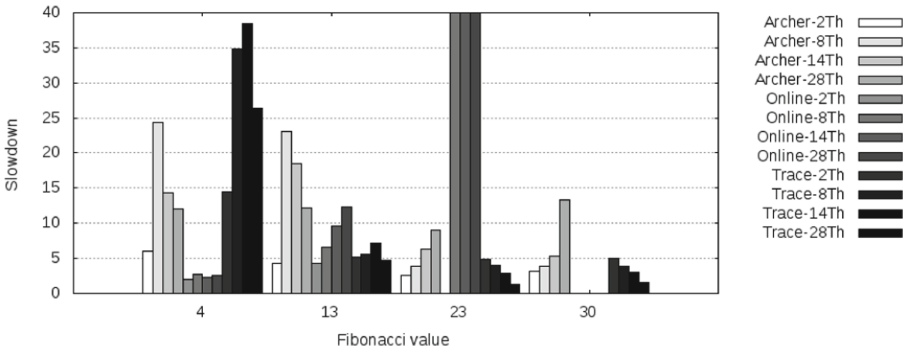


Fig. 3. Overhead for multiple fibonacci values and number of threads.

applied on the current execution scheduling, the data race is not detected on every run. Both tools, upon detection of a race condition, only retrieve a subset of possible cases/scheduling leading to the race condition. In our ad-hoc cases, Tasksanitizer did find many false positives, mainly arising from poorly support of *taskwait* and *dependencies* in nested tasks with dependencies context.

Study of Overhead. We evaluated our tool overhead on the Fibonacci and the Strassen benchmarks. The tests were conducted on an Intel XEON node with 28 physical cores and 186 GB of memory ram. In our results, we illustrate the slowdown factor (*i.e.* execution time with a tool divided by the time of the standard version of the code) for different tools: Archer, our tool with both online and post-mortem analysis. Tasksanitizer exhibited very high overhead for Fibonnacci (from one hundred to several thousands) and was segfaulting on Strassen, hence its results are not displayed.

Archer is more complete and performs more analyses than our tool. We use this time as an upper bound overhead to not overcome.

The evaluation of the modified Fibonacci was conduct on the Fibonacci values $fib(x)$, where $x \in \{4, 13, 23, 30\}$, representing respectively the creation of 12, 1128, 139101 and 4038804 tasks at runtime (see Fig. 3). For a small number of tasks, the online version is efficient, whereas the trace-based version is slower than Archer. This is due to our tracing mechanism which is very basic (no I/O delegation or asynchronism), and the cost of waiting to write the trace is too high regarding the benchmark execution time. For a large number of tasks the online version spends to much time checking each pair of tasks, and has prohibitive overhead. On the other hand, tracing becomes very competitive. The evaluation of the Strassen benchmark was conducted on square matrices with power of two sizes from 512 to 8192. Two cutoffs were set for the switching value to regular method and for the max depth, controlling the task nesting up to four levels. On Strassen (Fig. 4), overheads of both online and trace-based methods are lower than Archer. The slowdown is up to 7.7 for online resolution and a maximum of 4.6 for trace generation. With online resolution, only size 4096 provides high overheads. Further investigation is needed to understand these results.

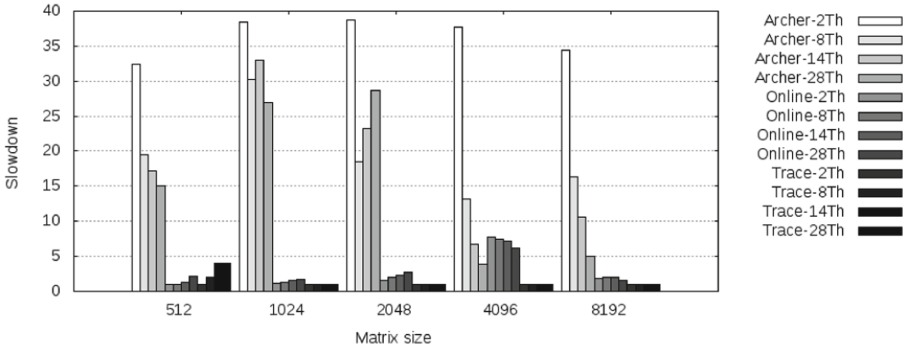


Fig. 4. Overhead for multiple Strassen matrix sizes and number of threads.

7 Conclusion

Since version 4.0, the OpenMP standard includes the notion of data dependencies between tasks created by the same parent (either another task or a thread). But combining nested tasks with data dependencies may lead to race conditions, some uncovering unexpressed/hidden dependencies. This paper proposed an algorithm to detect such problems based on the `depend` clauses exposed by the programmer. We implemented this method in a tool providing both dynamic and post mortem approaches, based on the recently released OMPT interface and our extensions for data sharing attributes. We demonstrated that this method can effectively detect race conditions with a reasonable slowdown compared to existing tools. The proposed OMPT extension for data sharing attributes can be useful for any tools relying on addresses passed in clauses. For future work, we plan to study a restricted use of code instrumentation to detect data accesses inside OpenMP tasks, and then be able to detect any data races between `depend` clauses and actual variable accesses.

Acknowledgments. This work was performed under the Exascale Computing Research collaboration, with the support of CEA, Intel and UVSQ.

References

1. OpenMP Architecture Review Board: OpenMP Application Program InterfaceVersion 5.0, November 2018
2. Pérez, J., Beltran, V., Labarta, J., Ayguadé, E.: Improving the integration of task nesting and dependencies in OpenMP. In: IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, pp. 809–818. IEEE Computer Society, Orlando, FL, USA (2017)
3. Virouleau, P., Roussel, A., Broquedis, F., Gautier, T., Rastello, F., Gratien, J.-M.: Description, implementation and evaluation of an affinity clause for task directives. In: Maruyama, N., de Supinski, B.R., Wahib, M. (eds.) IWOMP 2016. LNCS, vol. 9903, pp. 61–73. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45550-1_5

4. Rana, V.S., Lin M.: A scalable task parallelism approach for LU decomposition with multicore CPUs. In: Proceedings of Second International Workshop on Extreme Scale Programming Models and Middleware (ESPM2 2016), Salt Lake City, November 2016
5. Podobas, A., Brorsson, M., Vlassov, V.: TurboBLYSK: scheduling for improved data-driven task performance with fast dependency resolution. In: DeRose, L., de Supinski, B.R., Olivier, S.L., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2014. LNCS, vol. 8766, pp. 45–57. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11454-5_4
6. Ghane, M., Malik, A.M., Chapman, B., Qawasmeh, A.: False sharing detection in OpenMP applications using OMPT API. In: Terboven, C., de Supinski, B.R., Reble, P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2015. LNCS, vol. 9342, pp. 102–114. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24595-9_8
7. Agullo, E., Aumage, O., Bramas, B., Coulaud, O., Pitoiset, S.: Bridging the gap between OpenMP and task-based runtime systems for the fast multiple method. IEEE Trans. Parallel Distrib. Syst. (TPDS) **28**, 2794–2807 (2017)
8. Dinh, D., Harsha, S., Tang, Y.: Extending the nested parallel model to the nested dataflow model with provably efficient schedulers. In: Proceedings of the 28th Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, pp. 49–60. ACM, Asilomar State Beach/Pacific Grove, CA, USA (2016)
9. Protze, J., et al.: Towards providing low-overhead data race detection for large OpenMP applications. In: Proceedings of the 2014 LLVM Compiler Infrastructure in HPC, LLVM 2014, pp. 40–47. IEEE Computer Society, New Orleans, LA, USA (2014)
10. Protze, J., Hahnfeld, J., Ahn, D.H., Schulz, M., Müller, M.S.: OpenMP tools interface: synchronization information for data race detection. In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2017. LNCS, vol. 10468, pp. 249–265. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65578-9_17
11. Matar, H.S., Unat, D.: Runtime determinacy race detection for OpenMP tasks. In: Aldinucci, M., Padovani, L., Torquati, M. (eds.) Euro-Par 2018. LNCS, vol. 11014, pp. 31–45. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96983-1_3
12. Eichenberger, A.E., et al.: OMPT: an OpenMP tools application programming interface for performance analysis. In: Rendell, A.P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2013. LNCS, vol. 8122, pp. 171–185. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40698-0_13
13. Virouleau, P., et al.: Evaluation of OpenMP dependent tasks with the KASTORS benchmark suite. In: DeRose, L., de Supinski, B.R., Olivier, S.L., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2014. LNCS, vol. 8766, pp. 16–29. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11454-5_2
14. Duran, A., Teruel, X., Ferrer, R., Martorell Bofill, X., Ayguadé Parra, E.: Barcelona OpenMP tasks suite: a set of benchmarks targeting the exploitation of task parallelism in OpenMP. In: Proceedings of the International Conference on Parallel Processing (ICPP) (2009)
15. Carribault, P., Pérache, M., Jourdren, H.: Enabling low-overhead hybrid MPI/OpenMP parallelism with MPC. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., de Supinski, B.R. (eds.) IWOMP 2010. LNCS, vol. 6132, pp. 1–14. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13217-9_1