A toolchain to verify the parallelization of OmpSs-2 applications

Simone Economo[1,2][0000−0002−4796−956X], Sara Royuela[1][0000−0002−7644−0868], Eduard Ayguadé[1][0000−0002−5146−103X], and Vicenç Beltran[1][0000−0002−3580−9630]

[1] *Barcelona Supercomputing Center (BSC)*
{simone.economo,sara.royuela,eduard.ayguade,vbeltran}@bsc.es
[2] *DIAG Antonio Ruberti, Sapienza Università di Roma*

Abstract. Programming models for task-based parallelization based on compile-time directives are very effective at uncovering the parallelism available in HPC applications. Despite that, the process of correctly annotating complex applications is error-prone and may hinder the general adoption of these models. In this paper, we target the OmpSs-2 programming model and present a novel toolchain able to detect parallelization errors coming from non-compliant OmpSs-2 applications. Our toolchain verifies the compliance with the OmpSs-2 programming model using local task analysis to deal with each task separately, and structural induction to extend the analysis to the whole program. To improve the effectiveness of our tools, we also introduce some ad-hoc verification annotations, which can be used manually or automatically to disable the analysis of specific code regions. Experiments run on a sample of representative kernels and applications show that our toolchain can be successfully used to verify the parallelization of complex real-world applications.

Keywords: Synchronization · Software testing and debugging · Parallel programming.

1 Introduction

In the last twenty years, the conceptual hardware organization of computing systems has changed significantly. Complex multi-core and heterogeneous architectures are ubiquitous nowadays and represent a cost-effective way to support the high degree of parallelism of many High-Performance Computing (HPC) applications. Several new ideas have been put into the software in terms of parallel programming supports to adapt to this paradigm shift [16]. In order to implement parallelization via these supports, applications need to be redesigned or ported to a different programming language with parallelization constructs. In some cases, the user is also responsible for how the parallelism is implemented. A direct consequence of this is that the effort of maintaining the source code increases, and tasks like debugging or testing become quite tricky. Parallel programming models based on compiler directives such as OpenMP [2] are an alternative to

Economo, S. [et al.]. A toolchain to verify the parallelization of OmpSs-2 applications. A: International European Conference on Parallel and Distributed Computing. "Euro-Par 2020: Parallel Processing, 26th International Conference on Parallel and Distributed Computing: Warsaw, Poland, August 24–28, 2020: proceedings". Berlín: Springer, 2020, p. 18-33. ISBN 978-3-030-57675-2. DOI 10.1007/978-3-030-57675-2_2.
The final authenticated version is available online at https://doi.org/0.1007/978-3-030-57675-2_2.

the approaches mentioned above. These models allow the programmer to disclose parallelism within programs through source-code annotations, which are interpreted by the compiler as commands to perform transformations that parallelize the code. The annotation-based approach is very effective as it allows users to parallelize applications incrementally without sacrificing the programmability and portability of code. Starting from the sequential version of the application, the user can add more and more annotations to specify the parallelism of different parts of the application. Despite the high potential of annotation-based models, the parallelization process remains manual and prone to errors by the user. Incorrect usage of annotations can lead to performance and correctness issues and many hours of bug-hunting, thus forcing developers to debug their programs in conventional (and typically ineffective) ways to try to get to the root cause of the problem.

In this article, we focus on the OmpSs-2 task-based programming model. OmpSs is a shared-memory multiprocessing API developed at the Barcelona Supercomputing Center (BSC) for C, C++, and Fortran programs. OmpSs takes from OpenMP its idea of providing a way to, starting from a sequential program, produce a parallel version through `pragma` annotations in the source code. Parallelization is achieved by annotating certain code regions as tasks that can execute independently on the available threads, and synchronization constraints between them. OmpSs has also been a forerunner for many of the task-based features later introduced in OpenMP. The second version of OmpSs, called OmpSs-2, features a fine-grained data-flow execution model for tasks that has been recently proposed for integration into OpenMP [15]. The OmpSs programming model is interesting because it has clear rules when it comes to specifying tasks and synchronization constraints. For this reason, it is possible to verify that applications comply with it in a programmatic manner. Applications that are compliant to the OmpSs programming model are less likely to be affected by parallelization errors that undermine the performance and correctness of the program. Therefore, proving that an application complies with the rules of OmpSs eliminates some of the errors that can be introduced upon parallelizing the code of an application, thus potentially saving many hours of tedious debugging.

In this work, we illustrate a programmatic approach to checking parallelization errors in OmpSs-2 based on local task analysis to verify task-level compliance, and structural induction to verify application-level compliance. We also propose a novel toolchain that implements this analysis for real-world OmpSs-2 applications. The toolchain is based on a framework that involves three pieces: compile-time analysis to check the compliance of code before execution, run-time analysis to verify code that could not be checked at compile-time, and verification annotations to mark code that should not be explicitly analyzed by our toolchain. Our experiments suggest that our toolchain's hybrid nature is key to making our programmatic approach viable for checking the compliance of real-world applications.

2 Task-based parallelization in OmpSs-2

In this section, we describe the parallelization annotations available in the OmpSs-2 task-based programming model, and the rules that must be respected to comply with it. Failing to do so is a *compliance error*, denoted with the 'E' prefix, which may impact both the performance and the correctness of the parallelized application. We describe these errors in detail in the rest of this section.

2.1 Tasks and dependencies

OmpSs-2 allows expressing parallelism through *tasks*, independent pieces of code that can be executed by the computing resources at runtime. Whenever the program flow reaches a section of code declared as a task, the system creates an instance of that task and delegates its execution to the OmpSs-2 runtime system. Tasks are created via the `task` directive. Any directive that defines a task can also appear within the definition of a task, thus naturally supporting task nesting. Note that, in OmpSs-2, everything is a task. The user program runs in the context of an implicit task region, called the *initial task*. This makes all user-defined tasks to be nested tasks to that initial region.

OmpSs-2 tasks commonly require to access data to do meaningful computation. These data references can be declared via the `in`, `out`, or `inout` clauses[3]. The set of all data references constitutes the *dataset* of a task. Each time a new task is created, its dataset is matched against those of previously-created tasks to produce execution-order constraints between them. We call these constraints *dependencies*. This process creates a *task dependency graph* at runtime that guarantees a *correct* order of execution for the application, i.e., an order which respects the dependencies between tasks. Tasks aren't considered for execution until all their predecessors in the graph, if any, have finished.

Whether the task actually uses data in the declared way is the responsibility of the programmer. In Listing 1.1 it is an error (E1) to access `a` from inside T1, because `a` is not in the dataset of T1 and thus doesn't generate any dependency. If there is another task T2 accessing the same variable, the two tasks can't synchronize their accesses. Another error (E2) is declaring an element in the dataset that is not accessed. For example, if T2 declares to access `d` when the variable is not accessed, there may be undesired synchronization between T2 and another task T3 accessing the same variable.

2.2 Dependency domains

The OmpSs-2 model states that dependencies between any two tasks can be established if those tasks share the same *dependency domain*. By default, a task t can only have dependencies with its *sibling tasks*, i.e., tasks that share with

[3] For a thorough explanation of the admitted syntax for data references, see the official OmpSs-2 specification: `https://pm.bsc.es/ftp/ompss-2/doc/spec/`.

```
int a, b;
long c[N];

#pragma oss task in(b) out(c[i:j]) label(T1)
{
  a = 5; // Error E1: No matching dependency for 'a'
}
#pragma oss task inout(a, b, d) label(T2)
{
  a += b; // OK
  // Error E2: No matching access for 'd'
}
#pragma oss task in(d) label(T3)
{
  int x = d; // OK
}

```

Listing 1.1. Definition of tasks and dependencies.

```
#pragma oss task in(a) weakout(b) label(T1)
{
  int x = a; // OK
  #pragma oss task out(b, c) label(T1.1)
  { ... }
  // Error E3: No matching 'c' dependency in T1
}
#pragma oss task in(b) weakout(d) label(T2)
{
  int y = b; // OK
  // Error E4: No matching 'd' dependency in T2.1
  #pragma oss task out(b) label(T2.1)
  { ... }
}

```

Listing 1.2. Connecting tasks via weak dependencies.

t the same parent task. To connect two tasks that are not siblings, the dependency model in OmpSs-2 supports *weak dependencies*. These are created via the `weakin`, `weakout`, and `weakinout` clauses, but are not real dependencies. Their sole purpose is to inform the runtime that some descendant of a task is accessing the data elements specified in the `weak` variant. To connect the dependency domains of two arbitrary tasks t_1 and t_2, we must propagate the dataset of both t_1 and t_2 upwards, using the `weak` prefix, until we find a common ancestor t_a (which can coincide with t_1 or t_2). By doing this, the runtime will merge the dependency domain of all tasks from t_1 to t_a, and from t_2 to t_a, thus being able to establish a dependency between t_1 and t_2.

The mechanism of synchronization via weak dependencies can be unintuitive at times. In Listing 1.2, failing to weakly pass the reference to c from T1.1 upwards is an error (E3) because the model states that if dependency domains are not properly connected, accesses to the same object in different domains cannot be synchronized. Another error (E4) is to declare an object in the weak dataset of T2, when no descendant task is accessing it. Even if the runtime doesn't perform any actions on T2 that require the enforcement of those dependencies, it may suggest an error elsewhere, e.g., a missing `out` reference to d in task T2.1.

```
#pragma oss task label(T1)
{
  int x = 0, y = 2;
  #pragma oss task inout(x) label(T1.1)
  { ... }
  #pragma oss task in(x) inout(y) label(T1.2)
  { ... }
  #pragma oss taskwait in(x)
  assert(x == 1); // OK
  // Error E5: No 'taskwait' or 'taskwait in(x, y)' before the assertion
  assert(x == y);
}
```

Listing 1.3. Synchronization via the `taskwait` construct.

2.3 Taskwait synchronization

By design in OmpSs-2, to synchronize the code of task t with any of its descendants t' we need to use the `taskwait` directive. Taskwait synchronization means that the runtime waits until the previously-created descendant tasks (including the non-direct children tasks) complete their execution. The set of sibling tasks targeted by a taskwait depends on the data references added to the `taskwait` directive. If no data references are specified, the taskwait blocks the task waiting for the completion of all previous descendant tasks.

Appropriately placing taskwaits in task code is a process prone to mistakes in OmpSs-2 applications. In Listing 1.3, failing to place a `taskwait` before the last assert is an error (E5) because the parent task is allowed to execute the statement without waiting for its children (which access both x and y) to finish.

3 Programmatically checking compliance

Our programmatic approach verifies application-level compliance through task-level compliance analysis and inductive reasoning on the recursive structure of OmpSs-2 applications. The former is used to verify the absence of errors in each task separately; the second is used to verify increasing portions of the program until we reach the initial entry point. These two techniques rely respectively on two aspects of the OmpSs-2 model: (1) compliance errors in a task t can be verified without having to look at the internal code of other tasks, nor at the datasets of tasks at nesting levels that cannot be directly reached from t; (2) the code of the program can be represented as a hierarchy of tasks, with the initial task wrapping the initial entry point. Any task-based programming model satisfying these properties admits a programmatic approach for checking compliance like the one described in this section.

3.1 Task-level compliance

Table 1 provides a compact list of the errors that were discussed in Section 2. To check that a task is free of these errors, OmpSs-2 states that we only look at

E	Description
$E1$	No matching dependency for an access
$E2$	No matching access for a dependency
$E3$	No matching dependency in the parent task
$E4$	No matching dependency in the child task
$E5$	No taskwait between an access and previous tasks

Table 1. Compliance errors in OmpSs-2

what happens (i) within the code of the task itself, (ii) in the dataset annotations of its parent (if any), and (iii) in the dataset annotations of its children (if any). This fact is exploited in our tools to analyze each task separately. We call *local task analysis* (LTA) the kind of processing we carry out to check that a single task is compliant with the OmpSs-2 model. It is local because such analysis does not need to reason globally, i.e., at the level of the whole program. To understand how local task analysis works, let's consider a task t in the program. Let t_p be its parent task, and t_c be a child task. Let $d_{(t,i)}$ be the i-th dataset element of t, defined as a tuple $\langle m, clk, r\rangle$, where $m \in \{read, write\}$ is the access mode, clk is the time at which the corresponding task was created, and r is the memory range of that entry. Let $a_{(t,j)}$ be the j-th memory access performed by t, defined as a tuple $\langle m, clk, r\rangle$ with m being once again the access mode, clk being the time at which the access was performed, and r the memory range of the access. Let $\mathcal{D}_t$ be the set of all dependencies of t. Let $\mathcal{A}_t$ be the sequence of all accesses of t (also called the *access-set* of t). Finally, let $\mathcal{W}_t$ be the set of taskwaits dependencies inside task t. Each entry $w_{(t,k)}$ is a tuple $\langle m, clk, r\rangle$, where clk is the time at which the corresponding taskwait was created, and m and r are defined like their counterparts in $d_{(t,i)}$. In the following, we show a conceptual description of LTA, focusing on the errors E1, E3, and E5 for the sake of simplicity. LTA for the remaining cases can be defined likewise.

Condition 1 (E1 detection). *Verify if there is at least one access performed by t that does not have a corresponding dataset entry. Formally speaking, check if, for each $a_{(t,j)} \in \mathcal{A}_t$, there is no $d_{(t,i)} \in \mathcal{D}_t$ for which:*

- $a_{(t,j)}.r \subseteq d_{(t,i)}.r$, *and*
- $d_{(t,i)}.m = a_{(t,j)}.m$

If there is any $a_{(t,j)}$ for which it is true, then t is affected by E1.

Condition 2 (E3 detection). *Verify if there is at least one dataset entry of t (weak or not) that does not have a corresponding dataset entry in its parent (at least weak). Formally speaking, check if, for each $d_{(t,i)} \in \mathcal{D}_t$, there is no $d_{(t_p,i_p)} \in \mathcal{D}_{t_p}$ for which:*

- $d_{(t,i)}.r \subseteq d_{(t_p,i_p)}.r$, *and*
- $d_{(t,i)}.m = d_{(t_p,i_p)}.m$

If there is any $d_{(t,i)}$ for which it is true, then t is affected by $E3$.

Condition 3 (E5 detection). *Verify if there is at least one access performed by t such that: (i) the access has a corresponding dataset entry in one of the previously-created child tasks; (ii) at least one amongst the access and the dataset entry is a write; (iii) the access is not guarded by a taskwait that blocks until the termination of the conflicting child task. Formally speaking, check if, for each $a_{(t,j)} \in \mathcal{A}_t$, there is at least one $d_{(t_c,i_c)} \in \bigcup_{t_c} \mathcal{D}_{t_c}$ for which:*

- $a_{(t,j)}.r \subseteq d_{(t_c,i_c)}.r$, *and*
- $a_{(t,j)}.m = write$, *or* $d_{(t_c,i_c)}.m = write$, *or both, and*
- $a_{(t,j)}.clk > d_{(t_c,i_c)}.clk$, *and*
- *there is no* $w_{(t,k)} \in \mathcal{W}_t$ *for which:*
  - $w_{(t,k)}.clk < a_{(t,j)}.clk$, *and*
  - $w_{(t,k)}.clk > d_{(t_c,i_c)}.clk$, *and*
  - $w_{(t,k)}.r \cap d_{(t_c,i_c)}.r \neq \varnothing$, *and*
  - $w_{(t,k)}.m = write$, *or* $d_{(t_c,i_c)}.m = write$, *or both.*

If there is any $a_{(t,j)}$ for which it is true, then t is affected by $E5$.

Conditions 1 to 3 give us a way to detect the errors in Table 1. However, to make these conditions operational, we need to convert them into an algorithm, and the mathematical structures on which such conditions rely must be turned into concrete data structures. Section 4 briefly describes an experimental implementation of LTA based on compile-time and run-time analysis.

3.2 Application-level compliance

Local task analysis is used in our approach to check that a task is free of compliance errors. However, we need a way to prove that the entire application is also free of these errors. To do this, we reason inductively on the task-nested structure of OmpSs-2 applications. The OmpSs-2 model represents a program as a hierarchy of tasks. It states that no parts of the program can be executed outside of a task. The recursive nature of tasks can be exploited to prove application-level compliance using *structural induction*, which is a generalization of the inductive proof technique over natural numbers. The property that we wish to prove inductively is *OmpSs-2 compliance*, defined as follows:

Definition (OmpSs-2 compliance). *A task t is OmpSs-2 compliant if and only if the following condition holds: (1) the task is not affected by any of the errors in Table 1, and (2) for every task t' that is a child of t, t' is also OmpSs-2 compliant.*

By using LTA and structural induction on the nested task structure of an OmpSs-2 application, it is possible to prove its compliance in an incremental manner. According to the definition of OmpSs-2 compliance, if the initial task is OmpSs-2 compliant, then the whole application is compliant.

```
#pragma oss lint in(sendbuf[0:size]) out(recvbuf[0:size])
{
  MPI_Send(sendbuf, size, MPI_BYTE, dst, block_id+10, MPI_COMM_WORLD);
  MPI_Recv(recvbuf, size, MPI_BYTE, src, block_id+10, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
}

double A[N/TS][M/TS][TS][TS];
#pragma oss lint out (A[i][j])
for (long ii = 0; ii < TS; ii++)
  for (long jj = 0; jj < TS; jj++)
    A[i][j][ii][jj] = value ;

for (int i = 0; i < N; ++i) {
  #pragma oss task verified(i != 0 && i != N-1 && i % M != 0)
  { ... }
}
```

Listing 1.4. Examples of the `lint` directive and the `verified` clause.

3.3 Capabilities of the programmatic approach

In Section 2, we introduced the notion of compliance error and explained that it might affect the parallelization of an application in an undesired way. Generally speaking, we call *parallelization error* any error that was introduced upon parallelizing the original sequential program, and that affects the parallelized program's behavior in an unintended way. In this article, we are concerned with two main types of parallelization errors: performance and correctness errors. *Performance errors* can create additional synchronization constraints that defer the execution of a task unnecessarily. *Correctness errors* are typically caused by an unintended lack of synchronization between tasks that alters the original sequential program' semantics. Parallelization errors can be hard to spot and to debug. Usually, they don't manifest predictably, as it depends on the relative timing between the interfering tasks. Nevertheless, it can be shown that the absence of compliance errors is a sufficient condition for the absence of specific parallelization errors [7], such as those described in this article. However, it is worth observing that not all compliance errors produce parallelization errors. There are cases in which the application doesn't comply with the model, but the synchronization between tasks doesn't produce correctness or performance errors at run-time. Viceversa, not all parallelization errors that may negatively affect the application are compliance errors that can be detected with this approach. Some parallelization errors are *semantics errors*, i.e., errors that require a knowledge of the semantics of the application to be detected programmatically. These errors are out of the scope of this work. Lastly, limitations coming from concrete LTA implementations (such as those mentioned in Section 4) may too affect the accuracy of the analysis.

4 An OmpSs-2 verification toolchain

This section describes our novel toolchain for checking the compliance of OmpSs-2 applications[4]. It is made of three key elements: (1) a static source-code analyzer that works at compile-time (also called the *compile-time tool*); (2) a dynamic binary-code analyzer that works at run-time (aka the *run-time tool*); (3) a set of `pragma` directives and clauses (also called *verification annotations*) that can be used as an interface between the user, the compile-time tool, and the run-time tool. The reason behind this hybrid architecture is to overcome some limitations of both compile-time and run-time analysis that might undermine the effectiveness of the programmatic approach described in Section 3.

4.1 Manual user pass

Initially, users can annotate portions of code that must be ignored by our toolchain. To this extent, we have introduced support for ad-hoc verification annotations into the Mercurium source-to-source compiler [8]. They instruct the compile-time and run-time tools to pause the analysis inside the wrapped code region. The verification annotations we introduced in OmpSs-2 are: (1) the `lint` directive, followed by optional `in`, `out`, or `inout` data-references; and (2) the `verified` clause, optional in the `task` construct.

The `lint` directive can be used to ignore code inside tasks. To extend its applicability, users can also declare which accesses to shared-memory (if any) performed within the ignored region are relevant for LTA. For the compile-time tool, the directive is especially useful to mark calls to inaccessible code. In the first example of Listing 1.4, the `MPI_Send` and `MPI_Recv` functions are not available for analysis, but their semantics is clear: they respectively read/write N bytes from/to memory. For the run-time tool, marking code is useful to prevent tracing memory accesses that, albeit executed inside tasks, don't relate to the application business logic. This scenario includes, amongst many, accesses performed in libraries to shared-memory variables that are not visible to the application, as well as accesses to shared-memory objects that are synchronized independently of OmpSs-2 (e.g., spinlocks). In the MPI example, the implementation of `MPI_Send` and `MPI_Recv` may perform accesses to some internal variables used for synchronization purposes, hence not relevant for LTA. The `verified` clause works at the level of whole tasks. It is used to tell both tools that the task is OmpSs-2 compliant, and that no LTA is needed. It accepts an optional boolean expression to decide, at run-time, whether that particular task instance has to be verified. This expression can be used to conditionally evaluate task instances

[4] Compared to the reference description in Section 2, our tools support additional OmpSs-2 features: `commutative` and `concurrent` dependencies (treated like `inout`), explicit release of dependencies, `final` and `if` clauses. Primitives for task reductions, atomic operations, and critical regions are currently unsupported. Additional information, included the instructions on how to install and use the toolchain, can be found here: `https://github.com/bsc-pm/ompss-2-linter`.

that are more likely to be subject to programming errors (e.g., tasks related to boundary loop iterations). It can also be used to implement task-level sampling and reduce the overall memory tracing overhead of the application (e.g., instrument a fraction of all task instances at run-time). In the third example of Listing 1.4, we only instrument a subset of the tasks that represent distinct loop iterations: the first task, the last task, and one every M of the remaining ones.

4.2 Compile-time pass

The compile-time tool aims at two main goals. The first goal is to anticipate errors that are independent of the input of the application and may later appear at run-time. To this extent, we have extended Mercurium and its built-in infrastructure for static analysis with an LTA implementation, evaluating task-level compliance for every task definition in the source code. Notice that compile-time LTA cannot always derive the full program state at every point in the code. Additionally, it cannot analyze code that is unavailable at compile-time (e.g., code coming from other compilation units, or code that is dynamically loaded). When lacking information, it doesn't state anything about OmpSs-2 compliance and leaves task-level analysis to the run-time tool. The run-time tool circumvents these limitations, but only for specific input and while introducing overhead during the execution of the application. For this reason, to ease the burden of the run-time tool, the second goal of the compile-time tool is to mark those sections of code that have been verified by the compiler and therefore do not need to be instrumented at run-time. In the second example of Listing 1.4, a nested loop structure is used to perform a linear array walk. The compile-time tool can detect this scenario and can mark it with a verification annotation. The TS^2 accesses performed within the loop are ignored by the run-time tool, but an equivalent representation of these accesses is placed in the annotation so as to be considered at run-time.

The algorithm to place verification annotations around portions of code, or whole task definitions, performs a bottom-up/inside-out traversal over the *Parallel Control Flow Graph* (PCFG) [17]. It uses induction variables and scalar evolution analysis in an attempt to wrap adjacent statements incrementally until a terminating condition is encountered (e.g., a call to a function whose code is not reachable). The compile-time tool also makes use of the manually-placed verification annotations to try to extend their scopes to more extensive code regions. At the end of this pass, any detected error is reported to the user before execution. The parts of the code that could be verified statically are marked using verification annotations, while the others are left for run-time instrumentation.

4.3 Run-time pass

The run-time tool is invoked to complement the compile-time analysis and to provide complete coverage of the code, but only for a given input. Run-time analysis can observe the actual program execution state at any moment in time, so it doesn't need to be conservative. However, it has other limitations. It cannot

always distinguish memory accesses that are relevant for LTA (e.g., accesses to shared-memory variables visible to the application) from non-relevant ones (e.g., access to shared-memory variables private to a library and synchronized separately). Additionally, the instrumentation introduced at run-time for the sake of tracing can alter the timing of some events, thus leading to observe artificial and slower application executions.

In order to circumvent such accuracy and overhead issues, run-time analysis exploits verification annotations placed by the user or by the compile-time tool, and only runs LTA for code that lacks such annotations. The tool operates at two different levels of abstraction: (1) the abstraction provided by the OmpSs-2 programming model to deal with tasks and dependencies, as explained in Section 2; (2) the abstraction provided by the target Instruction Set Architecture (ISA) to recognize accesses to memory, which in our case is AMD64[5]. Our run-time instrumentation tool is based on Intel Pin [12] and is composed of three main components: the Pin Virtual Machine (VM) to perform dynamic binary instrumentation, and two modules that perform memory access tracing on the binary executable. The *frontend* module (or *trace generator*) is devoted to intercepting the accesses performed by the application at run-time, as well as generating the actual traces. The *backend* module (or *trace processor*) is responsible for the processing of traces and the generation of the final report for the user. At the end of this pass, the tool generates a report of the encountered errors for that specific application execution, thus complementing the report produced at compile-time.

5 Experimental assessment

In this section we provide an experimental evaluation of the analysis overhead[6] of our toolchain on a set of nine different benchmarks, made of five execution kernels (*matmul*, *dot-product*, *multisaxpy*, *mergesort*, and *cholesky*) and four proxy application (*nqueens*, *nbody*, *heat*, and *HPCCG*). These benchmarks are representative of real-world scientific applications and use popular HPC libraries for advanced mathematical operation (such as Intel MKL) as well as well-known APIs for coarse-grained parallelism (i.e., MPI). Our objective is to demonstrate that our toolchain can be effectively used to evaluate the task-based parallelization of these applications.

All the experiments have been conducted on the MareNostrum4 supercomputer. Each compute node is equipped with two 24-core Intel Xeon Platinum 8160 CPUs, totaling 48 cores per node, and 96 GB of main memory. The interconnection network is based on 100 Gbit/s Intel OmniPath HFI technology. The MPI benchmarks (*nbody*, *heat*, and *HPCCG*) are run on four different nodes, while the other benchmarks are run on a single node. Figure 1 shows the

[5] Although our tool targets the AMD64 instruction set, this does not limit the scope of our work as it can be easily ported to other ISA and processor models.

[6] A comprehensive evaluation of the accuracy of our toolchain will be provided in a subsequent study.

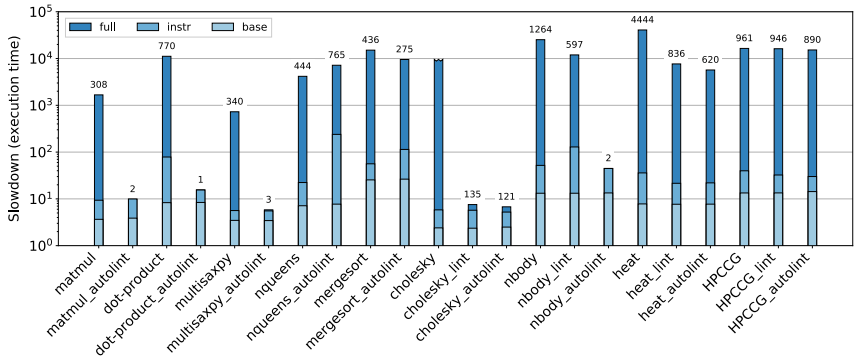

Fig. 1. Slowdown (y-axis) and absolute execution time (numbers on top of bars, in seconds) for the selected benchmarks.

slowdown (y-axis) and the absolute execution time (numbers on top of bars, in seconds) for running the selected benchmarks through the run-time tool. Each bar represents a different benchmark and a different set of experiments. The `_lint` suffix represents the case of running the benchmark without the aid of the compile-time tool, but using the `lint` directive to manually annotate calls to third-party libraries. The `_autolint` suffix represents the case of running the benchmark with the aid of the compile-time tool, which places additional `lint` directives (if possible) around regions of verified code. The absence of a suffix means that the benchmark is run without the aid of the compile-time tool or `lint` directives. For each bar, we also report a breakdown of the slowdown, split into three different contributions: (a) the instrumentation cost to run the application using Pin (the `base` label in the legend); (b) the instrumentation cost to actually instrument memory instructions, without processing them (the `instr` label); (c) the full instrumentation + processing cost (the `full` label in the legend).

As we can see from the figure, the slowdown for the pure runtime instrumentation case (no suffix) can be quite high for some benchmarks (e.g., *dot-product* or *mergesort*). In the case of *cholesky*, the overhead is considerably high due to the heavy use it makes of Intel's MKL library. It is reported with a truncated bar and no number on top because it exceeded the maximum time allocation for a single job (two days). We conducted an extended analysis of these cases and detected the major source of overhead to be the insertion of accesses in an ad-hoc interval tree, used to aggregate contiguous accesses coming from the same instruction over time and compare them with task dependencies. Although we intend to develop a more efficient implementation for this data structure, we are still bound to pay the instrumentation cost depicted in the `base` and `instr` cases. Nevertheless, we think that the observed slowdown doesn't limit the effectiveness of our tools. Except for *cholesky*, we note that the absolute execution time of all the instrumented benchmarks is in the order of minutes, thus not

undermining the toolchain's usability. Moreover, using larger input sizes is often unnecessary. In many task-based HPC applications (which are well-represented by the benchmarks we use), a change in the input size typically has a considerable impact on *how many* tasks are executed, rather than *which* types of tasks. Even when it substantially modifies the control flow at run-time (e.g., by activating different tasks, or code paths inside tasks), these variations could have been stimulated already with smaller input sizes.

In all those cases in which it is necessary to test an application with large or production-level inputs, we can exploit the `lint` and `verified` annotations to focus the analysis only on the specific code activated by those inputs. This approach makes our toolchain more effective because it allows us to spare the tracing overhead on the parts that could be tested with smaller inputs. In our experiments, the improvements in terms of the slowdown in the `lint` case were often significant. By appropriately marking calls to external libraries with verification annotations, the run-time instrumentation tool only intercepts a number of accesses that are proportional to the number of data-references specified in the `in`, `out`, or `inout` parameters of the pragma itself. This aspect is critical for the case of *cholesky*, as each task only performs a single call to a function in the MKL library, but those calls internally perform a huge number of accesses to memory that are the main source of overhead. Improvements can also be observed for the case of MPI benchmarks, which use the Intel MPI library, although the impact tends to be smaller than that observed in the previous benchmarks. For example, while *heat* is communication-intensive and so protecting calls to MPI is highly effective, *nbody* and *HPCCG* are computation-intensive. Therefore, the use of pragmas doesn't improve the execution overhead by much.

The `autolint` case brings the most evident benefits, as it can be seen for *matmul*, *dot-product*, and *multisaxpy*. In this case, the compile-time tool can automatically wrap whole for-loop cycles into pragmas, or even mark whole tasks within loops as `verified`. In all these cases, the performance improvements are drastic because the instrumentation tool can disable tracing during most of the application's execution time. We note that these improvements are not uncommon for real-world scenarios, as many kernels have a regular loop structure, which can be easily analyzed using techniques like those mentioned in Section 4.2. The case of *nqueens* is peculiar because it internally uses recursion. In this case, the compile-tool is unable to recognize this execution pattern and ends up marking each memory-accessing statement independently. The net effect of this is a deterioration of the run-time overhead, compared to when the compile-time tool is disabled. Similar considerations can be made for the MPI benchmarks and especially for *HPCCG*, where the main kernel performing an MKL-like *dgemm* operation couldn't be annotated at all because a sparse matrix representation is internally used. As for *cholesky*, we observe that each task only performs a single call to an MKL library function. Thus, the compile-time tool can successfully promote the manual `lint` directives to `verified` clauses at the level of tasks. However, this brings little additional benefits compared to the `lint` case.

Overall, our experimental evaluation suggests that the absolute execution cost of running the selected applications against the toolchain is affordable. Furthermore, the synergistic exploitation of compile-time analysis and verification annotations can drastically reduce this cost.

6 Related work

The strategies for verifying the parallelization of applications can be classified in static tools, which analyze the code at compile-time, and dynamic tools, which analyze the code at run-time. As for the fork-join part of OpenMP, there are static solutions focused on the polyhedral model to detect errors in OpenMP parallel loops [5], or on symbolic analysis and Satisfiability Modulo Theories (SMT) to detect data races and deadlocks [13]. A more general solution is provided by Lin [11], who described a control flow graph and a region tree to statically detect non-concurrent blocks of code and race conditions in OpenMP2.5 programs with the Sun Studio 9 Fortran compiler. Techniques to detect synchronization issues in task-based OpenMP programs also exist and are focused on race conditions that may produce non-deterministic output and run-time failures [17]. In concurrent models based on tasking such as Ada, there have been efforts to introduce model checking techniques at compile-time [1]. However, although these techniques are very mature, their usefulness depends on contracts that are written by programmers, hence are liable to have errors. For the dynamic detection of parallelization errors, most of the literature is focused on tools that check for data and determinacy races, using the Happens-Before (HB) relation to detect if two memory accesses are concurrent [18, 10]. Archer [3] adapts ThreadSanitizer, which can detect data races in unstructured parallel programs, to the case of basic OpenMP tasking with no dependencies. It employs a static phase to discard all sequential code, and a dynamic phase to check for data races in the remaining concurrent parts. Sword [4] is a tool that is capable of detecting all and only data races in OpenMP programs comprised of nested fork-join parallelism (i.e., `parallel` constructs). TaskSanitizer [14] is a tool that detects determinacy races in task-parallel OpenMP programs by computing the HB relation on tasks. ROMP is another tool targeting OpenMP with tasking [9]. It uses an approach close to Sword to build the HB relation for nested fork-join parallelism parts, and one similar to TaskSanitizer for the HB relation of tasks with dependencies. StarSscheck [6] is a run-time tool to detect parallelization errors commonly occurring in StarSs applications (task dependencies without nesting).

Our approach significantly differs from the ones adopted by the above works. First of all, we don't explicitly check for correctness errors. Our tools look for compliance errors, which may affect both correctness and performance. The detection of such errors is based on a programmatic approach that is compatible with the OmpSs-2 programming model, but that can be ported to all task-based programming models satisfying the properties in Section 3. To this extent, our analysis is also different. Being always local to a task, it only compares accesses and data references of a task with other data references. In comparison, algo-

rithms built around the HB relation directly compare accesses from a task with accesses from another task, thus having to perform a number of comparisons that, in principle, can be quite higher than LTA. Lastly, to improve the overall accuracy and overhead of detection, our toolchain combines the best of static and dynamic techniques with the proposal of verification annotations, which are used as an abstract interface between the user, the compile-time tool, and the run-time tool.

7 Conclusions and future work

We have presented a toolchain to detect parallelization errors in applications using OmpSs-2, a task-based parallel programming model. Our toolchain is composed of a compile-time tool that analyzes source code, and a run-time tool that analyzes binary code. The outcome of our toolchain is a report which informs the user about compliance errors of OmpSs-2 applications. Our tools only perform local task analysis of code, i.e., independently for each task. Because of the way the OmpSs-2 programming model is defined, we can evaluate the compliance with the model for each task and then infer it for the whole program. We have also introduced verification annotations to mark specific code regions as verified. Our compile-time and run-time analysis tools can safely ignore the code inside these regions. At the same time, they can also be informed about any relevant access performed within verified code regions. Thanks to these annotations, we can improve both the performance and accuracy of the analysis. Experiments run on a series of benchmarks varying from simple execution kernels to real-world applications suggest that our tools can effectively analyze a wide range of applications with acceptable overhead. Future work is aimed at improving our analysis to detect inefficient parallelization constructs and suggesting the use of more efficient ones.

Acknowledgements. This project is supported by the European Union's Horizon 2021 research and innovation programme under grant agreement No 754304 (DEEP-EST), by the European Union's Horizon 2020 research and innovation programme under grant agreement No 871669 (AMPERE) and the Project HPC-EUROPA3 (INFRAIA-2016-1-730897), by the Ministry of Economy of Spain through the Severo Ochoa Center of Excellence Program (SEV-2015-0493), by the Spanish Ministry of Science and Innovation (contract TIN2015-65316-P), and by the Generalitat de Catalunya (2017-SGR-1481).

References

1. AdaCore, Altran, Astrium Space Transportation, CEA-LIST, ProVal at INRIA and Thales Communications: Project Hi-Lite: GNATprove. http://www.open-do.org/projects/hi-lite/gnatprove (2017)
2. Arb: OpenMP specification v5.0. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf (2018)

3. Atzeni, S. and Gopalakrishnan, G. and Rakamaric, Z. and Ahn, D. H. and Laguna, I. and Schulz, M. and Lee, G. L. and Protze, J. and Mller, M. S.: ARCHER: Effectively Spotting Data Races in Large OpenMP Applications. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 53–62 (2016)
4. Atzeni, S. and Gopalakrishnan, G. and Rakamaric, Z. and Laguna, I. and Lee, G. L. and Ahn, D. H.: SWORD: A Bounded Memory-Overhead Detector of OpenMP Data Races in Production Runs. In: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 845–854 (2018)
5. Basupalli, V. and Yuki, T. and Rajopadhye, S. and Morvan, A. and Derrien, S. and Quinton, P. and Wonnacott, D.: ompVerify: Polyhedral Analysis for the OpenMP Programmer. In: OpenMP in the Petascale Era. pp. 37–53. Springer Berlin Heidelberg (2011)
6. Carpenter, P. M. and Ramirez, A. and Ayguadé, E.: Starsscheck: A Tool to Find Errors in Task-Based Parallel Programs. In: Euro-Par 2010 - Parallel Processing. pp. 2–13. Springer Berlin Heidelberg (2010)
7. Economo, S.: Techniques and tools for program tracing and analysis with applications to parallel programming. Ph.D. thesis, Sapienza Università di Roma (2020)
8. Ferrer, R. and Royuela, S. and Caballero, D. and Duran, A. and Martorell, X. and Ayguadé, E.: Mercurium: Design decisions for a s2s compiler
9. Gu, Y. and Mellor-Crummey, J.: Dynamic Data Race Detection for OpenMP Programs. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis. SC 18, IEEE Press (2018)
10. Jannesari, A. and Bao, K. and Pankratius, V. and Tichy, W. F.: Helgrind+: An efficient dynamic race detector. In: 2009 IEEE International Symposium on Parallel Distributed Processing. pp. 1–13 (May 2009)
11. Lin, Y.: Static Nonconcurrency Analysis of OpenMP Programs. In: OpenMP Shared Memory Parallel Programming, pp. 36–50. Springer Berlin Heidelberg (2008)
12. Luk, C.-K. and Cohn, R. and Muth, R. and Patil, H. and Klauser, A. and Lowney, G. and Wallace, S. and Reddi, V. J. and Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation p. 190200 (2005)
13. Ma, H. and Diersen, S. R. and Wang, L. and Liao, C. and Quinlan, D. and Yang, Z.: Symbolic Analysis of Concurrency Errors in OpenMP Programs. In: 2013 42nd International Conference on Parallel Processing. pp. 510–516 (2013)
14. Matar, H. S. and Unat, D.: Runtime Determinacy Race Detection for OpenMP Tasks. In: Euro-Par 2018: Parallel Processing. pp. 31–45. Springer International Publishing (2018)
15. Perez, J. M. and Beltran, V. and Labarta, J. and Ayguadé, E.: Improving the Integration of Task Nesting and Dependencies in OpenMP. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 809–818 (2017)
16. Reinders, J.: Intel threading building blocks - outfitting C++ for multi-core processor parallelism (1 2007)
17. Royuela, S. and Ferrer, R. and Caballero, D. and Martorell, X.: Compiler analysis for OpenMP tasks correctness. In: Computing Frontiers. Acm (2015)
18. Serebryany, K. and Iskhodzhanov, T.: ThreadSanitizer: Data Race Detection in Practice. In: Proceedings of the Workshop on Binary Instrumentation and Applications. p. 6271. WBIA 09, Association for Computing Machinery (2009)