

Enabling Ada and OpenMP runtimes interoperability through template-based execution

Sara Royuela^{a,*}, Luís Miguel Pinho^{b,*}, Eduardo Quiñones^{a,*}

^a Barcelona Supercomputing Center, Spain

^b ISEP, Polytechnic Institute of Porto, Portugal

ARTICLE INFO

Keywords:

Concurrency
Parallelism
Ada
OpenMP
Safety
Runtimes

ABSTRACT

The growing trend to support parallel computation to enable the performance gains of the recent hardware architectures is increasingly present in more conservative domains, such as safety-critical systems. Applications such as autonomous driving require levels of performance only achievable by fully leveraging the potential parallelism in these architectures. To address this requirement, the Ada language, designed for safety and robustness, is considering to support parallel features in the next revision of the standard (Ada 202X). Recent works have motivated the use of OpenMP, a de facto standard in high-performance computing, to enable parallelism in Ada, showing the compatibility of the two models, and proposing static analysis to enhance reliability. This paper summarizes these previous efforts towards the integration of OpenMP into Ada to exploit its benefits in terms of portability, programmability and performance, while providing the safety benefits of Ada in terms of correctness. The paper extends those works proposing and evaluating an application transformation that enables the OpenMP and the Ada runtimes to operate (under certain restrictions) as they were integrated. The objective is to allow Ada programmers to (naturally) experiment and evaluate the benefits of parallelizing concurrent Ada tasks with OpenMP while ensuring the compliance with both specifications.

1. Introduction

Safety-critical systems have evolved to such a degree that the use of parallel paradigms is crucial to deliver the levels of performance necessary to implement the most advanced functionalities (e.g., autonomous driving). This trend has arrived to Ada, a language designed for safe and secure programming which is widely used in safety-critical domains, such as avionics and aerospace. In this regard, two complementary research lines are tackling the extension of Ada to support parallelism: a) The simple yet powerful language-based parallel model that, based on a fully strict fork-join model, is able to exploit structured parallelism on shared memory architectures; and b) the incorporation of the OpenMP parallel programming model into Ada, to efficiently exploit structured and unstructured parallelism. This work focuses on the latter approach (although it is discussed its comparison with the former, and the use-cases where that restricted model can be exploited).

OpenMP is a parallel programming model extensively used in High-Performance Computing (HPC) domains, that offers a tasking model very suitable to cope with unstructured and highly dynamic parallelism. It defines *tasks* as units of parallelism composed of the task's executable code and its data environment, as well as different synchronization mechanisms (e.g., point-to-point synchronizations via data

dependencies, and full synchronizations via memory fences). This, coupled with the accelerator model, allows targeting from simple SMP (Symmetric Multiprocessing) machines, to complex and heterogeneous architectures, all using the same programming model.

This paper presents the integration of the OpenMP parallel programming model into the Ada language to fully exploit the benefits of OpenMP, in terms of portability, programmability and performance, while providing the safety benefits of the Ada language, in terms of correctness. We divide our contribution in three main pillars: (1) The *programming model*, i.e., how the OpenMP directives are integrated in Ada at the language level, (2) the *compiler*, i.e., the static analysis and transformations needed to ensure correctness, and (3) the *runtime*, i.e., the interoperability needed between the Ada runtime and the OpenMP runtime. These three contributions have been presented in [1–3] respectively.

Concretely, regarding the programming model, we propose a new syntax for OpenMP and Ada (OpenMP is only supported by C, C++ and Fortran languages) that aims to maintain the clarity and certainty, a distinct characteristic of Ada. Regarding the compiler, we propose a series of compiler analysis techniques that seek data races in Ada and Ada+OpenMP programs and provide the user with feedback to solve the errors. Finally, regarding the runtime, we prove that OpenMP fully

* Corresponding authors.

E-mail address: sara.royuela@bsc.es (S. Royuela).

supports the Ada 202X parallel model (and hence can be used to implement it), as well as analyze the information that must be interchanged between the two runtimes (Ada and OpenMP) in order to ensure a correct interoperability among them and guarantee safety requirements (such as a priority driven scheduling).

This paper further extends the work done at the runtime level and proposes a source-code transformation that enables the OpenMP and the Ada runtimes to operate (under certain restrictions) as they were actually integrated into a unified framework. The objective of our proposal is to allow Ada programmers to naturally experiment and evaluate the benefits of parallelizing concurrent Ada tasks with OpenMP, ensuring that both, the Ada and the OpenMP runtimes, are compliant with the respective specifications.

The remainder of this paper is organized as follows: [Section 2](#) introduces the parallel programming models used in this work, which are the Ada 202X parallel model and OpenMP; [Section 3](#) analyzes the benefits that exploiting OpenMP can provide to Ada users in terms of programmability and performance, and hence motivates the use of this parallel model to boost Ada applications; [Section 4](#) compares the Ada 202X parallel model and the OpenMP programming model to prove that OpenMP can be used to implement the Ada parallel model, and also exposes the syntax needed to use OpenMP directly in Ada applications; [Section 5](#) presents a series of compiler analysis techniques needed to ensure that Ada and Ada + OpenMP codes are data race free; [Section 6](#) introduces a new source-code template that allows Ada programmers to introduce OpenMP naturally in their codes while ensuring the correct interoperability between the two runtimes, and evaluates the actual interaction between Ada and OpenMP at thread level; finally, [Section 7](#) shows the conclusions of our work.

2. Programming models

For this work we consider two programming models: the Ada language-based parallel model, which offers extensions to the Ada language to support fine-grained parallelism, and OpenMP, which offers a complete API for exploiting several forms of parallelism. This section first motivates the selection of OpenMP. Then, it introduces the two parallel programming models, describing the execution and memory model, to ease the reading of the rest of the document.

2.1. Why OpenMP?

Programming multi-cores is difficult due to the multiple constraints it involves. Hence, the success of a multi-core platform relies on its productivity, which combines performance, programmability and portability. With such a goal, a multitude of programming models coexist. The different approaches can be grouped in three paradigms: (1) *Hardware-centric* models aim to replace the native platform programming with higher-level, user-friendly solutions, and focus on tuning an application to match a chosen platform, making their use a neither scalable nor portable solution (e.g., NVIDIA® CUDA [4]); (2) *application-centric* models deal with the application parallelization from design to implementation, and offer less explicit parallel constructs, which, although portable, may require a full rewriting process to accomplish productivity (e.g., OpenCL [5]); and (3) *parallelism-centric* models provide typical parallelism constructs in a simple and effective way, and at various levels of abstraction, bringing flexibility and expressiveness, while decoupling design from implementation (e.g., OpenMP [6]).

Given the vast amount of options available, there is a noticeable need to unify programming models for many-cores [7]. In that sense, OpenMP has proved many advantages over its competitors considering all performance, programmability and portability. On one hand, the OpenMP Application Program Interface (API) offers a simple yet complete and flexible platform for writing multi-threaded applications with C/C++ and Fortran by means of a number of compiler directives, runtime library routines and environment variables. It relies on compiler and

runtime support to implement its functionalities. In essence, the language is built around systems where multiple concurrent threads have access to a shared-memory space; however, it has evolved to target more complex and heterogeneous systems. On the other hand, different evaluations demonstrate that OpenMP delivers comparable performance and efficiency compared to highly tunable models such as TBB [8], CUDA [9], OpenCL [10], and MPI [11]. Moreover, OpenMP has different advantages over low-level libraries such as Pthreads: a) It offers robustness without sacrificing performance [12], and b) OpenMP does not lock the software to a specific number of threads. Another advantage is that the code can be compiled as a single-threaded application just disabling support for OpenMP, thus easing debugging.

Overall, the use of OpenMP presents three main advantages: (1) An expert community has constantly reviewed and augmented the language for more than twenty years, thus, less effort is needed to introduce fine-grained parallelism in Ada; (2) OpenMP is widely implemented by several chip and compiler vendors (e.g., GNU [13], Intel® [14], and IBM [15]), meaning that less effort is needed to manage parallelism as the OpenMP runtime will manage it; and (3) OpenMP provides greater expressiveness due to years of experience in its development; in this regard, the language offers several directives for parallelization and synchronization, along with a large number of clauses that allow to contextualize concurrency, providing a finer control of the parallelism. In a nutshell, OpenMP is a good candidate to introduce fine-grained parallelism to Ada by virtue of its benefits.

2.2. OpenMP

Initial versions of OpenMP, up to version 2.5 [16], implemented a *thread-centric* model of parallelism that defines a conceptual abstraction of user-level threads exposing the management of the underlying resources to the user. This model relies on the `parallel` and a series of worksharing constructs (e.g., `for` and `sections`), and enforces a rather structured parallelism. Next releases, since version 3.0 [17], introduced support for a *task-centric* model (a.k.a. *tasking model*), which is oblivious of the physical layout, and focuses on exposing parallelism rather than mapping parallelism to threads. As a result, this model allows defining unstructured and highly dynamic parallelism by means of the `task` construct. Finally, since version 4.0 [18], OpenMP includes support for accelerators, error handling, thread affinity and SIMD extensions, as well as augments the tasking model (e.g., data dependencies, the `taskloop` construct), expanding the language beyond its traditional boundaries.

2.2.1. Execution model

OpenMP implements a fork-join model of parallelism. The program begins as a single thread of execution, called the *initial thread*. The `parallel` construct spawns a *team* of threads at the beginning of the parallel region, and joins the team at the implicit barrier at the end of the parallel region. The amount of computing resources can be defined by means of the `num_threads` clause (if none is defined, then the number is implementation defined, although the number of cores is commonly considered). Within the parallel region, work can be distributed among threads by means of work-sharing constructs or tasking constructs. The two models have equivalent performance [19].

The OpenMP tasking model defines preemption points for tasks, called *task scheduling points* (TSPs). These points, defined in the specification (Section 2.10.6 [20]), are the moments at which a thread can stop executing a specific task and start executing a different one. It is the responsibility of the runtime to decide whether a task is preempted (and potentially migrated) or not. Furthermore, OpenMP defines two different approaches to relate tasks to threads: (1) *Tied* tasks are those that are tied to the thread that starts executing them, and (2) *untied* tasks are those that can migrate among threads. This connection between threads and tasks exists because the introduction of the tasking model in version

3.0 had to maintain coherency with the already existing thread-model and, for that reason, tasks are tied by default.

Mutual exclusion is accomplished via the `critical` and `atomic` constructs (while the former allows an arbitrary block of code, the latter only accepts specific simple operations such as assignments and binary operations). Furthermore, synchronization can be defined depending on the granularity: *Full synchronization* is defined by means of the `barrier` and the `taskwait` constructs (while a barrier synchronizes all threads in the current team, a `taskwait` only synchronizes child tasks of the binding task¹), and *point-to-point synchronization* is accomplished by means of dependency clauses. These can define three different ways of data-flow synchronization among tasks, based on the particular dependency clause, which can be: (1) `In`, a task with an l-value as input dependency is eligible to run when all previous tasks with the same l-value as output dependency have finished their execution; (2) `out`, a task with an l-value as output dependency is eligible to run when all previous tasks with the same l-value as input or output dependency have finished their execution; and (3) `inout`, a task with an l-value as inout dependency behaves as if it was an output dependency.

2.2.2. Memory model

OpenMP is based on a relaxed-consistency, shared-memory model. This means there is a memory space shared for all threads, called *memory*. Additionally, each thread has a temporary view of the memory. Intuitively, the temporary view is not always required to be consistent with the memory. Instead, each private view synchronizes with the main memory by means of the OpenMP *flush* operation, specified with a `flush` construct, or implied at different locations such (e.g., at entry of a `parallel` region or at exit from a `critical` region). Hence, memory operations can be freely reordered except around flushes. This synchronization can be implicit (in any, implicit or explicit, synchronization operation causing a memory fence) or explicit (using the `flush` directive). Data cannot be directly synchronized between the temporary view of two different threads.

The view each thread has for a given variable is defined using data-sharing clauses, which can determine the following sharing scopes: (1) *Private*, a new fresh variable is created within the scope; (2) *firstprivate*, a new variable is created in the scope and initialized with the value of the original variable; (3) *lastprivate*, a new variable is created within the scope and the original variable is updated at the end of the execution of the region, and (4) *shared*, the original variable is used in the scope, opening the possibility of race conditions. Additionally, the data-sharing attributes for variables referenced in a construct can be: (1) *Predetermined*, those that, regardless of their occurrences, have a data-sharing attribute determined by the OpenMP model; (2) *explicitly determined*, those that are referenced in a given construct and are listed in a data-sharing attribute clause on the construct; or (3) *implicitly determined*, those that are referenced in a given construct, do not have predetermined data-sharing attributes and are not listed in a data-sharing attribute clause on the construct.

2.3. Ada 202X parallel model

The Ada language includes support for concurrency as part of the language standard, by means of Tasks,² which are entities that denote concurrent actions, and inter-task communication mechanisms such as protected objects or the *rendezvous* mechanism. This model is targeted to support the concurrent functionalities that the software should support, providing coarse-grained parallelism. Hence, it is not suitable to support fine-grained parallelization on the hardware platform, leading in this cases to higher overhead [21].

¹ The *binding region* is the enclosing region that determines the execution context and limits the scope of the effects of the bound region.

² Ada tasks are coarse-grained concurrent entities, not related to OpenMP fine-grained parallel tasks.

To address the evolution for parallel support, a proposal was made to extend Ada with a fine-grained parallel model, based on the notion of *tasklets* [22], where parallelism is not fully controlled by the programmer: the programmer specifies the parallel nature of the algorithm, and the compiler and the runtime have the freedom to organize parallel computations. Based on this model, specific language extensions have been proposed [23] to cover two cases where parallelization is suitable: Parallel blocks and parallel loops, including reductions and iterators. In fact, reductions are more general than their use in loops, but that is not necessary for the work in this paper.

This proposal led to a set of proposed changes of the next revision of the Ada language (Ada 202X, currently in its final working draft [24]). The changes specify that an Ada task (a concurrent activity) can represent multiple logical threads of control (Ada 202X, Section 9) which can proceed in parallel within the context of well specified parallel regions: Parallel blocks and parallel loops).

2.3.1. Execution model

In the Ada parallel model, parallel execution follows a fork-join model, with clear (language-based) parallel regions. In both cases (loops and blocks), the keyword `parallel` allows the compiler to split the work into logical threads of control. In the case of parallel loops, the loop range is split into non-overlapping chunks, each one being possible to process in parallel. For the parallel blocks, separate sequences of statements can execute in parallel, each sequence being mapped to a logical thread of control.

The draft Ada 202X standard does not define how the logical threads of control are executed by the runtime. This provides freedom to the compiler and runtime, as long as the semantics of parallel constructs are guaranteed. In particular, the draft allows a *run-to-completion* model [25] where the logical threads of control are executed by a unique runtime executor (e.g., an operating system thread) until it completes. Note that executors do not necessarily have to run uninterruptedly or to execute on the same core, since they may be scheduled in a preemptive fashion.

2.3.2. Memory model

As the Ada language supports concurrency in the language since its beginnings (Ada 83), it already provides a memory model that considers data races, which is now updated to consider logical threads of control. The language allows a relaxed-consistency memory model where the visibility of the variables may vary within parallel regions, but clearly specifies the semantics which allow for concurrent access to the shared variables (Ada 202X, Section 9.10). For safety reasons, Ada delegates the responsibility of defining this visibility to the compiler, which is in charge to ensure a safe execution.

3. Motivation: The performance benefits of OpenMP

The idea of introducing OpenMP in Ada is quite appealing, but still we need some evidence that: (1) The Ada tasking model may not deliver competitive levels of performance when running fine-grained tasks, (2) OpenMP can efficiently exploit the parallelism introduced in the Ada parallel Model, and (3) OpenMP offers mechanisms, that exist neither in the Ada model nor in the Ada parallel model, to exploit further forms of parallelism. With such a purpose, we have conducted a series of experiments that evaluate the benefits of OpenMP compared to other implementations that exploit parallelism in Ada, i.e., native Ada tasks [26] and Paraffin [27]. The experimental setup used is the following:

Runtimes. We use three runtime implementations that support parallelism:

- *libgomp*, the GNU runtime library for OpenMP from GCC 7.2.
- *GNAT* [28], the GNU runtime library for Ada from GCC 7.2.
- *Paraffin 5.0* [27], a suite for Ada.

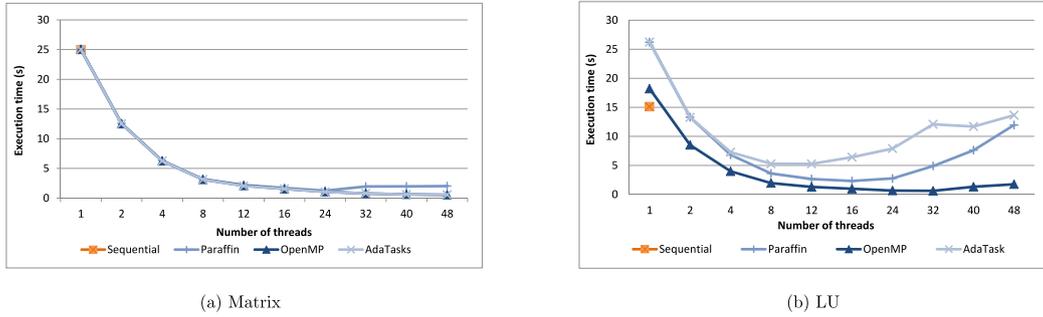


Fig. 1. Scalability analysis of the Ada parallel model implemented with OpenMP, Ada tasks and Paraffin.

Applications and Implementations. We consider three applications: an embarrassingly parallel matrix intensive computation (*Matrix*), the LU factorization (*LU*), and the Cholesky decomposition (*Cholesky*). We use four different parallelization strategies:³ The Ada parallel model implemented with OpenMP, OpenMP (including task dependencies, not available in the Ada parallel model), Ada tasks, and Paraffin. Following we detail the relevant aspects of each version:

- *Matrix*. This application, resembling image processing algorithms, iterates 50,000 times over a 512x512 matrix, and performs independent arithmetical operations on each element. The OpenMP version divides the matrix into blocks, each processed by a different OpenMP task; the number of threads is independent from the number of tasks. The Ada tasks version creates an array of Ada tasks, and assigns a set of rows to each task; the number of threads is determined by the Ada runtime, which uses one thread for each task, and a thread for the main task. Finally, the Paraffin version splits the matrix into rows, and processes in parallel the elements of each row; the number of threads can be defined by the user.
- *LU*. This application computes the LU factorization of a matrix of 64x64 elements, where each element is a 32x32 matrix. The OpenMP version adapts the SparseLU benchmark from the Barcelona OpenMP Task Suite (BOTS) [29], to use a dense matrix instead of a sparse one. The kernel is divided in four phases: *lu0*, *fwd*, *bdiv* and *bmod*; and there are three full synchronizations that divide the execution in three stages: The first containing *lu0*, the second containing *fwd* and *bdiv*, and the third containing *bmod*. These stages are traversed several times. The Ada tasks and the Paraffin implementations are based on the OpenMP version. In both cases, the code is split in three stages and full barriers are implemented in between the stages. In the Ada tasks implementation, for each phase, a different task executes a chunk of iterations (the number of tasks created is the number of threads available, plus one task for the main function). In the Paraffin implementation, each phase is processed as a parallel loop.
- *Cholesky*. This application computes the Cholesky decomposition of a matrix of 128x128 elements, where each element is a 32x32 matrix. The OpenMP version is based on the Cholesky implementation of Ayguadé et. al for extending the OpenMP tasking model [30] to target heterogeneous architectures. As for LU, the Ada tasks and the Paraffin implementations are based on the OpenMP one and mimic the stages of that version.

Platform. We run our experiments in a computing node from the MareNostrum IV [31] supercomputer.⁴ It consists of a 2 sockets Intel Xeon Platinum 8160 CPU with 24 cores each, operating at 2.10GHz, and featuring a 33MB shared L3 cache. The L1 and L2 caches are private to each socket: the former has 32KB, and the latter has 1MB. The system runs a SUSE Linux Enterprise Server 12 SP2 operating system.

First, we analyze the need for fine-grained parallelization and synchronization mechanisms in Ada. With such a purpose, we evaluate the scalability of the *Matrix* and the *LU* benchmarks implemented with three strategies: The Ada 202X parallel model, Ada tasks and Paraffin. Since the Ada 202X parallel model is not supported by any Ada runtime yet, we use OpenMP directives, i.e., the `task` construct to create units of concurrency, and the `taskwait` construct to synchronize tasks, to implement the proposed Ada operations for *parallel loops* and *parallel blocks*. Interestingly, this shows how OpenMP can be used to implement the Ada parallel model (Section 4 analyzes the equivalence of the two parallel models).

Fig. 1 depicts the mentioned scalability analysis for the *Matrix* and the *LU* benchmarks. Particularly, each plot shows the execution time (in seconds) of the three versions when modifying the number of threads, and the time of the sequential version, only for one thread. In the *Matrix* example, in Fig. 1a, all implementations show a good exploitation of the resources: Up to 24 threads, all have a ideal speedup; after that, only Ada tasks and OpenMP have linear speedup, while Paraffin requires more time for synchronization.⁵ The structured and embarrassingly parallel nature of the algorithm allows the three techniques to extract benefits from the parallel execution. However, it is important to note that the granularity of the OpenMP tasks is much finer than the other two versions. With this example, we show how OpenMP can be used to efficiently implement the Ada parallel model. For the *LU* example, in Fig. 1b, the Ada parallel model implemented with OpenMP clearly outperforms the other implementations. Particularly, the Ada parallel model shows ideal speedup up to 24 threads; after that, the synchronization costs limit the performance gain of the parallel execution. This is so because the architecture used is a NUMA machine and L1 and L2 caches are private to each socket; hence, each time a `taskwait` is encountered, and so a memory flush occurs (enforced by the OpenMP specification), the different cache levels have to be updated for cache coherence. Compared to the other versions, the fine-grained synchronization mechanisms provided by OpenMP show much better efficiency than those of Ada tasks and Paraffin.

⁴ We evaluate the performance of OpenMP on a chip from the HPC domain because it offers more computational capabilities than typical embedded systems. However, tests conducted in embedded platforms show similar trends regarding performance and scalability with OpenMP [32,33].

⁵ A new version of the Paraffin suite is to be released soon. This new version reduces synchronization costs and, possibly, enhances the results shown in this article.

³ The source codes of all implemented strategies of the *Matrix*, *LU*, and *Cholesky* benchmarks are publicly available at https://github.com/sroyuela/ada_omp_jsa_apps.

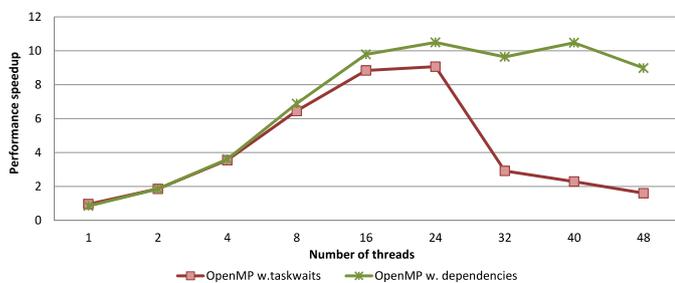


Fig. 2. Speedup of Cholesky using structured (Ada parallel model) and unstructured (OpenMP task dependencies) parallelism.

To further analyze the use of OpenMP on top of Ada, we have used the point-to-point synchronizations provided by OpenMP in the form of task dependencies. This mechanism allows to extract parallelism out of highly unstructured applications. Fig. 2 shows the results obtained with the Cholesky benchmark parallelized with two versions of OpenMP: One implementing structured parallelism using the `taskwait` construct, and the other implementing unstructured parallelism using task dependencies. The version with dependencies outperforms when then number of cores is higher than 4, because it takes profit of all the parallelism existing in the application, while taskwaits are coarse-grained synchronizations that limit parallelism. Furthermore, the cost of synchronizing threads (noticeable in the taskwaits version) is higher when the number of cores used is bigger than 24 because of the NUMA architecture of the platform, as explained for the *LU* benchmark.

4. OpenMP for fine-grained parallelism in Ada

The OpenMP tasking model follows the same principle as the Ada parallel model, where the compiler and the runtime system are the ones responsible for generating and executing the OpenMP tasks. We take advantage of that with the aim of introducing OpenMP into Ada. This section analyzes the compatibility of the Ada 202X parallel model and OpenMP, and proves that OpenMP can be used to implement the Ada parallel model as well. Furthermore, this section shows the language extensions we propose in order to use OpenMP on top of Ada to further exploit unstructured and highly dynamic parallelism. The work summarized in this Section has been presented in [1] and [3].

4.1. Supporting the Ada parallel model with OpenMP

This section provides insight about the OpenMP features necessary to implement the Ada parallel model based on their execution models. It focuses in three main aspects: The preemption model, the progression model and the fork-join model. The next paragraphs dig into each aspect.

Preemption. The limited form of run-to-completion that was proposed in the tasklet model can be mapped to the OpenMP tasking model straightforwardly: The logical threads of control are mapped to OpenMP tasks, and are executed by OpenMP threads. Furthermore, untied tasks are more suitable to implement this, because tasks can migrate between threads. Moreover, untied tasks have better time predictability than tied tasks, due to their work-conserving nature [34]. On the other hand, although the work-sharing constructs provided by the thread-centric model can implement the same semantics as Ada parallel blocks and parallel loops do, work-sharing entities cannot be preempted by the runtime, and therefore this model is not suitable to support the Ada completion model.

Progression model. The OpenMP specification does not impose any model of progression; the same is being prescribed in the Ada 202X draft. Both models rely on the implementation to guarantee safe execution.

Fork-join model. The fully strict fork-join model required by the Ada parallel model is fully supported by OpenMP. Since OpenMP does not force the distribution of work to be done at the same point as the spawn of parallelism, OpenMP constructs are more flexible. For example, when implementing parallel nested blocks with the OpenMP tasking model two possibilities are valid: (1) Use a unique parallel region (hence a unique team of threads) with nested tasks, or (2) spawn parallelism twice by nesting parallel regions. The first option may reduce the overhead of creating and destroying extra teams of threads (the nested ones). However, it is interesting to have the possibility of exploiting two different levels of parallelism for different reasons: Parallelism is not exposed at the same level, or the application is not balanced, among others.

4.2. Supporting OpenMP in Ada

Besides the feasibility of using OpenMP to implement the Ada parallel model, as shown in Section 4.1, OpenMP can be used on top of Ada to exploit its benefits, as demonstrated in Section 3. This section shows the language extensions required to use OpenMP in Ada, and analyzes the expressiveness of OpenMP against that of Ada.

4.2.1. Language extensions

The current OpenMP specification is defined for C, C++ and Fortran. In this regard, the syntax of Ada is closer to that of Fortran than the one for C/C++, because Ada does not group a sequence of statements by bracketing the group (as in C), but uses a more structured approach with a closing statement to match the beginning of the group (as in Fortran). Since Ada already defines *pragmas* of the form `pragma Name (Parameter_List)`, our proposal introduces a new kind of pragma, `pragma OMP`, together with the directive name (e.g., `task`, `barrier`, etc.), and the clauses that go with the directive (e.g., `dependencies`), included as parameters of the pragma (although we propose the use of pragmas, a similar approach can be used with Ada aspects). The snippet in Listing 1 shows an example of the proposed syntax when an OpenMP construct (`taskloop` in this case) applies to one statement (the loop associated to the construct), and the snippet in Listing 2 shows an example where the construct (`task`) applies to more than one statement (the structured block associated to the task).

OpenMP defines the argument of a data-sharing clause as a list of items. This does not match directly with the syntax allowed in Ada for pragmas, as shown in Listing 3, where there can only be one element (a name or an expression) associated with a particular identifier. To simplify the syntax needed to define data-sharing clauses, we propose to allow a list of names or expressions associated with each `pragma_argument_association` with a list of expressions. We use this proposed syntax for the rest of the document.

With these extensions, OpenMP can be used to express the same forms of parallelism as the Ada parallel model (i.e., parallel blocks and parallel loops, including a limited form of reduction), and further exploit other forms of parallelism (unstructured and highly dynamic

```

1 pragma OMP (taskloop, num_tasks=>N);
2 for i in range 0..I loop
3   ... -- statements here
4 end loop;

```

Listing 1. OpenMP proposed syntax for pragmas applying to one statement.

```

1 pragma OMP (task, shared=>var);
2 begin
3   ... -- statements here
4 end;

```

Listing 2. OpenMP proposed syntax for pragmas applying to several statements.

```

1 pragma ::=
2   pragma identifier [(pragma_argument_association {,
3   pragma_argument_association})];
4 pragma_argument_association ::=
5   [pragma_argument_identifier =>] name
6   | [pragma_argument_identifier =>] expression

```

Listing 3. Ada syntax for pragmas.

```

1 if N < 2 then
2   return N;
3 parallel do
4   X:= Fibonacci(N - 2);
5 and
6   Y:= Fibonacci(N - 2);
7 end do;
8 return X + Y;

```

Listing 4. Parallel Fibonacci sequence with Ada extensions.

```

1 if N < 2 then
2   return N;
3 pragma OMP (parallel, shared=>X,Y,
4   firstprivate=>N);
5 pragma OMP (single, nowait);
6 begin
7   pragma OMP (task, shared=>X,
8   firstprivate=>N);
9   X:= Fibonacci(N - 2);
10  pragma OMP (task, shared=>Y,
11  firstprivate=>N);
12  Y:= Fibonacci(N - 2);
13 end;
14 return X + Y;

```

Listing 5. Parallel Fibonacci sequence with OpenMP tasks.

```

1 parallel for i in 0..MAX_I loop
2   for j in range 0..MAX_J loop
3     for k in range 0..MAX_K loop
4       RES(i,j) := RES(i,j)
5         + M1(i,k) * M2(k,j);
6     end loop;
7   end loop;
8 end loop;

```

Listing 6. Parallel matrix multiplication with Ada extensions.

applications). The next paragraphs show snippets of how the Ada tasklet model can be expressed using the OpenMP tasking model.

Parallel blocks. A parallel block denotes two or more concurrent sections. The Ada extensions proposed for such a purpose are shown in Listing 4. In OpenMP, a parallel block can be written using the thread-centric model (using the `sections` and `section` constructs) or the task-centric model (using the `single` and `task` constructs), depicted in Listing 5. In this code, the `parallel` construct spawns parallelism, the `single` construct indicates that only one thread in the team executes the associated region, and the `task` constructs distributes parallelism among threads of the team.

Parallel loop. A parallel loop defines a loop where iterations may be executed in parallel. The Ada syntax for such a structure is depicted in Listing 6. OpenMP offers two different constructs this purpose: (1) The `for` construct, from the thread-centric model, and (2) the `taskloop` construct, from the tasking model, shown in Listing 7. In both cases, we illustrate the directives using the well-known matrix multiplication benchmark, that considers two matrices M1 and M2, and the matrix RES, where their multiplication is stored.

```

1 pragma OMP (parallel);
2 pragma OMP (taskloop,
3   private=>i, j, k,
4   firstprivate=>MAX_I, MAX_J, MAX_K,
5   shared=>RES, M1, M2,
6   grainsize=>chunk_size,
7   nowait);
8 begin
9   for i in range 0..MAX_I loop
10    for j in range 0..MAX_J loop
11      for k in range 0..MAX_K loop
12        RES(i,j) := RES(i,j)
13          + M1(i,k) * M2(k,j);
14      end loop;
15    end loop;
16  end loop;
17 end

```

Listing 7. Parallel matrix multiplication with OpenMP taskloop.

Parallel reduction. The Ada parallel model defines a reduction as an operation which transforms a collection of values into a single value result, allowing builtin operations to be used (e.g., +, -, *, etc.), as well as used-defined reducers and combiners. This is achieved by *reduction expressions*, which can be made parallel. Similarly, OpenMP defines a reduction as a parallel operation which result is stored in a variable, supporting builtin and used-defined reductions. The reduction itself is implemented in OpenMP by means of a clause that can be added to multiple constructs like `parallel` and `taskloop` among others. Listing 8 shows the syntax proposed for Ada2020, while Listing 9 shows the syntax adapted to our proposal for Ada.

4.2.2. Expressiveness

The Ada 202X parallel model is a simple yet powerful model to exploit structured parallelism in shared memory architectures. However, fully strict fork-join models limit the exploitation of unstructured parallelism. In that respect, OpenMP supports point-to-point synchronization by means of the `depend` clause, which defines the input and/or output data dependencies existing between tasks. The *task dependency graph* that honors these dependencies is then used at runtime to drive the execution. The use of dependencies can significantly improve performance of parallel Ada programs, as shown in Section 3.

Additional to data dependencies, OpenMP allows programmers to manually define the data access model of the variables in a construct by means of data-sharing clauses. The examples shown before specify the access to the data within the OpenMP constructs. For example, in Listing 5, X and Y are marked as shared because their value has to be visible outside the parallel region, after the implicit barrier, and there is no data-race condition in these accesses, and N is marked as firstprivate because the value is just read within the parallel region. In the Ada parallel model, the philosophy is different: data-sharing accesses which are not protected are expected to be flagged by the compiler, hence no data-sharing attributes are specified. For example, in Listing 4, the compiler can detect that no unsafe access is made to N, X or Y in the parallel block, thus conclude no synchronization is required, except for the one at the end of the parallel block. Moreover, it can privatize X and Y, copying out their value after the parallel computation completes. This however, may harm performance due to the extra copies (it remains as a compiler decision). The logic behind the choice to make data-sharing

```

1  parallel (Chunk in Partial_Sum'Range)
2  for I in Arr'Range loop
3      Partial_Sum(Chunk) := Partial_Sum(Chunk) + Arr(I);
4  end loop;
5  Sum := Partial_Sum'Reduce('+',0.0);

```

Listing 8. Parallel reduction with Ada extensions.

```

1  pragma OMP parallel (taskloop, in_reduction=>+,Sum);
2  begin
3      for I in Arr'Range loop
4          Sum := Arr(I);
5      end loop;
6  end;

```

Listing 9. Parallel reduction with OpenMP taskloop.

transparent to the user is based on simplicity and readability, whilst safe.

Furthermore, OpenMP offers different mechanisms to tune the scheduling of parallel work. For example, the `for` worksharing construct allows to define how iterations are mapped to threads by means of the `schedule`, `order` and `ordered` clauses, and the `taskloop` construct allows defining how many tasks are created and hence their granularity, using either the `num_tasks` or the `grainsize` clauses (these are mutually exclusive). In opposition, the Ada parallel model is limited to defining the maximum number of chunks of a parallel loop.

Finally, OpenMP supports an accelerator model seamlessly integrated with the tasking model that features the efficient distribution of parallelism in heterogeneous systems, which widens the spectrum of architectures that can be targeted by Ada applications.

Overall, the possibilities with OpenMP underscore their versatility in the face of the proposed Ada extensions. However, despite the clear benefits of OpenMP to boost performance in Ada applications, there is still work to do to fulfill the safety-critical domain requirements. Firstly, OpenMP does not impose the compiler to identify errors that may affect the correctness of the application, e.g., data-races or deadlocks. Moreover, OpenMP is not reliable because it does not define any recovery mechanism, with the exception of the cancellation model, for Ada exception handling. In that regard, different approaches have been proposed and some of them have been already adopted (see further details in Section 5.1). Finally, both programmers and compilers must satisfy some requirements to make possible whole program analysis (such as programmers adding information in headers libraries, and compilers implementing techniques like IPO [35]). The next section studies compiler analyses techniques that, applied to OpenMP and Ada compilers, can significantly improve the safety of Ada programs parallelized with OpenMP, and so enabling safety-critical systems to efficiently exploit highly parallel and heterogeneous architectures.

5. Compiler support for functional safety

A fundamental requirement of Ada systems is safety, which can be certified at different levels by means of particular standards (e.g., the ISO26262 [36] for automotive, the DO178C [37] for avionics or the IEC61508 [38] for industry). Problems with certification might be due to error-prone features (compromising reliability) or features with complex semantics (complicating analyzability). For this reason, the nature of Ada is to prevent users from making errors, providing a series of mechanisms for data synchronization and mutual exclusion, among others. Furthermore, the language is designed such that the compiler can detect the maximum number of risky situations, like race conditions and deadlocks. And the recent additions to Ada 202X in this domain augment the capability to detect the unprotected use of shared variables and potentially blocking operations [23]. Still, it is the responsibility of the programmers to use Ada mechanisms correctly in order to avoid errors.

OpenMP also provides mechanisms for data synchronization and mutual exclusion. As for Ada, the correct use of these mechanisms relies on

the programmer. This is stated in the specification, when it says that “*application developers are responsible for correctly using the OpenMP API to produce a conforming program*”⁶. Thus, frameworks do not need to check for issues such as data dependencies, race conditions or deadlocks. As a result, the implementation of the standard is quite easy and light, and that boosts the spreading of the language even in architectures with few resources.

In this context, it is fundamental to consider correctness checking mechanisms to ensure programs are free from errors and hence increase productivity in parallel programming. This section, summarizing the work presented in [39] and [2], includes an analysis of the safety of both OpenMP and the Ada parallel model, and provides an algorithm that allows detecting race conditions in pure Ada programs and in mixed Ada/OpenMP programs as well.

5.1. Safety

Considering the Ada Parallel model, safety can be guaranteed through the use of atomic variables and protected objects to access shared data. Moreover, the compiler shall be able to complain if different parallel regions might have conflicting side-effects. In that respect, due to the hardship of accessing the complete source code to perform a full analysis, the proposed Ada extensions suggests a two-fold solution [23]: a) Eliminate race conditions by adding an extended version of the SPARK `Global` aspect to the language (this will help the compiler to identify those memory locations that are read and written without requiring access to the complete code); and b) address deadlocks by the defined execution model, together with a new aspect called `Potentially_Blocking` that indicates whether a subprogram contains statements that are potentially blocking.

On the other hand, considering OpenMP, safety can be jeopardized due to the use of different features. The most relevant ones are the following:

- *Data-sharing*. Users can explicitly modify the data-sharing attributed defined in the specification (concretely, in Section 2.15.1 [40]) for the variables appearing in a specific construct. But manually defining data-sharing clauses is a cumbersome and error-prone process because programmers have to be aware of the memory model and analyze the usage of the variables. Fortunately, there are compiler analysis techniques that allow automatically defining data-sharing clauses [41] and statically catch incoherencies in the user-defined attributes [42].
- *Data Races and Synchronization*. Detecting exact data races at compile time is an open challenge. Still, current mechanisms have been shown to work on specific subsets of OpenMP [43,44]. Additionally, static analysis techniques have proved to be able to detect wrong synchronizations causing non-deterministic results and runtime failures [42].

⁶ An OpenMP *conforming* program is one that follows all rules and restrictions of the OpenMP specification.

- *Deadlocks*. The different mechanism offered in OpenMP to synchronize threads (directives such as `critical` and `barrier`, and runtime routines, such as `omp_set_lock`) can cause deadlocks. There is only one sound approach, to the best of our knowledge, which detects deadlocks in C programs using Pthreads [45]. This technique can easily be applied to OpenMP because Pthreads mutexes (e.g., `pthread_mutex_lock`) are comparable to OpenMP locking routines (e.g., `omp_set_lock`).
- *Error Handling*. In the critical domain, software is required to be resilient, hence behavior upon failures must be understood and specified. The technique to enable such a property is error handling. Although only some minor mechanisms have been included in the specification (i.e., cancellation constructs), there are different proposals to improve OpenMP reliability by adopting error handling mechanisms in OpenMP [46,47].

In this sense, OpenMP has been shown to provide the safety requirements imposed by critical systems [1] if the language incorporates:

- Limits in the specification that may vary depending on the level of criticality (e.g., task priorities and explicit flushes).
- Extensions to the specification (the two new directives `globals` and `usage`) to enable whole program analysis when third-party components are used, hence detect race conditions and illegal nesting⁷ (including nested regions that can cause deadlocks).
- Extensions to include error-handling techniques.
- Compiler implementation guidelines to check correctness.
- Runtime implementation guidelines to avoid faulty results.

5.2. Static data race detection for Ada/OpenMP

As introduced previously, parallel computation gives rise to two main problems: race-conditions and deadlocks. In this section we focus on the former, and we propose a compiler mechanism to detect race conditions in programs using Ada, OpenMP and both of them. This mechanism is composed of two steps: first the representation of the parallel semantics of the code in a Parallel Control Flow Graph (PCFG), and second an algorithm that allows automatically synchronizing both tasks and data to avoid race conditions.

The remaining of this section is organized as follows: first we introduce the PCFG, then we describe the algorithm to avoid race conditions, and finally we use a use-case to illustrate the application of our technique.

5.2.1. Representing parallel semantics: The PCFG

To represent the behavior of an Ada/OpenMP program we use the classic control flow graph (CFG) representation extended to support Ada concurrency and OpenMP parallelism. Our graph draws from the parallel control flow graph for C/C++ and OpenMP/OmpSs [48] developed by Royuela et al. [42], and the control flow graph for Ada developed by Fechete and Kienesberger [49]. We have included in the PCFG the concept of *block of concurrency*, or *concurrent block*, which defines a set of portions of code that may execute in parallel.

The PCFG is a meta-graph composed of a set of nodes and a set of edges. Nodes can be *simple*, representing sequential execution of one or more statements, or *structured*, representing control flow (i.e., selection and iteration statements) or parallel semantics (e.g., OpenMP task). Structured nodes are PCFGs. Edges can represent synchronous flow (e.g., a jump statement), asynchronous flow (e.g., an OpenMP task creation) or synchronization (e.g., precedence relation between OpenMP tasks due to dependency clauses).

Currently, the PCFG represents the semantics of OpenMP, and also the Ada Ravenscar profile. The latter is easily supported because in this

restricted model all tasks are created at library level, meaning that they start executing at the beginning of the program (after elaboration) and terminate when the program ends (task allocators, task termination and abortion, and task hierarchies, among others, are not allowed). As a result, there are only two blocks of concurrency, which correspond to the code executed during elaboration, and the rest of the code.

The use of the full Ada concurrency model, however, complicates the representation. In this sense, the PCFG should be extended to include further edges between tasks (e.g., master dependencies, task termination, rendezvous, etc.). These edges must be taken into account when determining the concurrency blocks (considering when tasks come to life and terminate), and also to tune the accuracy of the results of the race condition algorithm proposed in the following section (considering when data is actually accessed, if possible). A detailed analysis and construction of the PCFG for the full Ada concurrency model remains as future work. For this reason, and although the analysis described in the following section applies to the whole Ada model, for this work we consider the Ada Ravenscar profile.

5.2.2. Correctness analysis for Ada/OpenMP data-race detection

Inspired by the algorithms presented in the scope of OpenMP to automatically determine the data-scoping attributes [41] and the dependency clauses [50] of an OpenMP task, we present an algorithm able to find data-race conditions in Ada concurrent programs, containing or not OpenMP tasks. The high-level description of the algorithm is outlined in Listing 1.

Our approach is based on the fact that Ada protected objects are a robust and lightweight mechanism for mutual exclusion and data synchronization. For this reason, protected objects are to be preferred to OpenMP mechanisms whenever possible to solve race conditions, i.e., when race conditions occur between Ada tasks, between Ada and OpenMP tasks, and between OpenMP tasks that belong to different binding regions.⁸ The last case is particularly interesting because in C/C++/Fortran OpenMP programs, tasks in such a situation cannot be synchronized, and only data synchronization is available via the flush operation, a highly unrecommended mechanism when safety is essential due to the difficulty of analyzing its behavior. The extra layer of concurrency introduced by Ada comprises the need for such a synchronization, hence only protected objects are safe enough for that purpose. Finally, to exploit the flexibility of OpenMP, race conditions between OpenMP tasks that belong to the same binding region are to be solved using OpenMP mechanisms: mutual exclusion constructs (i.e., `atomic` and `critical` constructs), synchronization constructs (e.g., `taskwait` and `barrier`), synchronization clauses (i.e., `depend`) and data-sharing clauses (e.g., `private`, `firstprivate` and `lastprivate`).

5.2.3. Use case: Ravenscar

We use the Ada Ravenscar example application, defined in Section 7 of the Ada Ravenscar Profile Guide [52], as test case because it includes several features of Ada that are of our interest: protected objects, other shared data, synchronous and asynchronous synchronizations, etc. The system modeled in this application includes a periodic process (*Regular_Producer*) that handles offers for a variable amount of workload (*Small_Whetstone*). When the requested workload exceeds a given threshold (*Due_Activation*), the excess load is processed by a sporadic process (*On_Call_Producer*). Additionally, interrupts may appear at any point (*External_Event_Server*), and different priorities are used to ensure precedence among the different tasks. Fig. 3 shows the HRT-HOOD⁹ representation of the Ravenscar application. There, red dashed boxes

⁸ In OpenMP, a *binding region* is the enclosing region that determines the execution context. The binding region of a task is the innermost enclosing parallel region.

⁹ Hard Real-Time Hierarchical Object-Oriented Design (HRT-HOOD) is an object-based structured design method for hard real-time systems [53].

⁷ Section 2.17 of the specification [40] defines a series of rules that determine which constructs cannot be nested within each other.

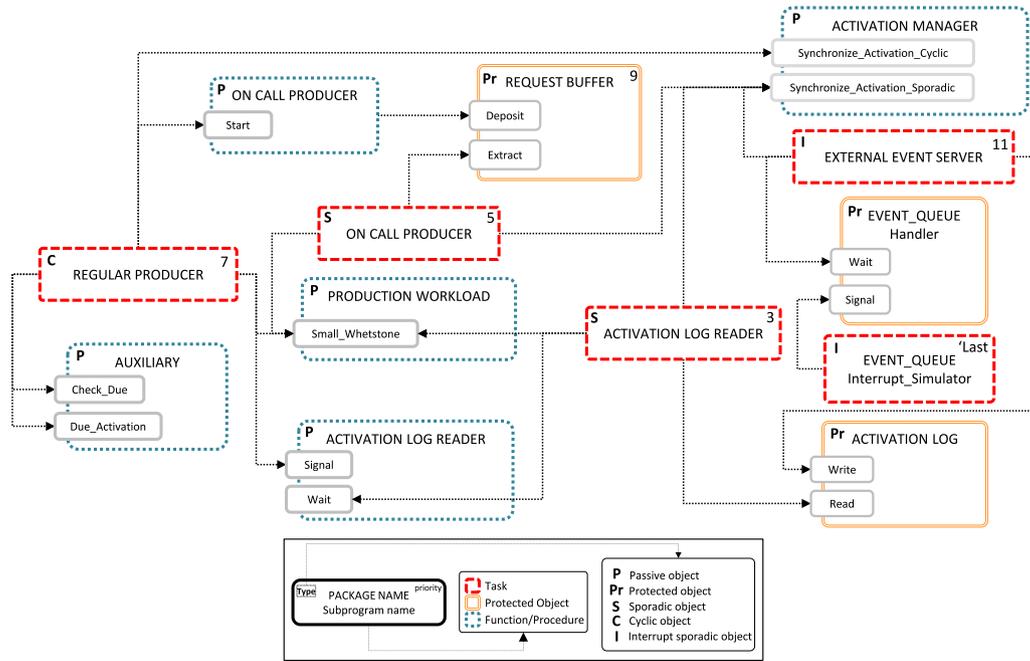


Fig. 3. HRT-HOOD representation of the Ravenscar application.

```

1 package body Production_Workload is
2   type S is range 1 .. 512;
3   type M is array (S, S) of Float;
4   M_A, M_B, M_C: M;
5
6   procedure Read_Sensor_A is begin
7     pragma OMP (parallel);
8     pragma OMP (single);
9     pragma OMP (taskloop);
10    for I in S loop
11      for J in S loop
12        M_A(I,J) := sensor(1, I, J);
13      end loop;
14    end loop;
15  end Read_Sensor_A;
16
17  procedure Read_Sensor_B is begin
18    pragma OMP (parallel);
19    pragma OMP (single);
20    pragma OMP (taskloop);
21    for I in S loop
22      for J in S loop
23        M_B(I,J) := sensor(2, I, J);
24      end loop;
25    end loop;
26  end Read_Sensor_B;
27
28  procedure Fuse_Sensors is
29  begin
30    pragma OMP (parallel);
31    pragma OMP (single);
32    pragma OMP (taskloop);
33    for I in S loop
34      for J in S loop
35        M_C(I, J) := M_A(I, J)
36                  + M_B(I, J);
37      end loop;
38    end loop;
39  end Fuse_Sensors;
40
41  procedure Small_Whetstone
42    (Workload: Positive) is
43  begin
44    case Workload is
45      when 1 => Read_Sensor_A;
46      when 2 => Read_Sensor_B;
47      when 3 => Fuse_Sensors;
48      when others => null;
49    end case;
50  end Small_Whetstone;
51
52 end Production_Workload;

```

Fig. 4. OpenMP code inserted in the *Production_Workload* package of the Ravenscar application.

represent tasks, blue dotted boxes represent packages with functions and procedures, and yellow double-lined boxes represent protected objects with entries and procedures.

To exemplify how the analysis handles Ada concurrency and OpenMP parallelism, we have turned the procedure *Small_Whetstone* into the entry point of a sensor fusion operation implemented with OpenMP. This new functionality, described in Fig. 4, uses an argument to indicate the parallel operation to carry out: 1 for reading sensor A, 2 for reading sensor B, and 3 for fusing the two sensors by adding up their values. Sensor A is read periodically from *Regular_Producer*, sensor B is read sporadically from *On_Call_Producer*, and the fusion is performed sporadically from *Activation_Log_Reader*.

The PCFGs of the original *Ravenscar* application and the new OpenMP code are shown in Figs. 5 and 6 respectively. Both figures show the code executed at elaboration time (on the top), the code run during the execution of the program (in the middle), and the most significant shared data in turquoise square boxes (on the bottom) connected with the nodes that access the data by different edge styles depending on the type of access: Read (dotted dark red), write (solid yellow) and read/write (dashed green). In the former figure, each partial PCFG represents a task (*Regular_Producer*, *On_Call_Producer* and *Activation_Log_Reader*); the special nodes *En* and *Ex* express the entry and the exit points of each task; and the OpenMP code is pointed with dashed-dotted purple lines. In the latter figure, the different procedures are

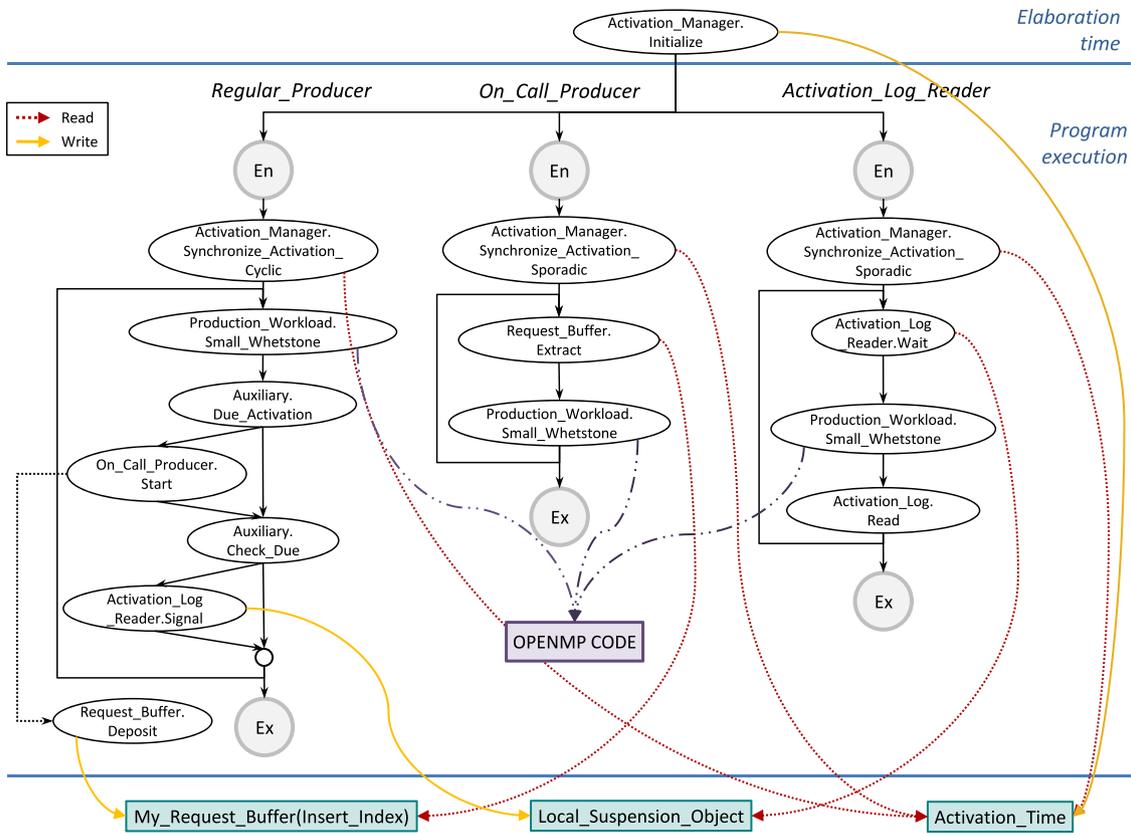


Fig. 5. Simplified PCFG of the Ravenscar application.

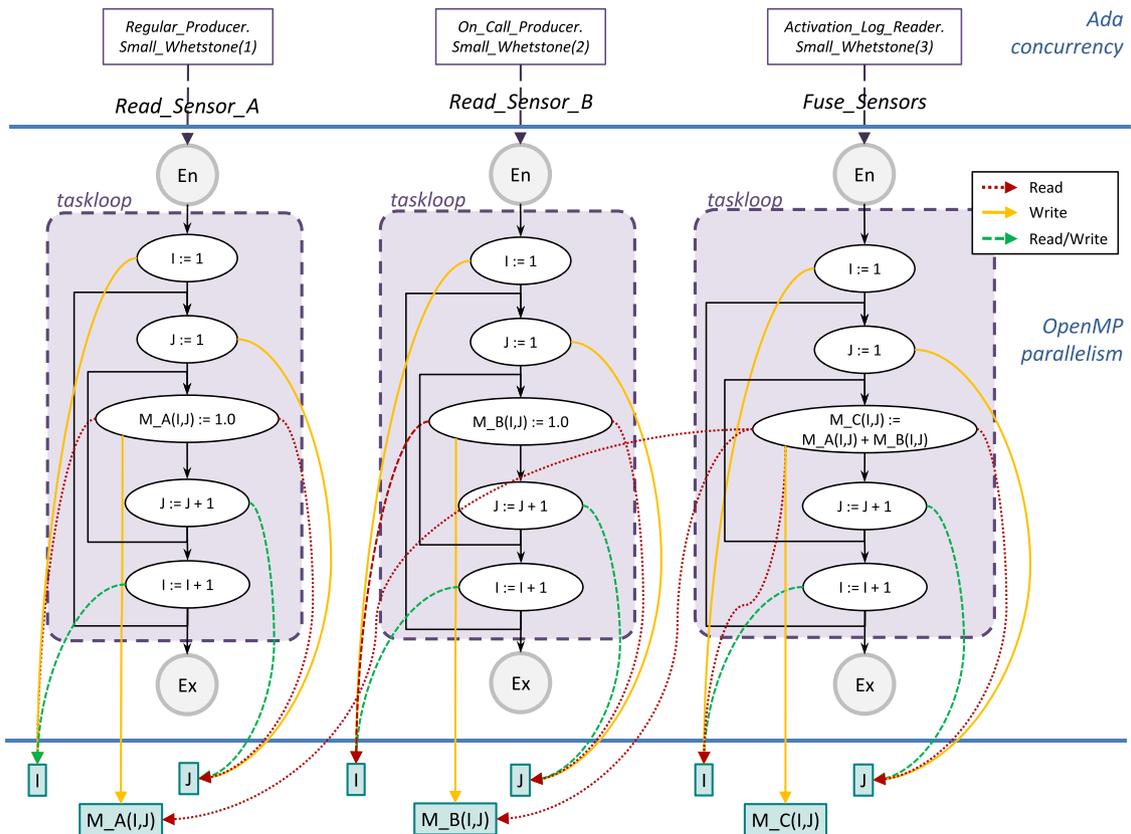


Fig. 6. PCFG of the OpenMP code introduced in the Small_Whetstone procedure.

Algorithm 1: Rules to detect race conditions in Ada/OpenMP [51].

```

Data: source := An Ada/OpenMP program.
Result: target := A race-free version of the source program.
target := source;
pcfg := build_interprocedural_CFG(target);
concurrency_blocks := compute_concurrency_blocks(pcfg);
foreach  $c \in$  concurrency_blocks do
  shared_data := collect_shared_data(c);
  foreach  $s \in$  shared_data do
    all_accesses := collect_accesses(s);
    if within_openmp_same_binding_region(all_accesses) then
      if commutative(all_accesses)[51] then
        target := protect all_accesses with atomic
        orcritical
      else
        target := (full sync) insert taskwait or
        barrierbetween accesses || (point-to-point sync)
        use auto-dependencies mechanism[50]
      else
        target := wrap the shared data in a protected object
    end
  end
end

```

concurrent because they are called from within different Ada tasks, which are in turn concurrent.

As an illustration, we apply [Algorithm 1](#) to the modified *Ravenscar* application, and the result is described as follows:

1. The PCFG of the application is the one shown in [Section 5.2.1](#), [Figs. 5 and 6](#).
2. All Ada and OpenMP tasks correspond to the same block of concurrency, hence potential race conditions may occur among all Ada and OpenMP tasks. However, since OpenMP and Ada tasks manage different shared data, we can treat them separately. As a result:
 - (a) For the Ada part, the algorithm decides that: (a) *Activation_Time* is not in a race condition because the read and the write accesses are in different concurrent blocks, (b) *Local_Suspension_Object* is not in a race condition because the operations performed on it are atomic with respect to each other, as the standard says, and (c) *My_Request_Buffer(Insert_Index)* is not in a race condition because this object is part of the protected object *Request_Buffer*.
 \Rightarrow The algorithm confirms that the original *Ravenscar* application contains no race conditions.
 - (b) For the OpenMP part (note that the OpenMP data-sharing rules dictate a private copy of the induction variable of the taskloop for each thread) the algorithm reveals that accesses to *I* and *J* are not in a race condition, but accesses to the matrices *M_A* and *M_B* are in a race condition because the write access to *M_A* and *M_B* from *Read_Sensor_A* and *Read_Sensor_B* respectively collide with the read access to both variables from *Fuse_Sensor*.
 \Rightarrow The algorithm suggests the use of partial synchronizations in the form of task dependency clauses:
 - *Read_Sensor_A*: depend=>in, M_A(0:Dim,0:Dim).
 - *Read_Sensor_B*: depend=>in, M_B(0:Dim,0:Dim).
 - *Fuse_Sensors*: depend=>in, M_A(0:Dim,0:Dim)
 M_B(0:Dim,0:Dim),
 depend=>out, M_C(0:Dim,0:Dim).

6. Ada and OpenMP runtimes interoperability

Ada supports a concurrency model that allows interleaved execution on single-core architectures, and parallel execution of concurrent work on multi-core architectures. To do so, Ada includes a set of features to achieve concurrency, including Ada tasks, protected objects and priorities. Moreover, the Annex D (Real-Time Systems) [54] of the Ada

specification defines additional characteristics of Ada implementations intended for real-time systems, that limits how these features can be safely used. Among these, priorities and scheduling policies are crucial aspects. For example, the *Ravenscar* profile forces priority-based preemptive scheduling. This means that tasks with higher priority can preempt tasks with lower priority, and the latter will later be resumed depending on the scheduling policy (e.g., FIFO_Within_Priorities, Round_Robin_Within_Priorities). In this regard, OpenMP offers the priority clause that can be attached to the task construct to allow the scheduler to execute the task in a priority-based fashion. Furthermore, OpenMP allows limited preemptive scheduling where tasks can be preempted at task scheduling points (see [Section 2.2.1](#) for further details).

Including OpenMP in an Ada program forces the concurrent model of Ada to coexist with the parallel model of OpenMP. To that end, the two runtimes require some kind of interaction so the scheduling policy of the whole system holds, while each scheduler complies with its corresponding specification. As an illustration, [Fig. 7a](#) shows a program composed of two Ada tasks, a high priority one, *HPT*, and a low priority one, *LPT*, both parallelized using the OpenMP tasking model. [Fig. 7b](#) and [Fig. 7c](#) show different preemption strategies depending on the communication available between the Ada and the OpenMP runtimes.

A first approach that minimizes the interaction between the two runtimes is to completely suspend the execution environment of the OpenMP runtime derived from the lower priority Ada task, when the higher priority Ada task is released. This behavior is shown in [Fig. 7b](#), where the Ada program in [Fig. 7a](#) is executed on two cores and, when *HPT* is released and a preemption point is reached¹⁰, both *OMPT2* and *OMPT3* from *LPT* stop. Then, when *OMPT1* finishes, both *OMPT2* and *OMPT3* can resume. As shown, this approach may force a non-work-conserving scheduling as the *Core 1* is idle while *HPT* executes, and hence introduce unnecessary delays. Moreover, a significant overhead may occur due to the suspension of the complete OpenMP runtime execution.

A second possible approach is to let the two runtimes communicate so just the necessary resources are released when a high priority Ada task is encountered. This behavior is shown in [Fig. 7c](#). There, the Ada program depicted in [Fig. 7a](#) is executed on two cores, but this time only *OMPT2* from *LPT* is stopped to execute *OMPT1* from *HPT* when the preemption point is reached. This is because just one core is needed to execute the high priority Ada task, and hence the other Ada task, although having lower priority, can continue running.

Clearly, the desired behavior is that shown in the second approach, where only the computing resources that are needed by higher priority tasks are preempted. The reason is that this is the only behavior that ensures a work-conserving execution while it honors the priorities in the system as a whole, including the Ada and the OpenMP realms. This strategy, however, implies interoperability at two levels: (1) Between the Ada and the OpenMP runtimes, to handle task priorities and scheduling policies, and (2) within the OpenMP runtime, to communicate different parallel regions.

The remainder of the section is organized as follows: [Section 6.1](#) introduces a theoretical study of how the two runtimes could be integrated. [Section 6.2](#) presents a novel source-code transformation technique based on template-based execution that allows Ada programmers to experiment with OpenMP without requiring the runtimes to be actually integrated. Finally, [Section 6.3](#) presents an evaluation of the use of the source-code transformation templates regarding the interaction of the runtimes and the resources.

6.1. Integration between OpenMP and Ada runtimes: The theory

To support OpenMP in Ada, or simply to implement the Ada parallel model on top of OpenMP, a level of interoperability between the

¹⁰ A limited preemption strategy is being considered in the example.

```

1 task body HPT is begin
2   pragma OMP (parallel);
3   pragma OMP (single);
4   begin
5     pragma OMP (task); -- OMPT1
6     ... -- body
7   end;
8 end HPT;

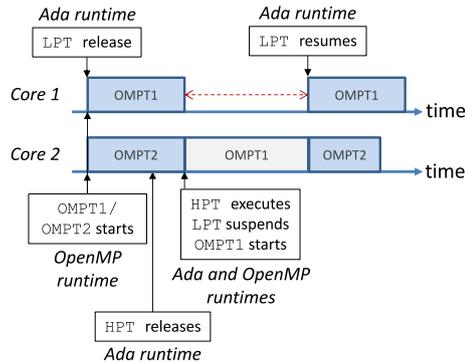
```

```

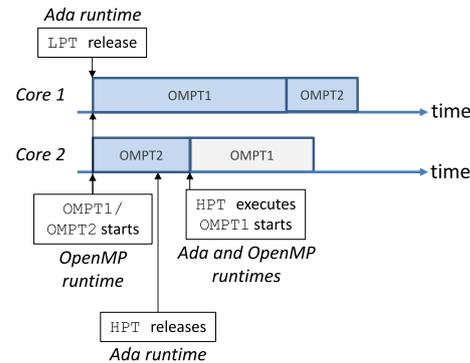
1 task body LPT is begin
2   pragma OMP (parallel);
3   pragma OMP (single);
4   begin
5     pragma OMP (task); -- OMPT2
6     ... -- body
7     pragma OMP (task); -- OMPT3
8     ... -- body
9   end;
10 end LPT;

```

(a) Hybrid Ada/OpenMP system with concurrency and parallelism



(b) Preemption of all inner tasks



(c) Preemption of necessary tasks

Fig. 7. Interoperability between the Ada and the OpenMP runtimes: preemption.

OpenMP and the Ada runtimes is required so compliance with the respective specifications is not compromised. There are three aspects to take into account: (1) Ada tasks scheduling, (2) Ada tasks synchronization, and (3) Ada and OpenMP control structures. These are analyzed as follows.

Ada Task Scheduling. The Ada runtime is in charge of scheduling Ada tasks. When the scheduling conditions change, e.g., a high priority task arrives, a running Ada task with lower priority can be preempted in favor of the high priority one. This scenario is shown in Fig. 7c. When this occurs, the Ada runtime must inform the OpenMP runtime so parallel execution derived from lower priority Ada task can be stopped, in case the high-priority Ada task needs it. The preempted portion of the parallel execution must be safely stopped because OpenMP does not allow dynamically changing the number of threads of a team. A possible solution is the Ada runtime informing the operating system (OS) to release the corresponding cores from the selected Ada task, and the OpenMP runtime informing the OS when the OpenMP tasks executing on the cores to be stopped have reached a *task scheduling point*. Preempted tasks are put back into the *task ready queue* to resume their execution when an OpenMP thread becomes available for the low priority Ada task.

Ada task synchronization: Protected objects. Ada incorporates a deadlock-free mutual exclusion mechanism, named *protected objects*, that can be applied at both Ada task and tasklet levels. Protected objects are commonly implemented with *conditional locks*. When applying protected objects to tasklets from the same Ada task (synchronizing tasklets from different Ada tasks is not allowed), the OpenMP runtime has access to all threads spawned by the Ada task, so OpenMP synchronization mechanisms can be used to implement protected objects. However, when synchronizing two different Ada tasks, the corresponding OpenMP data structures are not shared among Ada tasks, hence they cannot access their respective team of threads. As a result the synchronization must be managed by the Ada runtime, although initiated within the OpenMP runtime. That said, when an OpenMP task accesses a protected object, the Ada runtime is invoked to determine the value of the associated conditional lock. If it is available, the corresponding Ada task

acquires it. If not, the OpenMP task is preempted and placed in the waiting queue of the conditional lock, and the OpenMP thread executing that task is assigned to a different OpenMP task. When the conditional lock becomes available, the Ada runtime must inform the OpenMP runtime, which is in charge of putting the OpenMP tasks associated to that conditional lock back to the ready queue to acquire the lock and continue execution.

Ada task attributes. When executing an OpenMP parallel region (corresponding to either the lowering of Ada parallel code or an OMP `parallel` pragma call), threads must have access to some information of the Ada task (e.g., task identifier). To do so, OpenMP control structures must include information about the Ada task, so any thread in the parallel region can have access to it. Similarly, Ada control structures must include information about OpenMP execution (e.g., the team of threads that is being executed by an Ada task at any point).

6.2. A first step towards an integration between OpenMP and Ada runtimes: Source-code template

The previous sections show how OpenMP can be used to efficiently implement the Ada tasklet model, as well as the benefits of using OpenMP on top of Ada to exploit forms of parallelism that cannot be expressed with Ada tasklets. Furthermore, we analyze what is the interplay needed between the runtime of Ada and that of OpenMP to fulfill compliance with the respective specifications. However, there is not yet an implementation that allows the exchange of information between the two runtimes.

The effort of providing a full integration of OpenMP and Ada may be significant, and remains as a future work. However, the OpenMP and the Ada runtimes still can play together under certain restrictions to deliver valid applications. In this regard, this section presents a novel source-code template that allows Ada programmers to naturally integrate OpenMP into Ada and experiment with the benefits of parallelizing Ada concurrent applications. This template ensures that the Ada and the OpenMP schedulers, are compliant with the respective specifications,

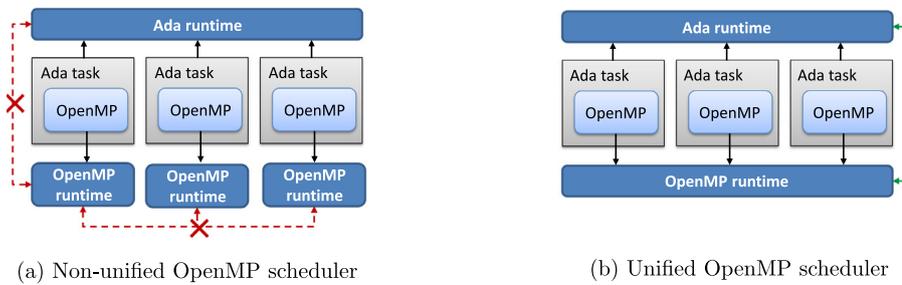


Fig. 8. Interoperability between the Ada and the OpenMP runtimes: Visibility.

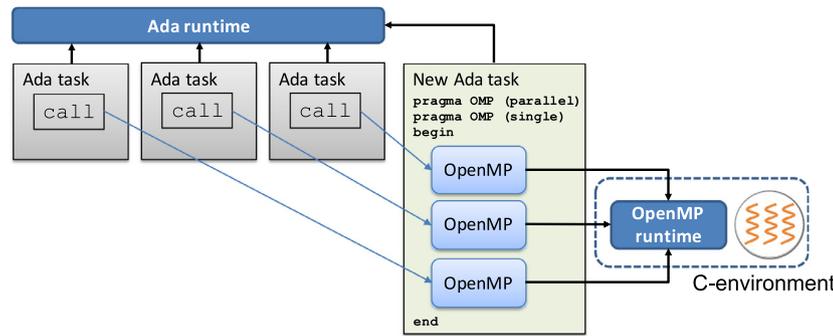


Fig. 9. Schema of the source-code template for Ada and OpenMP light integration.

while it enables the OpenMP runtime to fulfill work-conserving priority-driven policies that match the timing analysis performed at analysis time. This is, in our opinion, a fundamental step towards the full integration of both runtimes.

The next Section (1) present the difficulties of using the Ada concurrent model and the OpenMP parallel model together, (2) introduce the proposed source-code transformation based on a predefined execution template, which is needed for a correct execution of concurrent Ada tasks and parallel OpenMP kernels, and (3) evaluates the correctness and usefulness of the template based on a synthetic use-case that uses Ada and OpenMP.

6.2.1. Scheduler: Need for a unified view

In an Ada application, the Ada scheduler is the component in charge of guaranteeing that the system meets its constraints. These, in real-time systems, are: (1) *Task priorities*, determining the order of execution of ready tasks, (2) a *preemption strategy*, determining when a task can be temporarily interrupted, and (3) an *allocation strategy*, determining the computing resources (cores) in which Ada tasks can execute. For that reason, the scheduler needs to access the complete set of Ada tasks.

When introducing OpenMP into Ada, there are two levels of scheduling: (1) An outer level scheduling composed of Ada tasks, and managed by the Ada runtime, and (2) an inner level scheduling composed of OpenMP tasks, and managed by the OpenMP runtime. The OpenMP scheduler however has limited information when executing a parallel region, particularly, it has access to the OpenMP task scheduling points and OpenMP task priorities of those OpenMP tasks assigned to the current team, but it is oblivious of other concurrent parallel regions, as depicted in Fig. 8a, where each Ada task creates a new OpenMP environment and hence the different OpenMP environments cannot communicate among them, and the Ada runtime cannot communicate with OpenMP because there is no integration in the actual runtimes. This situation may lead to executions where work-conserving and priority-driven strategies cannot be guaranteed.

In this regard, a previous work already states that real-time systems implemented with OpenMP must be composed of a unique OpenMP parallel region [55]. This approach, depicted in Fig. 8b, together with a particular way to instantiate OpenMP tasks from the Ada code (the source-code template presented in the next Section), a unified view of the OpenMP scheduler can be achieved, and the Ada and the OpenMP runtimes can work as if they were integrated.

6.2.2. Source-code template

The objective of the source-code template is to provide an execution environment in which the OpenMP and the Ada runtimes operate as if they were integrated to experiment with the OpenMP tasking model.

The principle behind our proposed source-code transformation is depicted in Fig. 9. The OpenMP parallel code included in the different Ada tasks is centralized into a unique parallel region, so a single team of threads is in charge of managing the complete OpenMP parallel execution [55]. To do so, all OpenMP code is moved to a new Ada task, in which a single parallel region is created, and the OpenMP code in the original Ada tasks is replaced by a call to an entry of this new task. This strategy allows to have a single OpenMP runtime in charge of scheduling all the OpenMP tasks spawned by the different Ada tasks. Moreover, with the objective of guaranteeing that the priorities of the Ada tasks are fulfilled, the OpenMP tasks inherit the same priority of the Ada task that created it (by using the priority clause of the task construct).

Concretely, our proposed source-code transformation to experiment with the OpenMP tasking model considers a set of Ada tasks parallelized using: (1) The `parallel` and `single` constructs to create a parallel region and allow only one thread to execute the inner code, and (2) the `task` construct to distribute work within the parallel region. Note that with this environment, the scheduler will not have a unified view of the system, as multiple OpenMP parallel regions (and so OpenMP runtimes) will exist.

The process to generate the templated program applies the following transformations:

1. Create a new Ada task that implements an entry for each of the parallel regions of the original code. This Ada task creates an OpenMP parallel region with a `single` construct inside (see Fig. 9). Within the single region, a loop accepts calls to the defined entries until no call exists. Each entry implements an OpenMP task that encloses a `taskgroup` construct containing the code inside the OpenMP parallel region of the original Ada task that is now calling the entry. The `taskgroup` ensures that all inner OpenMP tasks finish before the OpenMP task finishes (i.e., it sequentializes different calls to the same entry). This new Ada task is to have the lowest priority in the whole Ada system, so it does not interfere with the original Ada tasks. However, each entry in this task inherits the priority of the caller Ada task.

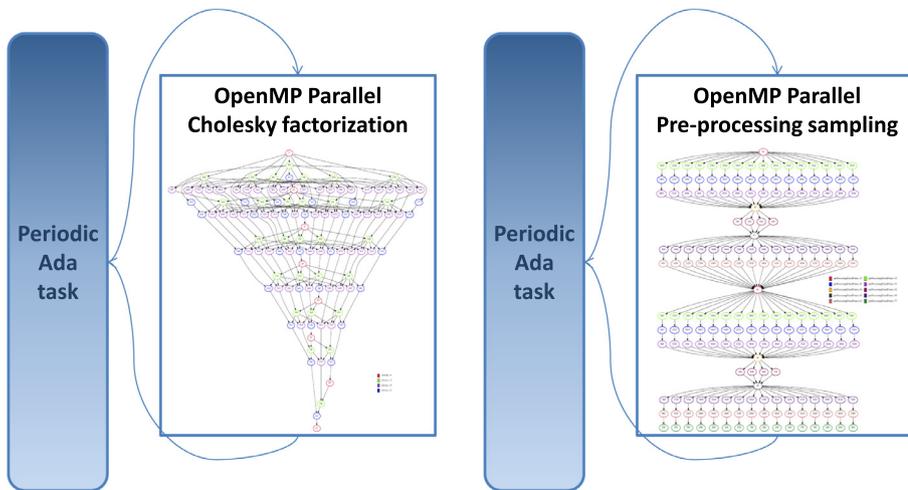


Fig. 10. Example of hybrid Ada/OpenMP application.

2. For each entry in the new Ada task, propagate the priority of each Ada task generating an entry to the OpenMP tasks created within the code of the respective entry.
3. Replace each original parallel region with a call to the corresponding entry of the new Ada task, and include the Ada synchronization mechanism (i.e., a protected object) that allows simulating the implicit barrier at the end of the original parallel region. This means synchronizing the end of the OpenMP task implementing an entry of the new Ada task with the end of the original parallel region generating that entry.

For illustration purposes, we have designed the hybrid Ada/OpenMP system depicted in Fig. 10. The system is composed of: (1) A periodic Ada task with priority 3, *ada_chol*, that generates calls to a Cholesky decomposition implemented using OpenMP tasks, and (2) a periodic Ada task with priority 2, *ada_pps*, that generates calls to an image processing sampling application also parallelized using OpenMP tasks. In order to describe the transformations needed in the application to achieve the templated source code, we show only the *ada_chol* Ada task and the new created Ada task. The original code of this task is shown in Listing 10. The code after the application of the template is shown in Listings 11–13. The same transformation is applicable to the *ada_pps* task as well. Listing 11 shows the new Ada task, *ada_omp*, that generates an OpenMP environment with the `parallel` and `single` constructs, and implements an entry that contains the code that was originally within the parallel environment of the *ada_chol* task. The task is created with the lowest priority in the system, so the tasks defined by the user have higher priority. Finally, a `taskgroup` construct is added to synchronize all tasks generated within the region, so the implicit barrier at the end of the original parallel region is met. After the task group, a call to a protected object synchronizing the end of the execution of the original parallel region with the call to the entry in which the original parallel region has been transformed, as shown in Listing 12. Finally, Listing 13 shows the implementation of the protected object used to synchronize the execution of the Cholesky benchmark between the different Ada tasks involved.

6.3. Evaluation

This section demonstrates the interoperability between the Ada and the OpenMP runtimes accomplished via the source-code templated execution. The following subsections present the application, the complete software and the architectural environment and the execution analysis performed.

6.3.1. Experimental setup

Application. The use case used for this evaluation is the hybrid Ada/OpenMP application introduced in Section 6.2.2. This application

```

1 task Cholesky_Periodic is
2   pragma Priority (2);
3 end Cholesky_Periodic;
4 task body Cholesky_Periodic is begin
5   for i in 1..24 loop
6     pragma OMP (parallel);
7     pragma OMP (single);
8     begin -- OpenMP Cholesky
9       for ... loop
10        pragma OMP (task);
11        potrf;
12        for ... loop
13          pragma OMP (task);
14          trsm;
15          for ... loop
16            for ... loop
17              pragma OMP (task);
18              gemm;
19            end loop;
20            pragma OMP (task);
21            syrk;
22          end loop;
23        end loop;
24      end;
25    end loop;
26 end Cholesky_Periodic;

```

Listing 10. Periodic Ada task implementing a Cholesky factorization using OpenMP tasks.

comprises two Ada periodic tasks that instantiate two benchmarks, a Cholesky decomposition and an image processing algorithm based on a Histogram of Oriented Gradients (HoG), respectively. An illustration of the application after applying the template is shown in Listings 11 and 12.

Runtimes. We use two runtime implementations that support parallelism: 1) GNU libgomp for OpenMP from GCC 7.2 [56], and 2) GNAT Ada from GCC 7.2 [28]. We use the `OMP_NUM_THREADS` environment variable to define the number of OpenMP threads to be used.

Tools. To analyze the execution of the Ada/OpenMP application, we have used two performance tools: (1) *Extræe* [57], a dynamic instrumentation package to trace programs compiled and run with the shared memory model (e.g., OpenMP, Pthreads and OmpSs), the message passing (MPI) programming model or combinations of these two paradigms; and (2) *Paraver* [58], a flexible parallel program visualization and analysis tool based on an easy-to-use wxWidgets GUI that uses the tracing information collected with *Extræe*. These two tools combined are commonly used in HPC studies to analyze the performance of applications qualitatively, thanks to the global perception provided of the

```

1 task OpenMP_Parallel_Task
2   pragma Priority (1);
3 end OpenMP_Parallel_Task;
4 task body OpenMP_Parallel_Task is
5 begin
6   pragma OMP (parallel)
7   pragma OMP (single)
8   begin
9     loop
10      select
11        accept Cholesky do
12          pragma OMP (task);
13          begin
14            pragma OMP (taskgroup);
15            begin -- OpenMP Cholesky
16              ...
17            end;
18            Cholesky_Sync.Release;
19          end;
20        end Cholesky;
21      or
22        accept ImageProcessing do
23          ...
24        end ImageProcessing;
25      or
26        exit;
27      end select;
28    end loop;
29  end
30 end OpenMP_Parallel_Task;

```

Listing 11. Ada task implementing the entry point of OpenMP in the templated source code.

```

1 task body Cholesky_Periodic is begin
2   for i in 1..24 loop
3     OpenMP_Parallel_Task.Cholesky;
4     Cholesky_Sync.Wait;
5   end loop;
6 end Cholesky_Periodic;

```

Listing 12. Transformed periodic Ada task implementing a Cholesky factorization.

```

1 protected body Cholesky_Sync is
2   entry Wait when Open is begin
3     Open := False;
4   end Wait;
5   procedure Release is begin
6     Open := True;
7   end Release;
8 end Cholesky_Sync;

```

Listing 13. Ada protected object to synchronize different releases of a periodic task.

application, and quantitatively, by allowing a microscopic analysis of the specific points of interest.

Platform. The execution takes place on an Intel® Core™ i7-5600U CPU at 2.60 GHz with 2 processors, and 2 hardware threads per processor. The system runs 64-bit Ubuntu 18.04 LTS.

6.4. Execution analysis

To obtain the information we need about the interoperability between Ada and OpenMP, we exploit the fact that both Ada and OpenMP use Pthreads to implement parallelism. In this sense, we use the Extrae library to instrument Pthreads, *libpttrace*.

The trace extracted from the Ada/OpenMP application using the setup introduced in the previous subsection is shown in Fig. 11. There, bluish colors relate to the Cholesky benchmark, and reddish colors re-

late to the image processing benchmark. There are seven rows, each corresponding to one thread created by a call to `pthread_create`. Threads from 1 to 4 are created by the Ada runtime, and they are: thread 1 is created for the Ada main task; thread 2 is created for the Ada task generated by the templated execution to create and manage the OpenMP environment; thread 3 is created for the Ada periodic task calling the Cholesky benchmark; and thread 4 is created for the Ada periodic task calling the image processing benchmark. Additionally, the OpenMP environment is created by thread 2 and contains four threads, as we define with the `OMP_NUM_THREADS` environment variable. Hence, the threads used by the OpenMP runtime are threads 2, 5, 6 and 7.

A detailed analysis of the content of each thread reveals that: (1) Thread 1 executes a tiny portion of time (the Ada main contains a null statement) at the beginning and it is stopped by the Ada runtime doing nothing; (2) thread 2 executes the loop that accepts calls to the Cholesky and the image processing benchmarks until there is no more work to be done; then, it is used by the OpenMP runtime to finish some tasks of the image processing benchmark; finally it calls the ending functions for freeing memory and printing results of the two benchmarks; (3) thread 3 executes the Cholesky initialization corresponding to creating data structures and filling them, and then performs the periodic calls to the Cholesky benchmark; (4) thread 4 executes the image processing initialization, and then the periodic calls to the image processing benchmark; (5) threads 5, 6 and 7 execute the OpenMP tasks corresponding to both the Cholesky and the image processing benchmarks, prioritizing the calls to Cholesky, because the priority of the corresponding Ada task is higher, and this priority is passed to the OpenMP runtime via the templated execution.

There are two important aspects to highlight regarding the interoperability accomplished by means of the templated execution:

1. The Ada and the OpenMP runtimes share the Pthread corresponding to *thread 2*. This thread is first used by the Ada runtime until it reaches the implicit barrier at the end of the parallel region. Then it is used by the OpenMP runtime to finish pending OpenMP work, until all OpenMP threads get to the barrier, when the thread returns to the Ada runtime to finish other Ada work after the OpenMP barrier.
2. The priorities of the Ada tasks are passed to the OpenMP tasks by means of the template, and so the Cholesky tasks, which are the OpenMP tasks with higher priority, run before the image processing tasks to the extent possible (i.e., whenever an entry is accepted and the dependencies are fulfilled).

6.5. Limitations of the source-code template

The objective of the proposed source-code template is to provide Ada programmers with a way to experiment with OpenMP without the need for an actual integration of the Ada and the OpenMP runtimes. This is a major task that remains as future work at this point. As a result, a number of limitations and considerations must be acknowledged:

1. The proposed transformation does not support OpenMP work-sharing constructs (e.g., `for` and `sections`) because the programming model does not consider assigning priorities to threads. Regarding the OpenMP tasking model, it currently only supports the `task` construct because it accepts the `priority` clause, which allows assigning a priority to the task; on the other hand, the tasks created with the `taskloop` construct cannot be assigned with priorities. This, however, requires a minimal implementation, e.g., the compiler could accept a `priority` clause together with the `taskloop` construct, and the runtime could use this information to manage the associated tasks in the corresponding priority queues.
2. The transformation performed by the template might slightly change the order in which OpenMP creates tasks. This is so because in the original supported code, calls to different OpenMP environments can be made concurrently (in the example, the parallel region of the Cholesky benchmark and that of the image processing benchmark

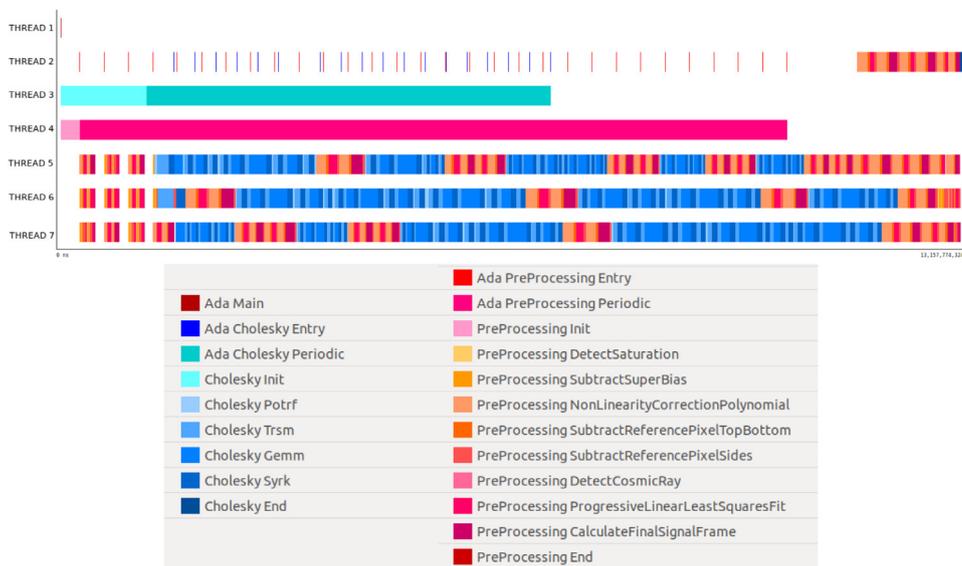


Fig. 11. Execution trace of the Ada/OpenMP application introduced in Section 6.2.2.

are concurrent). After the transformation, calls to the OpenMP environment are sequentialized due to the way task entries are managed (in the example, only one entry of the Cholesky or the image processing benchmarks will be processed at a time). However, since each entry actually creates an OpenMP task with the code within the original OpenMP environment, then several of these can run concurrently as well.

7. Conclusions

This paper tackles the challenge of allowing the use of the OpenMP fine-grained parallel model within the Ada language, by addressing the safety of the code in the presence of parallel computation, and the interoperability of the OpenMP and Ada runtimes. For this, the paper is built upon three main pillars: (1) The programming models syntax and semantics (considering all Ada, the Ada 202X parallel model and OpenMP), (2) the compiler support, and (3) the runtimes' implementation and interoperability. Regarding the first, we introduce a new syntax to use OpenMP in Ada based on a series of experiments that prove the benefits of OpenMP considering performance, programmability and portability, hence productivity. Particularly, we show that OpenMP can be used to implement the Ada 202X parallel model, and can also be used on top of Ada to exploit further forms of parallelism. Regarding the compiler support, we present a series of compiler analysis techniques that can identify potential race conditions in Ada, both considering Ada tasks and parallel OpenMP code. This contribution, together with the limitations and modifications that this work identifies as necessary to be done in the OpenMP specification to be portable to critical real-time systems, bring OpenMP closer to its adoption in safety-critical systems. Finally, regarding the runtimes, this paper extends previous work that analyze the requirements of the integration of the Ada and the OpenMP runtimes, with a novel source-code transformation that enables the OpenMP and the Ada runtimes to operate (under certain restrictions) as they were actually integrated into a unified framework. Furthermore, we use instrumentation and visualization tools to show the accomplished interoperability between the runtimes by virtue of templated execution. Together with previous work, this paper provides a further step to enable the use of the OpenMP fine-grained tasking model, together with, or supporting, the proposed parallel model to be included in the forthcoming revision of the Ada standard.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by the Spanish Ministry of Science and Innovation under contract TIN2015-65316-P, by the European Union's Horizon 2020 Research and Innovation Programme under grant agreements no. 611016 and No 780622, and by the FCT (Portuguese Foundation for Science and Technology) within the CISTER Research Unit (CEC/04234).

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.sysarc.2019.101702 .

References

- [1] S. Royuela, X. Martorell, E. Quiñones, L.M. Pinho, OpenMP tasking model for Ada: safety and correctness, in: Ada-Europe International Conference on Reliable Software Technologies, Springer, 2017, pp. 184–200.
- [2] S. Royuela, X. Martorell, E. Quiñones, L.M. Pinho, Safe parallelism: compiler analysis techniques for Ada and OpenMP, in: Ada-Europe International Conference on Reliable Software Technologies, Springer, 2018, pp. 141–157.
- [3] S. Royuela, L.M. Pinho, E. Quiñones, Converging safety and high-performance domains: Integrating OpenMP into Ada, in: 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE), IEEE, 2018, pp. 1021–1026.
- [4] NVIDIA@Corporation, NVIDIA@CUDA C Programming Guide, 2016, (<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>). [Online; accessed January-2017].
- [5] J.E. Stone, D. Gohara, G. Shi, OpenCL: A parallel programming standard for heterogeneous computing systems, CS&E 12 (3) (2010) 66–73.
- [6] B. Chapman, G. Jost, R. Van Der Pas, Using OpenMP: portable shared memory parallel programming, 10, MIT press, 2008.
- [7] A.L. Varbanescu, P. Hijma, R. Van Nieuwpoort, H. Bal, Towards an effective unified programming model for many-cores, in: IPDPS, IEEE, 2011, pp. 681–692.
- [8] P. Kegel, M. Schellmann, S. Gortlatch, Using OpenMP vs. Threading Building Blocks for Medical Imaging on Multi-Cores, in: EuroPar, Springer, 2009, pp. 654–665.
- [9] S. Lee, S.-J. Min, R. Eigenmann, OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization, SIGPLAN Not. 44 (4) (2009) 101–110.
- [10] J. Shen, J. Fang, H. Sips, A.L. Varbanescu, Performance gaps between OpenMP and OpenCL for multi-core CPUs, in: ICPPW, IEEE, 2012, pp. 116–125.
- [11] G. Krawezik, F. Cappello, Performance comparison of MPI and three OpenMP programming styles on shared memory multiprocessors, in: SPAA, ACM, 2003, pp. 118–127.
- [12] B. Kuhn, P. Petersen, E. O'Toole, OpenMP versus threading in C/C++ +, Concurrency - Practice and Experience 12 (12) (2000) 1165–1176.
- [13] GCC team, GOMP, 2016, (<https://gcc.gnu.org/projects/gomp/>).
- [14] Intel@Corporation, Intel@OpenMP® Runtime Library, 2016, (<https://www.openmp.rtl.org>).
- [15] IBM®, IBM Parallel Environment, 2016, (<http://www-03.ibm.com/systems/power/software/parallel/>).
- [16] OpenMP ARB, OpenMP Application Program Interface, version 2.5, 2005, (<http://www.openmp.org/wp-content/uploads/spec25.pdf>).
- [17] OpenMP ARB, OpenMP Application Program Interface, version 3.0, 2008, (<http://www.openmp.org/wp-content/uploads/spec30.pdf>).

- [18] OpenMP ARB, OpenMP Application Program Interface, version 4.0, 2013, (<http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf>).
- [19] A. Podobas, S. Karlsson, Towards Unifying OpenMP Under the Task-Parallel Paradigm, in: IWOMP, 2016, pp. 116–129.
- [20] OpenMP Architecture Review Board, OpenMP Application Program Interface, version 5.0, 2018, (<https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf>).
- [21] K.L. Sielski, Implementing Ada 83 and Ada 9X Using Solaris Threads, *Ada: Towards Maturity* 6 (1993) 5.
- [22] S. Michell, B. Moore, L.M. Pinho, Tasklettes—a fine grained parallelism for Ada on multicores, in: International Conference on Reliable Software Technologies - Ada-Europe, Springer, 2013, pp. 17–34.
- [23] S.T. Taft, B. Moore, L.M. Pinho, S. Michell, Safe parallel programming in Ada with language extensions, *ACM SIGAda Ada Letters* 34 (3) (2014) 87–96.
- [24] A.R. Group, Ada 202x Language Reference Manual, 2019, (<http://www.ada-auth.org/standards/ada2x.html>).
- [25] L.M. Pinho, B. Moore, S. Michell, S.T. Taft, An Execution Model for Fine-Grained Parallelism in Ada, in: Ada-Europe International Conference on Reliable Software Technologies, 2015.
- [26] IEC, 8652: 2012 Programming Languages and their Environments—Programming Language Ada, International Standards Organization (2012).
- [27] B.J. Moore, Parallelism generics for Ada 2005 and beyond, in: *Ada Letters*, ACM, 2010, pp. 41–52.
- [28] AdaCore, GNAT Users Guide for Native Platform, 2017, (https://gcc.gnu.org/onlinedocs/gnat_ugn.pdf).
- [29] P.M.B.S. Center, Barcelona openmp task suite (bots), 2019, (<https://github.com/bsc-pm/bots>).
- [30] E. Ayguadé, R.M. Badia, P. Bellens, D. Cabrera, A. Duran, R. Ferrer, M. González, F. Igual, D. Jiménez-González, J. Labarta, et al., Extending OpenMP to survive the heterogeneous multi-core era, *International Journal of Parallel Programming* 38 (5–6) (2010) 440–459.
- [31] BSC, MareNostrum IV, 2017, (<https://www.bsc.es/support/MareNostrum4-ug.pdf>).
- [32] G. Tagliavini, D. Cesarini, A. Marongiu, Unleashing fine-grained parallelism on embedded many-core accelerators with lightweight OpenMP tasking, *IEEE Transactions on Parallel and Distributed Systems* 29 (9) (2018) 2150–2163.
- [33] R.E. Vargas, S. Royuela, M.A. Serrano, X. Martorell, E. Quiñones, A lightweight OpenMP4 run-time for embedded systems, in: 2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC), IEEE, 2016, pp. 43–49.
- [34] M.A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna, E. Quiñones, Timing characterization of OpenMP4 tasking model, in: CASES, IEEE Press, 2015, pp. 157–166.
- [35] Intel Interprocedural Optimization, 2017, (<https://software.intel.com/en-us/node/522666>).
- [36] International Organization for Standardization, ISO/DIS 26262. Road Vehicles – Functional Safety, 2009.
- [37] RTCA, DO-178C, Software considerations in airborne systems and equipment certification (2011).
- [38] International Electrotechnical Commission, IEC 61508, Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems, Edition 2.0, 2009.
- [39] S. Royuela, A. Duran, M.A. Serrano, E. Quiñones, X. Martorell, A Functional Safety OpenMP* for Critical Real-Time Embedded Systems, in: International Workshop on OpenMP, Springer, 2017, pp. 231–245.
- [40] OpenMP ARB, OpenMP Application Program Interface, version 4.5, 2015, (<http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf>).
- [41] S. Royuela, A. Duran, C. Liao, D.J. Quinlan, Auto-scoping for OpenMP tasks, in: International Workshop on OpenMP, Springer, 2012, pp. 29–43.
- [42] S. Royuela, R. Ferrer, D. Caballero, X. Martorell, Compiler analysis for OpenMP tasks correctness, in: *Computing Frontiers*, ACM, 2015, p. 7.
- [43] H. Ma, S.R. Diersen, L. Wang, C. Liao, D. Quinlan, Z. Yang, Symbolic analysis of concurrency errors in OpenMP programs, in: *ICPP*, IEEE, 2013, pp. 510–516.
- [44] V. Basupalli, T. Yuki, S. Rajopadhye, A. Morvan, S. Derrien, P. Quinton, D. Wonnacott, ompVerify: polyhedral analysis for the OpenMP programmer, in: International Workshop on OpenMP, Springer, 2011, pp. 37–53.
- [45] D. Kroening, D. Poetzl, P. Schrammel, B. Wachter, Sound static deadlock analysis for C/Pthreads, in: 31st International Conference on Automated Software Engineering, IEEE, 2016, pp. 379–390.
- [46] A. Duran, R. Ferrer, J.J. Costa, M. González, X. Martorell, E. Ayguadé, J. Labarta, A proposal for error handling in OpenMP, *IJPP* 35 (4) (2007) 393–416.
- [47] M. Wong, M. Klemm, A. Duran, T. Mattson, G. Haab, B.R. de Supinski, A. Churbanov, Towards an error model for OpenMP, in: IWOMP, Springer, 2010, pp. 70–82.
- [48] A. Duran, E. Ayguadé, R.M. Badia, J. Labarta, L. Martinell, X. Martorell, J. Planas, ompSs: a proposal for programming heterogeneous multi-core architectures, *Parallel Processing Letters* 21 (02) (2011) 173–193.
- [49] R. Fehete, G. Kienesberger, A Framework for CFG-Based Static Program Analysis of Ada Programs, in: 13th Ada-Europe International Conference on Reliable Software Technologies, Springer, 2008, pp. 130–143.
- [50] S. Royuela, A. Duran, X. Martorell, Compiler automatic discovery of ompSs task dependencies, in: International Workshop on Languages and Compilers for Parallel Computing, Springer, 2012, pp. 234–248.
- [51] E. Lippe, N. van Oosterom, Operation-based Merging, in: Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments, in: SDE 5, ACM, 1992, pp. 78–87.
- [52] A. Burns, B. Dobbing, T. Vardanega, Guide for the use of the Ada Ravenscar Profile in high integrity systems, *ACM SIGAda Ada Letters* 24 (2) (2004) 1–74.
- [53] A. Burns, A.J. Wellings, HRT-HOOD: A structured design method for hard real-time systems, *Real-Time Systems* 6 (1) (1994) 73–114.
- [54] Ada Conformity Assessment Authority, Annex D: Real-Time Systems, 2012.
- [55] M.A. Serrano, S. Royuela, E. Quiñones, Towards an OpenMP Specification for Critical Real-Time Systems, in: International Workshop on OpenMP, Springer, 2018, pp. 143–159.
- [56] GNU, The GOMP project, 2017, (<https://gcc.gnu.org/projects/gomp>).
- [57] BSC, Extrae, 2017a, (<https://tools.bsc.es/extrae>).
- [58] BSC, Paraver, 2017b, (<https://tools.bsc.es/paraver>).



Dr. Sara Royuela got her PhD in Computer Architecture in 2018. She has been working at BSC for the last 9 years, first in the Programming Models department and then in the Parallel Predictable Computing department. She is a compiler expert, particularly in compiler analysis techniques for correctness and safety, with deep knowledge on OpenMP. Sara has participated as BSC member in several European Projects: ENCORE (FP7, '10-'13), PSOCRATES (FP7, '13-'16), CLASS (H2020, '18-'20) and ELASTIC (H2020, '19-'21), providing key contributions on compiler analysis techniques for parallel programming models in the context of critical embedded systems and HPC. She has co-led the efforts to introduce OpenMP into Ada, collaborating with research institutions (CISTER), companies (AdaCore) and organizations (OpenMP ARB and Ada ARG members). She has participated in bilateral projects with the ESA and Denso, and is currently involved in a project with Airbus Defense and Space (ADS) that evaluates the use of OpenMP in different space benchmarks. Her work has been published in international well-recognized conferences. She has been part of the program committee and participated in the organization of a number of conferences. Additionally, she mentors different PhD and Master students from the UPC, the research of which involves functional safety and parallel programming for embedded systems.



Dr. Eduardo Quiñones is a senior researcher at the Barcelona Research Center (BSC). He worked at the Intel Barcelona Research Center from 2002 till 2004. At BSC, he has previous experiences involved in the architectural definition for critical real-time systems in multiple European projects including MERASA, parMERASA, PROARTIS and PROXIMA FP7 projects. Current research interests focus on the applicability of HPC parallel programming models for critical real-time systems to increase performance. This research topic has been developed in the P-SOCRATES FP7 project and in several industrial projects from the automotive and the space domains, including projects with the ESA. He is currently coordinating three H2020 projects (CLASS, ELASTIC, AMPERE) related to the use of HPC distributed programming models in smart city domains and participates in the DeepHealth H2020 project related to distributed deep learning training projects. Eduardo is author of more than 80 scientific papers and holds three patents.



Dr. Luís Miguel Pinho is Professor at the Department of Computer Engineering of the School of Engineering (ISEP), Polytechnic Institute of Porto, Portugal, with a PhD in Electrical and Computer Engineering at the University of Porto, Portugal. He has more than 20 years of experience in research in the area of real-time and embedded systems, particularly in concurrent and parallel programming models, languages, and runtime systems. He is Research Associate in the CISTER research unit, where he was Vice-Director from 2010 to 2017, being responsible for creating several research areas and topics, among which the activities on parallel real-time systems, that he leads. He has participated in more than 25 R&D projects, was Project Coordinator and Technical Manager of the FP7 R&D European Project P-SOCRATES and national-funded CooperatES and Reflect Projects. He has been also coordinator of the participation of ISEP and work package leader in several other international and national projects. He has published more than 100 papers in international conferences and journals in the area of real-time embedded systems, participated in the Organization and Program Committees of several international conferences in the area, and served as guest editor in several scientific journals. He was Senior Researcher of the ArtistDesign NoE and is currently a member of the HIPEAC NoE. He was Keynote Speaker at the 16th IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2010). Among others, he was General Co-Chair of the 28th GI/ITG International Conference on Architecture of Computing Systems (ARCS 2015), and Program Co-Chair of the 24th International Conference on Real-Time Networks and Systems (RTNS 2016) and of the 21st International Conference on Reliable Software Technologies (Ada-Europe 2016).