© 2020 Springer. The final version can be found at:
https://link.springer.com/chapter/10.1007/978-3-030-59851-8_2
https://doi.org/10.1007/978-3-030-59851-8_2

Static analysis to enhance programmability and performance in OmpSs-2

Adrian Munera[1], Sara Royuela[1], Roger Ferrer[1], Raul Peñacoba[1], and Eduardo Quiñones[1]

Barcelona Supercomputing Center
{adrian.munera,sara.royuela,roger.ferrer,raul.penacoba, eduardo.quinones}@bsc.es

Abstract. Task-based parallel programming models based on compiler directives have proved their effectiveness at describing parallelism in High-Performance Computing (HPC) applications. Recent studies show that cutting-edge Real-Time applications, such as those for unmanned vehicles, can successfully exploit these models. In this scenario, OpenMP is a de facto standard for HPC, and is being studied for Real-Time systems due to its time-predictability and delimited functional safety. However, changes in OpenMP take time to be standardized because it sweeps along a large community. OmpSs, instead, is a task-based model for fast-prototyping that has been a forerunner of OpenMP since its inception. OmpSs-2, its successor, aims at the same goal, and defines several features that can be introduced in future versions of OpenMP. This work targets compiler-based optimizations to enhance the programmability and performance of OmpSs-2. Regarding the former, we present an algorithm to determine the data-sharing attributes of OmpSs-2 tasks. Regarding the latter, we introduce a new algorithm to automatically release OmpSs-2 task dependencies before a task has completed. This work evaluates both algorithms in a set of well-known benchmarks, and discusses their applicability to the current and future specifications of OpenMP.

Keywords: Programmability · Performance · Static Analysis · OmpSs-2

1 Introduction

The growing demands of society regarding mobility, health, and industry, among others, have motivated the convergence of computer systems towards complex ecosystems. This affects dissimilar domains such as High-Performance Computing (HPC) and Real-Time systems, e.g., supercomputers combining shared memory computational nodes in a distributed memory environment targeting HPC, and Multi-Processing Systems-on-Chip (MPSoC), containing multiple and heterogeneous processing elements, targeting Real-Time applications.

The need to exploit complex architectures effectively and efficiently has promoted the use of parallel programming models. As a result, several abstractions focusing on the *productivity* of parallel systems coexist [24]. A possible classification divides them in two groups: (1) *thread-based* models, which exploit data

parallelism (i.e., distributing data), and (2) *task-based* models, which exploit task parallelism (i.e., distributing units of work, or *tasks*). The former force programmers to manage low-level details of the computation, such as data distribution and synchronization, while the latter typically offer a higher-level abstraction that simplifies these complexities. Task-based models also offer greater flexibility, making them more suitable for the parallelization of dynamic and unstructured applications. For these reasons, these have gained a broad acceptance.

Some representative task-based models are Intel Threading Building Blocks [27], CUDA graphs [23], OpenMP [13] and OmpSs [7]. The two former are *hardware-centric* models that expose the architectural features in the language, requiring programmers a considerable level of expertise to achieve productivity, while also preventing portability. The two latter are *parallelism-centric* approaches offering high-level APIs based on compiler directives that hide the complexities of the architecture, and focus on providing mechanisms to describe parallel units and the synchronizations among them. The simplicity of the latter, together with their productivity, makes them very appealing for programmers.

Among all parallel abstractions, OpenMP has become the de facto standard for shared-memory HPC [6] by virtue of its productivity [10,28], while it also supports heterogeneous computation through the acceleration model [2,12]. Moreover, it has an increasing interest in embedded computing [22,20] because of its delimited functional safety [18] and its proven time-predictability [21].

Introducing changes in OpenMP is a long-distance race that requires consensus of a large community, and prototype implementations of the involved vendors (e.g., Intel, IBM, and NVIDIA). Interestingly, OmpSs is a programming model, implemented on top of the Mercurium compiler [3] and the Nanos++ runtime [4]. The main goal of OmpSs has been fast-prototyping tasking features to include them in the OpenMP standard. Some of them, like task dependencies, task priorities, task loops, task reductions, taskwait dependencies, multi-dependencies and data affinity, are already included in OpenMP.

OmpSs-2 [5] is the second generation of OmpSs. It extends its predecessor with features covering dependencies across different task nesting levels and early release of dependencies, among others (see details in §3). The reason-to-be of OmpSs-2 is, as in OmpSs, to extend OpenMP with new features. For this reason, the novelties that OmpSs-2 introduces to tasking models, and the proved success of OmpSs to influence the OpenMP standard, motivates the interest of this work, which contributions are: (1) an algorithm to detect the correct data-sharing of OmpSs-2 tasks, targeting programmability and correctness, (2) an algorithm for the automatic release of task dependencies, targeting performance, and (3) an evaluation of both algorithms with LLVM in a set of benchmarks.

The remainder of the paper is organized as follows: §3 introduces OmpSs-2; §2 exposes the related work; §4 describes the proposed algorithms targeting performance, correctness and programmability in OmpSs-2 programs; §5 details of the implementation done in LLVM, and shows the evaluation of the proposed techniques; and §6 discusses the benefits of this work for OpenMP.

2 Related Work

Programmability is an important aspect of high-level programming models, as it is crucial for their adoption. Focusing on OpenMP, the scope of variables is a cumbersome and error-prone process that jeopardizes not only programmability, but also correctness. As a consequence, several works tackle the automatic scope of variables in OpenMP. Lin et al. [11] proposed the use of the `default(AUTO)` clause for such a purpose, and defined a set of rules to accomplish auto-scope in parallel regions. They showed that automatic and user-defined clauses can obtain the same performance. Voss et al. [25] evaluated the impact of the same clause for auto-parallelization purposes, concluding that several regions cannot be automatically parallelized because of the limitations of the technique: it only works for the OpenMP *thread model*, it offers limited inter-procedural analysis for arrays, and it lacks support for API functions calls. Later, Royuela et al. [17] proposed an algorithm to automatically scope variables in OpenMP 3.1 task regions. This work evaluates the algorithm using, among others, the BOTS benchmarks [8] and compares its results with those of the Oracle Solaris Studio 12.3 compiler mechanism with the same goal [15]. They exhibit an accuracy close to 85% compared to that of Solaris, close to 78%. Wang et al. [26] approached the same problem from a simplicity point of view, proposing an algorithm that uses `taskwait` directives to avoid the need for analyzing concurrency among tasks. Although this work presents better accuracy, it has an important negative impact on the performance of the resulting parallel code.

3 The OmpSs-2 Programming Model

OmpSs-2[5] is a task-based parallel programming model built on top of a set of C/C++ and Fortran language directives and a runtime API. OmpSs-2 is similar to OpenMP[13] in which a sequential program is incrementally parallelized using annotations in the source code. In contrast, it is purely task-based, so the fundamental unit of concurrency, to exploit parallel execution, is the so-called *task*. Listing 1.1 shows the syntax of an OmpSs-2 task directive in C/C++.

```
1  int x = 3, y = 4;
2  #pragma oss task \
3            shared(x) \
4            firstprivate(y) \
5            label(T)
6  {
7    x++;
8    y++;
9  }
10 #pragma oss taskwait
11 assert(x == 4);
12 assert(y == 4);
```

Listing 1.1: Basic task syntax.

```
1  #pragma oss task label(A)
2  {
3    #pragma oss task label(A1)
4    {}
5  }
6  #pragma oss task label(B)
7  {
8    #pragma oss task label(B1)
9    {}
10   #pragma oss taskwait
11   // B1 is complete
12   // A and A1 may not be complete
13 }
14 #pragma oss taskwait
15 // A, A1, B, and B1 are complete
```

Listing 1.2: Synchronization and nesting.

When a thread of the program encounters a `task` construct, it *creates* a task. The execution of the structured block (in C/C++, this is the compound statement that follows the `#pragma`) of the task, is deferred until the task itself is *executed*. A task that has been created but not yet executed is called to be *ready*. A task is *complete* when it has finished its execution.

3.1 Data-sharings

Variables used in the structured block that are declared inside the task region are local to the task, while those declared outside have an associated *data-sharing*. A *shared* data-sharing means that the task will capture the address of the variable, i.e. it will access the original variable. A *firstprivate* data-sharing means that the task will capture the value of the variable, i.e. it will access a copy of the variable. Shared variables are prone to data-races between the thread that executes the task and other threads, including the one that created the task.

3.2 Task Synchronization and Nesting

Tasks can be nestes in OmpSs-2: the execution of the enclosing task construct leads to the creation of the inner tasks, whose execution is also deferred. Parent tasks do not execute a taskwait at the end of their associated construct, so the enclosing task construct may complete before the nested tasks constructs do.

Synchronization is achieved using the `taskwait` directive, which waits for all tasks created in the current task context and those created in nested task contexts to complete, as shown in Listing 1.2. A `wait` clause attached to a `task` directive behaves as if a `taskwait` is inserted at the end of the task construct.

Applications may expose concurrency difficult to exploit due its dynamic nature. For these cases OmpSs-2 provides *dependency* clauses, like `in`, `out` or `inout`. When a thread of the program encounters a task construct, the runtime annotates the expressions denoting the memory referenced in those clauses. A task becomes ready when executing it would not violate its data dependencies respect to data dependencies of previous tasks created and not completed yet.

OmpSs-2 provides two more dependency clauses, `concurrent` and `commutative`, that are useful when several tasks participate in a reduction operation. The former is like `inout`, but allows parallelism across tasks with a concurrent dependency of the same object. The latter also acts as `inout`, but allows any ordering between tasks with the same commutative dependency. Unlike commutative, which is mutually exclusive, concurrent dependencies require explicit synchronization (e.g. locks) to avoid data races. Furthermore, dependency clauses can also be used in a taskwait directive, acting as a task with an empty construct.

Code in Listing 1.3 shows a program that creates 4 tasks. Tasks `T1` and `T2` can be executed concurrently. Task `T3` will be ready once `T1` completes. Task `T4` will be ready once `T1` and `T2` complete. Task `T5.1` will be completed after the taskwait with dependencies, `W1`. That taskwait would not wait for task `T5.2`.

OmpSs-2 considers a unique domain for all the tasks of a program: tasks define the data they use as the regular dependency clauses, and the data used

```
1 int x, y, z1, z2;
2 #pragma oss task out(x) label(T1)
3 { x = 1; }
4 #pragma oss task out(y) label(T2)
5 { y = 1; }
6 #pragma oss task inout(x) label(T3)
7 {
8   assert(x == 1);
9   x++;
10 }
11 #pragma oss task in(x) in(y) label(T4)
12 {
13   assert(x == 2);
14   assert(y == 1);
15 }
16 #pragma oss taskwait
17 assert(x == 2);
18 assert(y == 1);
19 #pragma oss task out(z1) label(T5.1)
20 { z1 = 3; }
21 #pragma oss task out(z2) label(T5.2)
22 { z2 = 4; }
23 #pragma oss taskwait in(z1) // (W1)
24 assert(z1 == 3);
```

Listing 1.3: Tasks and dependencies.

```
1 int x = 1;
2 #pragma oss task weakinout(x) \
3                  label(A)
4 {
5   #pragma oss task inout(x) \
6                    label(A1)
7   {
8     assert(x == 1);
9     x++;
10   }
11 }
12 #pragma oss task weakinout(x) \
13                  label(B)
14 {
15 #pragma oss task inout(x) label(B1)
16   {
17     assert(x == 2);
18     x++;
19   }
20 }
21 #pragma oss taskwait
22 assert(x == 3);
```

Listing 1.4: Nested tasks and dependencies.

in nested tasks as a weaker form of dependency designated by the `weakin`, `weakout` and `weakinout` clauses. This links the dependency domains at all nesting levels while avoids unnecessary synchronizations between tasks.

Listing 1.4 shows an example of dependency between nested tasks `A1` and `B1` without synchronizing tasks `A` and `B`. Had the code used a regular `inout` dependency in tasks `A` and `B`, then task `B` would only be ready once `A` completes.

A created task becomes ready when all its dependencies have been *released* by its task predecessor set. Tasks release all their dependencies when they complete. OmpSs-2 allows executing tasks to early release their dependencies using the `release` directive. This provides fine-grained control for tasks that describe a large set of data-dependencies that are processed in chunks.

Listing 1.5 shows an example in which a task, `T1`, processes data in chunks. Such process is fast, so it may not be beneficial to create a task per chunk. Then, a slower process creates one task `T2` per chunk. Without the release of dependencies, all `T2` tasks would have to wait to the completion of `T1`. Because `T1` releases dependencies earlier, `T2` tasks may be ready even before `T1` completes.

```
1 #pragma oss task out(data[0; size]) label(T1)
2 { // Task with an out dependency to all data[k], where 0 <= k < size
3   for (int i = 0; i < size; i += chunk_size) {
4     for (int j = i; j < chunk_size; j++)
5       fast_process(&data[j]);
6     #pragma oss release inout(data[i; chunk_size])
7     // The current task releases the inout dependency on all data[k],
8     // where i <= k < i + chunk_size
9   }
10 }
11 for (int i = 0; i < size; i += chunk_size) {
12   #pragma oss task in(data[i; chunk_size]) label(T2)
13   { // Task with an input dependency to all data[k],
14     // where i <= k < i + chunk_size
15     slow_process(&data[i], chunk_size);
16   }
17 }
```

Listing 1.5: Early release of dependencies.

4 Compiler Analysis Techniques for OmpSs-2

This section introduces two compiler analysis aiming at enhancing the programmability and performance of OmpSs-2 programs: (1) the automatic scope of variables in `task` constructs, and (2) the automatic release of task dependencies. Both techniques assume the input code is correct (e.g., dependencies are defined correctly), and that it keeps the sequential consistency property.

4.1 Automatic Definition of Task Data-sharing Clauses, *Auto-scope*

The mechanism for automatically scoping variables in OmpSs-2 tasks we propose draws from a previous algorithm proposed for the same purpose in OpenMP 3.1 tasks [17]. That algorithm proceeds, for each task t, in two steps: (1) define the regions of code that can be concurrent with t, and (2) scope the variables based on their usage within t and its concurrent regions, and the liveness of the variables after the last point at which the task can be synchronized.

We have adapted the previously mentioned algorithm to the singularities of OmpSs-2, including (1) *nested tasks* and *weak dependencies*, (2) the `concurrent` and `commutative` dependency clauses, and (3) `taskwait` directives with dependency clauses. These features, missing in OpenMP 3.1 (some of them are included in OpenMP v5.0, as discussed in §6), have an impact in the first step of the algorithm, i.e., computing the regions of code concurrent with a task. These parts can be (a) other tasks, (b) portions of the parent task, and (c) different instances of the same task. Next we describe the first step of the *auto-scope* algorithm to adapt it to OmpSs-2:

1. Compute the points of the code delimiting the regions code that can be concurrent with a task t, which are:
   - *TSP(t)*, the task scheduling point (TSP) of the creation of t.
   - *Pre-sync(t)*, the last synchronization(s) before *TSP(t)*, which can be (a) `taskwait` directives, (b) the end of other `task` constructs with matching dependencies, and (c) the beginning of the function enclosing t or the task scheduling point of the creation of the parent task.
   - *Post-sync(t)*, the first synchronization(s) encountered after *TSP* where the task can be synchronized, which can be (a) `taskwait` directives, (b) the beginning of other `task` constructs with matching dependencies, and (c) the end of the function enclosing t or the end of the parent task.

   The pseudo-code that computes the *Pre-sync(t)* points is shown in Algorithm 1. The algorithm for computing the *Post-sync(t)* points is analogous to the one for the *Pre-sync(t)* points, but it traverses the paths starting from the successors of t, instead of searching in the predecessors.
2. Compute the regions of code that can be concurrent with t, which are:
   - Other tasks found between *Pre-sync* , and *Post-sync*, that are not synchronized yet by means of dependency clauses.
   - Different instances of t, if the task is within a loop, and *Post-sync* happens after the end of the loop.

- Code from the parent task found:
  - Between *TSP* and *Post-sync*.
  - Between the beginning of the loop and *TSP*, if t is enclosed within a loop and *Post-sync* occurs after the loop has ended.

Algorithm 1: Computation of the *Pre-sync(t)* set of a task t.

```
Data: Node containing task t
Result: Pre-sync(t): list of last synchronization points
foreach path(p): possible path from t to the creation of its parent task, t', or
 the beginning of the enclosing function, f, whatever comes first do
    list deps(p);                          /* dependencies found in p */
    node n = path_deps(p).leaf;            /* node being traversed */
    while n != NULL do
        if n represents a taskwait directive without dependencies then
            Pre-sync(t).insert(n);
            if n dominates t then
                return Pre-sync(t)
            else
                break;                     /* stop searching this path */
        else if (n represents a task construct
                || n represents a taskwait directive with dependencies)
              && n synchronizes^1 t then
            Pre-sync(t).insert(n);
            deps(p).insert(deps(n));
            if deps(p) == deps(t) then
                break;                     /* stop searching this path */
        n = parent(n);                     /* keep traversing ancestors */
```

Fig. 1 illustrates the points delimiting the concurrent regions of a task. There, Fig. 1a shows an OmpSs-2 sample code where a task `T1`: (1) creates task `T2`, which sets the value of y; (2) creates task `T3`, which updates the value of y; (3) updates the value of y; and (4) waits for the completion of tasks producing y. Additionally, Fig.1b depicts a flow graph of the code, including the code in all tasks, the task scheduling points corresponding to the creation of the tasks (TxC), and the `taskwait`, as well as the *TSP*, *Pre-sync* and *Post-sync* points of task `T3`, represented with different symbols.

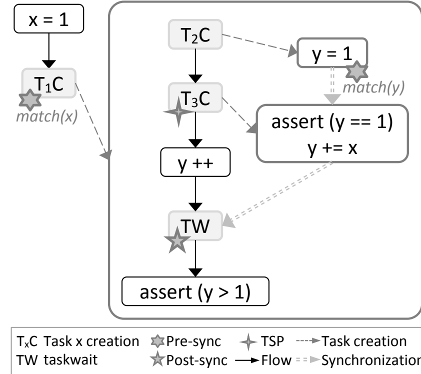
The rules that define the data-sharing attributes of a task remain the same as those defined in the original algorithm for OpenMP [17].

The adequate determination of the data-sharing attributes enhances the programmability of the model by freeing the programmer from the burden of manually defining these values. It also impacts correctness, for the algorithm can be used to check user-defined data-sharing attributes. Finally, it can also affect performance if variables unnecessarily privatized can be scoped as shared [19].

[1] A task t_2 synchronizes[19] a task t_1 if t_2 is created after t_1, and either (a) t_1 designates an `out` object that t_2 designates as `in` or `out`, or (b) t_1 designates an `in` object that t_2 designates as `out`, or (c) t_1 and t_2 designate the same `commutative` object.

```
1 int x = 1;
2 #pragma oss task weakin(x) \
3                  out(y) label(T1)
4 {
5   #pragma oss task out(y) label(T2)
6   { y = 1; }
7   #pragma oss task inout(y) in(x) \
8                    label(T3)
9   { // here, y might be 1 or 2
10    assert(x == 1);
11    y += x;
12  }
13  y ++;
14  #pragma oss taskwait in(y)
15  assert(y > 1);
16 }
```

(a) Code sample.



(b) Concurrent regions' limits for T3.

Fig. 1: Example of concurrent regions in the OmpSs-2 *auto-scope* algorithm.

4.2 Automatic Release of Task Dependencies, *Auto-release*

OmpSs-2 allows releasing the dependencies of a task before the task has finished (see §3 for details). This feature may positively impact the performance of the application when a task uses certain data only at the beginning, and then performs other lengthy operations that delay the release of this data dependencies.

In order to enhance the performance of OmpSs-2 programs while reducing the work needed by the programmer, we present next a compiler analysis that automatically introduces `release` directives within tasks. The algorithm works, for each task t in the program with a set of variables included in dependency clauses, *dep_vars(t)*, as follows:

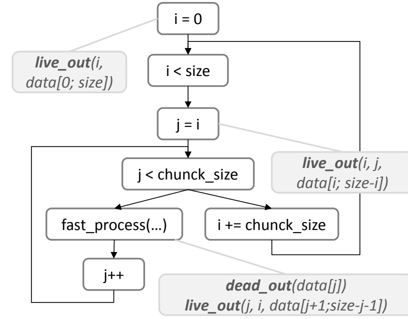
1. Compute liveness analysis[16] within t.
2. Traverse the statements of t in post-order and, for each statement s:
   (a) Gather the set of variables that are not live, *dead_vars(s)*.
   (b) Compute the set of variables to be released, *vars_to_release(s)*, as the intersection of *dead_vars(s)* and *dep_vars(t)*.
   (c) If *vars_to_release(s)* is not empty, introduce a `release` directive after s, including the variables in *vars_to_release(s)* as dependencies, as they were in the dependency clauses of t.
3. To reduce overhead, simplify the release directives as follows:
   (a) Remove the release directives if they are the last statement of the task, because the dependencies are going to be released at that point anyway.
   (b) Move release clauses outside loops to join the contiguous memory accesses of arrays and structures.

Figure 2 is an illustration of the liveness analysis. There, Figure 2a shows the portion of the code in Listing 1.5 corresponding to task `T1`, and Figure 2b shows a control-flow graph representation of that task with some nodes tagged with the information of liveness analysis. Overall, the analysis computes $live_out(n)$, where n is a node of the control-flow graph, as the set of variables that can

be used in any path reachable after n, taking into account the data-sharing attributes of nested tasks (i.e., if a variable is `private` in a nested task, then the original variable is not used at this point.) For example, after the call to $fast_process$, the value of $data[j]$ will never be used again, so this variable is dead. On the contrary, at the same point, the values of i and j are still live because the are used to update the next value of the same variables, respectively.

```
1  #pragma oss task out(data[0; size])
2  for (int i = 0; i < size;
3       i += chunk_size) {
4     for (int j = i;
5          j < chunk_size; j++) {
6        // in the first iteration
7        // data[i;chunk_size] is used
8        fast_process(&data[j]);
9        // data[j] is never used again
10       // i and j are used in the
11       // respective loop increments
12    }
13 }
```

(a) Snippet of code in Listing 1.5.



(b) Control flow and liveness analysis.

Fig. 2: Example of liveness analysis.

5 Evaluation

The algorithms described in §4 have been implemented in the LLVM compiler. Next subsections introduce some relevant details of the implementation, and show the evaluation of the two algorithms considering programmability (in the case of *auto-scope*) and performance (in the case of the *auto-release*).

5.1 LLVM Implementation

We have built the proposed techniques in LLVM, using a preliminary implementation of the OmpSs-2 model in Clang[2]. LLVM offers a stable and extensive tool-chain that includes several analysis, which simplifies the implementation task, while boosts the accuracy of the results. Particularly, we benefit from the following analysis already implemented: *dominator tree* (the LLVM DominatorTree class), used for detecting variable uses inside tasks and concurrent regions; *alias analysis* (the AAResults class), used to decide when different pointers and array accesses (may) point to the same memory location; and *scalar evolution* and other *loop analyses* (the ScalarEvolution, LoopInfo and IVUsers classes), used to recognize loops, analyze induction variables and recognize access patterns to arrays. Furthermore, we use two transformations required for the previous analysis to work: *loop-rotate*, which converts loops into do/while style loops,

[2] The artifact with the LLVM tool-chain with the proposed algorithms, the Nanos runtime library, and the test-suit used for the evaluation is publicly available in https://gitlab.bsc.es/ppc-bsc/research/c3po-artifact/-/tags/v1.0. A stable version of Clang for OmpSs-2 and Nanos will be released in the next months.

and *mem2reg*, which removes unnecessary *alloca* instructions. Finally, we take advantage of the *llvm-link* tool, which allows for linking several LLVM IR files into one, enhancing the possibilities for inter-procedural analysis, and hence the accuracy of our results.

Besides our algorithms, we have implemented *liveness* analysis in LLVM. This analysis uses the Value class, which gives information about the uses of a variable, while traverses the control-flow graph to search the paths that are reachable from a given node.

The implementation of the *auto-release* algorithm lacks its very last step, corresponding to the promotion of releases outside a loop when the individual releases inside the loop access contiguous memory. This remains as future work.

5.2 Benefits in OmpSs-2 Programmability: *Auto-scope*

To evaluate the enhancement in programmability of the auto-scope algorithm, we compute the number of variables automatically scoped for a series of benchmarks adapted to OmpSs-2 from the Barcelona OpenMP Task Suite [8], and other OmpSs-2 benchmarks[3]. Table 1 shows the results. There, each row refers to a benchmark; the first block of columns describe relevant aspects of the benchmarks; and the second block shows the number of variables automatically scoped for each attribute; and the success ratio as the number of variables automatically scoped out of the number of variables used in the tasks of the program.

	Description			LLVM results				
	#tasks	nested tasks	method	shared	private	firstprivate	undefined	(%)success
Alignment	1	no	iter	2	4	14	0	100%
FFT	41	no	rec	102	0	140	0	100%
Fib	2	no	rec	2	0	2	0	100%
Health	2	yes	iter&rec	1	1	2	0	100%
Floorplan	1	no	iter&rec	3	1	9	2	86.66%
NQueens	1	no	iter&rec	2	0	4	0	100%
Sort	9	yes	rec	27	0	10	0	100%
SparseLU	4	yes	iter	4	3	11	0	100%
UTS	2	no	iter&rec	2	1	3	0	100%
Cholesky	4	no	iter	4	0	12	0	100%
Saxpy	2	yes	iter	4	0	3	0	100%
Matmul	2	yes	iter	3	0	8	0	100%
TOTAL								**98.88%**

Table 1: Results of the *auto-scope* algorithm for OmpSs-2, implemented in LLVM, using different benchmarks.

The results reveal the strengths of the algorithm combined with the capabilities of LLVM. For eleven out of twelve benchmarks we have been able to automatically scope all variables. Only for *floorplan*, there are two variables that have not been automatically scoped. This is because these variables are only used in functions which code is not reachable (i.e. *memcpy*), and hence their usage cannot be determined (although many benchmarks use functions from standard C libraries, if the variables are also written within the function, then what happens within those calls can be omitted; this is not the case when the

[3] See footnote 2

variables are only read). Also composed variables (i.e. arrays) can be undefined if the *alias analysis* can not ensure there is no data race in any of the elements of the array, and it is used after the *Post-sync* points of the task.

The available mechanisms for the auto-scope of variables are not directly comparable with our technique because they target OpenMP, and do not support OmpSs-2 features. Nonetheless, a similar set of benchmarks was used to evaluate the algorithm we draw from [17], and we have improved the accuracy of the algorithm by virtue of the LLVM capabilities: the former offered an accuracy around the 85%, while our implementation is close to 100%. This, without losing performance, since we use the same rules for deciding the scope. Compared to other approaches [26] more conservative, we can advance that our approach will always perform equally or better because we do not use full-barriers (e.g., `taskwait` directives) to ease the algorithm, as they do.

5.3 Benefits in OmpSs-2 Performance: *Auto-release*

To evaluate the benefits in performance obtained with the *auto-release* algorithm, we have used two different configurations of the code shown in Listing 1.5: (1) the first one, called *Super fast process*, where *chunk_size* is set to 10000, *size* is set to 200000, and the call to *fast_process* takes 100μs; and (2) the second one, called *Fast process*, where *chunk_size* is set to 1000, *size* is set to 20000, and the call to *fast_process* takes 1000μs. For both configurations, the call to *slow_process* takes 5s. For this evaluation, we have indeed removed the `release` directive from the original benchmark. Furthermore, we have evaluated the overhead of introducing a release call, which is around 200ns.

Table 2 shows the execution time of the two configurations mentioned before, on an 8-core x86 2.60GHz Intel processor, for three different versions: (1) original user's code without releasing any dependency; (2) user's code with the *auto-release* algorithm introducing individual releases inside the loop; and (3) user's code with manual release of a full array section. The numbers show that, when the execution time of the code where the release is introduced is fast enough, i.e., *Super fast process*, the benefits of automatically adding the `release` directive are not as good as they could be. This is due to the overhead of the release call, together with the need for modifying the regions a task depends on very often (i.e., runtime overhead). Nonetheless, there is still a significant benefit compared to not releasing any dependencies. On the other hand, when the execution time of the code where the `release` directive is not that short, i.e., *Fast process*, then there is not a significant difference between inserting individual releases or joining contiguous releases in a unique `release` directive.

	Execution time (us)	
	Super fast process	**Fast process**
No release	46747405	36233344
Release within loop	40607605	26570543
Release outside loop	35894357	26533354

Table 2: Results of the *auto-release* algorithm for OmpSs-2, implemented in LLVM, using different configurations of code in Listing 1.5.

Although the evaluation shows that the auto-release of dependencies does not achieve better performance than the manual version, it reveals two important aspects: (1) the algorithm enhances the programmability of OmpSs-2 by relieving the user from the need to release data dependencies manually; and (2) the performance gain obtained with the automatic pass is obvious compared to not using our technique. The fact that we do not obtain the same results as perfectly releasing dependencies manually is because the implementation lacks the last step: promoting release directives outside loops. However, the use of this algorithm allows for existing kernels to benefit from the mechanism without needing any modification in the source code.

6 Discussion

The algorithms presented in this work target the OmpSs-2 parallel programming model. Nonetheless, some features introduced in OmpSs-2 already exist in the OpenMP 5.0 specification. On the contrary, OpenMP 5.0 and the research lines being conducted for future specifications also include some features to be considered in these algorithms. Next we discuss all these features:

- The `commutative` clause in OmpSs-2 has been introduced in the OpenMP 5.0 specification as `mutexinoutset`. Both clauses has the same behavior. This feature affects the proposed auto-scope algorithm when determining the tasks that can be concurrent with a given task.
- The `concurrent` clause in OmpSs-2 has not yet been introduced in OpenMP. However, the first preview for the future OpenMP specification 5.1 [14] includes the `inoutset` clause, which has the same behavior. This clause affects the computation of concurrent tasks in the auto-scope algorithm.
- The `release` directive does not exist in OpenMP, so its behavior can not yet be applied. However, there are other models, such as DepSpawn [9], that also include this feature and could benefit from our analysis.
- OmpSs-2 forces parent tasks to cover the dependencies of children tasks with either regular dependencies or weak dependencies to fulfill compliance. This boosts the safety of OmpSs-2. For this reason, this restriction could be applied to OpenMP if this model is to be used in critical real-time systems.

Overall, this paper introduces a series of compiler analysis for OmpSs-2 that enhance its performance while improving its programmability. Furthermore, we have shown how other programming models, specially OpenMP, could benefit from this analysis. This work tackles the automatic scope of variables in task constructs and the automatic release of task dependencies once these variables are already computed. There are however other features that remain as a future work, including the automatic definition of task, taskwait and taskloop dependencies, and the analysis of *detached tasks*, i.e., tasks that detach the finalization of task from the completion of the code included in the task, while attach it to the occurrence of a given event. This feature, accomplished by means of the `detach` clause in OpenMP, exists also in other models like StarPU [1] (by means of the *starpu_task_end_dep_release* call), but is not included in OmpSs-2.

References

1. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience **23**(2), 187–198 (2011)
2. Ayguadé, E., Badia, R.M., Bellens, P., Cabrera, D., Duran, A., Ferrer, R., González, M., Igual, F., Jiménez-González, D., Labarta, J., et al.: Extending OpenMP to survive the heterogeneous multi-core era. International Journal of Parallel Programming **38**(5-6), 440–459 (2010)
3. Barcelona Supercomputing Center: Mercurium. URL https://pm.bsc.es/mcxx
4. Barcelona Supercomputing Center: Nanos++. URL https://pm.bsc.es/nanox
5. Barcelona Supercomputing Center: Ompss-2. URL https://pm.bsc.es/ompss-2
6. Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. IEEE computational science and engineering **5**(1), 46–55 (1998)
7. Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: OmpSs: A proposal for programming heterogeneous multi-core architectures. Parallel processing letters **21**(02), 173–193 (2011)
8. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguade, E.: Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In: International Conference on Parallel Processing. pp. 124–131 (2009)
9. González, C.H., Fraguela, B.B.: A framework for argument-based task synchronization with automatic detection of dependencies. Parallel Computing **39**(9) (2013)
10. Kegel, P., Schellmann, M., Gorlatch, S.: Using OpenMP vs. Threading Building Blocks for Medical Imaging on Multi-Cores. In: Europar. pp. 654–665 (2009)
11. Lin, Y., Terboven, C., an Mey, D., Copty, N.: Automatic scoping of variables in parallel regions of an OpenMP program. In: International Workshop on OpenMP Applications and Tools. pp. 83–97. Springer (2004)
12. Martineau, M., McIntosh-Smith, S., Gaudin, W.: Evaluating OpenMP 4.0's effectiveness as a heterogeneous parallel programming model. In: International Parallel and Distributed Processing Symposium Workshops. pp. 338–347 (2016)
13. OpenMP ARB: OpenMP Application Program Interface, version 5.0 (2018)
14. OpenMP ARB: OpenMP Technical Report 8: version 5.1 preview (2019)
15. Oracle: Oracle Solaris Studio 12.2: OpenMP API User's Guide (2010), http://docs.oracle.com/cd/E18659_01/html/821-1381/toc.html
16. Royuela, S.: High-level Compiler Analysis for OpenMP. Ph.D. thesis (2018)
17. Royuela, S., Duran, A., Liao, C., Quinlan, D.J.: Auto-scoping for OpenMP tasks. In: International Workshop on OpenMP. pp. 29–43. Springer (2012)
18. Royuela, S., Duran, A., Serrano, M.A., Quiñones, E., Martorell, X.: A Functional Safety OpenMP* for Critical Real-Time Embedded Systems. In: International Workshop on OpenMP. pp. 231–245. Springer (2017)
19. Royuela, S., Ferrer, R., Caballero, D., Martorell, X.: Compiler analysis for OpenMP tasks correctness. In: Computing Frontiers. pp. 1–8 (2015)
20. Royuela, S., Pinho, L.M., Quiñones, E.: Enabling Ada and OpenMP runtimes interoperability through template-based execution. Journal of Systems Architecture **105**, 101702 (2020)
21. Serrano, M.A., Royuela, S., Quiñones, E.: Towards an OpenMP Specification for Critical Real-Time Systems. In: International Workshop on OpenMP (2018)
22. Tagliavini, G., Cesarini, D., Marongiu, A.: Unleashing fine-grained parallelism on embedded many-core accelerators with lightweight OpenMP tasking. Transactions on Parallel and Distributed Systems **29**(9), 2150–2163 (2018)

23. Toledo, L., Peña, A.J., Catalán, S., Valero-Lara, P.: Tasking in Accelerators: Performance Evaluation. In: 20th International Conference on Parallel and Distributed Computing, Applications and Technologies. pp. 127–132. IEEE (2019)
24. Varbanescu, A.L., Hijma, P., Van Nieuwpoort, R., Bal, H.: Towards an effective unified programming model for many-cores. In: IPDPS. pp. 681–692. IEEE (2011)
25. Voss, M., Chiu, E., Chow, P.M.Y., Wong, C., Yuen, K.: An evaluation of auto-scoping in openmp. In: International Workshop on OpenMP Applications and Tools. pp. 98–109. Springer (2004)
26. Wang, C.K., Chen, P.S.: Automatic scoping of task clauses for the openmp tasking model. The Journal of Supercomputing **71**(3), 808–823 (2015)
27. Willhalm, T., Popovici, N.: Putting intel® threading building blocks to work. In: 1st International Workshop on Multicore Software Engineering. pp. 3–4 (2008)
28. Yu, C., Royuela, S., Quiñones, E.: OpenMP to CUDA graphs: a compiler-based transformation to enhance the programmability of NVIDIA devices. 23rd international workshop on Software & compilers for embedded systems (2020)