Converging Safety and High-performance Domains: Integrating OpenMP into Ada

Sara Royuela†, Luis Miguel Pinho‡, Eduardo Quiñones†
†Barcelona Supercomputing Center (BSC), Spain
{*sara.royuela, eduardo.quinones*}*@bsc.es*
‡Polytechnic Institute of Porto (ISEP), Portugal
lmp@isep.ipp.pt

***Abstract*—The use of parallel heterogeneous embedded architectures is needed to implement the level of performance required in advanced safety-critical systems. Hence, there is a demand for using high level parallel programming models capable of efficiently exploiting the performance opportunities.**

In this paper, we evaluate the incorporation of OpenMP, a parallel programming model used in HPC, into Ada, a language spread in safety-critical domains. We demonstrate that the execution model of OpenMP is compatible with the recently proposed Ada *tasklet* model, meant to exploit fine-grain structured parallelism. Moreover, we show the compatibility of the OpenMP and tasklet models, enabling the use of OpenMP directives in Ada to further exploit unstructured parallelism and heterogeneous computation. Finally, we state the safety properties of OpenMP and analyze the interoperability between the OpenMP and Ada runtimes. Overall, we conclude that OpenMP can be effectively incorporated into Ada without jeopardizing its safety properties.

I. Introduction

Parallel computation is fundamental to provide the level of performance needed in the most advanced safety-critical systems, such as autonomous driving or unmanned aerial vehicles. In that regard, the use of parallel heterogeneous architectures rely on the use of parallel programming models to exploit their massively parallel performance capabilities. There is therefore a strong need to integrate these models in the development of safety-critical systems.

Safety and correctness are crucial concepts in languages targeting safety-critical systems. This is the case of Ada [1], a language widely used in critical systems (e.g. avionics or railway) because its design meets the safety requirements: reliability (allow compilers develop correctness techniques to certify algorithms regarding their functional specification) and analyzability (ensure predictability while facilitating analysis).

Recently there has been a significant effort to extend Ada to support fine-grain parallelism, with the objective of taking benefit from parallel architectures while maintaining safety guarantees. As a result, the recently proposed *tasklet model* [2], [3] includes features for exploiting structured parallelism on shared memory architectures.

There is however a need to introduce more advanced parallel programming models into Ada to exploit more complex forms parallelism and heterogeneous architectures. From the variety of languages available, OpenMP has proven many advantages: a) it offers performance and efficiency levels comparable to highly tunable models such as TBB [4] and OpenCL [5], b) it is robust while not locking the software to a specific number of threads compared to low level libraries such as Pthreads [6], and c) the use of OpenMP implies less effort to introduce fine-grained parallelism in Ada instead of implementing the Ada tasklet model. Additionally, recent works demonstrate that OpenMP provides the safety properties required by Ada [7].

Based on that, this paper evaluates the incorporation of OpenMP into Ada in three blocks:

1) A comparison of the Ada tasklet and the OpenMP execution models that states these are compatible. As a result, OpenMP can be used to implement the tasklet model, by automatically transforming Ada parallel constructs into OpenMP directives at compile-time.
2) Prove of the benefit of using raw OpenMP in Ada to exploit both structured and unstructured parallelism, and heterogeneous architectures (this paper leaves the accelerator model as a future work).
3) An analysis of the interoperability needed between OpenMP and Ada runtimes to fulfill the corresponding specifications, without jeopardizing the safety guarantees provided by Ada.

II. Related Work

Ada has long been a very strong language concerning its concurrency model, with the specification of tasks (independent threads of control) at the language level, and a set of language mechanisms for inter-task communication and synchronization. The rationale is that providing language concurrency mechanisms gives the compiler information on the tasking behavior, which allows building safer programs.

Nevertheless, Ada only considers coarse-grain concurrency. Without the tasklet model recently proposed, programmers need to manually craft fine-grain parallelism using Ada tasks, or use user-level libraries to manage parallel computation on top of Ada tasks. Hereof, Paraffin [8], [9] consists of a set of generic Ada libraries that dynamically manage fine-grain parallelism, incorporating mechanisms for parallel blocks, parallel loops and reductions, and recursive parallelism. This library provides parallelism managers following work-sharing, work-stealing and work-seeking approaches, on top of pools of worker tasks. Using Ada generics, it provides a simple interface to create and manage parallel execution, and delivers comparable performance to OpenMP or Cilk on structured parallelism for a small number of cores [10].

As noted, one of the strengths of Ada is that it has been carefully designed to prevent faulty executions. When considering

978-3-9819263-0-9/DATE18/©2018 EDAA

parallel programming, the main issues are data races and deadlocks. In that regard, the Ada tasklet model proposes to palliate the lack of information at compile-time when third-party libraries or external components are used by including two new *aspects* [2]: 1) `Global`, which identifies shared data used within a function, and 2) `Potentially_Blocking`, which identifies functions containing potentially blocking statements. In the same line, there is a proposal to extend OpenMP with new directives, i.e. `globals` and `usage critical`, to allow identifying data races and deadlocks when third-party code is used [11]. These directives cover the lack of safety features in C/C++ and Fortran, but they are not needed when using OpenMP with Ada, as the Ada aspects can be used.

III. Analysis of Ada and OpenMP Parallel Models

This section analyses Ada and OpenMP parallel models, comparing their specification and implementation.

A. Forms of parallelism

Ada tasklets and OpenMP implement a *fork-join* execution model where parallelism is spawned when a *parallel statement* (in Ada) or a *parallel construct* (in OpenMP) is reached, and it is joined at the end of the parallel region. Both models define *execution containers*, named *executor* in Ada and *thread* in OpenMP, and managed by the respective runtimes.

Ada tasklets introduce two new statements to spawn and distribute parallel work among executors: 1) the *parallel block* allows defining several blocks of code that can execute in parallel, and 2) the *parallel loop* denotes that loop iterations can execute in parallel. Both mechanisms define a form of *structured* parallelism. Listings 1 and 2 show the syntax of these new statements.

Listing 1: Parallel blocks Ada syntax

```
1 parallel
2    x := a * a;
3 and
4    y := b * b;
5 end parallel;
6 res := x + y;
```

Listing 2: Parallel loop Ada syntax

```
1 for I in parallel lb..ub loop
2    a[I] := a[I] + b[I];
3 end loop;
```

Unlike Ada, OpenMP does not for the spawning and distribution to be done at the same point. Instead, OpenMP defines the `parallel` construct to spawn work, and several constructs to distribute this work to threads. These can be classified in two different models:

- The *thread-centric* model exploits *structured* parallelism distributing work by means of work-sharing constructs. It provides a fine-grain control of the mapping between work and threads. The most representative constructs are `for` and `sections`.
- The *task-centric* model (*tasking model* henceforward) exploits both *structured* and *unstructured* parallelism distributing work by means of tasking constructs. It provides a higher abstraction level in which threads are fully controlled by the runtime. The most representative constructs are `task` and `taskloop`.

The two models have tantamount performance [12]. Listings 3 and 4 are equivalent to Listings 1 and 2, using the OpenMP tasking model instead of Ada tasklets.

Fig.1 illustrates the flexibility of the OpenMP fork-join model compared to that of Ada tasklets. Due to the separation

Listing 3: Parallel blocks OpenMP tasking model syntax

```
1 pragma OMP (parallel);
2 pragma OMP (single);
3 begin
4   pragma OMP (task);
5      x := a * a;
6   pragma OMP (task);
7      y := b * b;
8 end;
9 res := x + y;
```

Listing 4: Parallel loop OpenMP tasking model syntax

```
1 pragma OMP (taskloop);
2 for I in range lb..ub loop
3    a[I] := a[I] + b[I];
4 end loop;
```

of the spawn and distribution operations, OpenMP allows executing simultaneously several constructs, which can potentially increase parallelism and reduce unnecessary synchronizations (in the Figure, the `taskgroup` avoids the barrier after the first taskloop). Besides, in OpenMP the thread that spawns work may not be the same as the one that distributes it, while in Ada, the sames thread spawns and distributes.

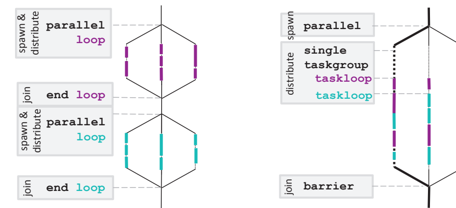


Fig. 1: Execution model of two parallel loops using Ada (left) and OpenMP tasks (right).

B. Execution model

In the Ada tasklet model, the *tasklet* is the unit of parallelism. Tasklets come into existence when the parallel work starts, and terminate at the end of the parallel work.

The Ada execution model is based on a limited form of *run-to-completion*, i.e. tasklets are typically executed by a unique executor, unless they perform an operation that requires blocking[1] or suspension; at these points, the tasklet is allowed to migrate to a different executor. Note that, even if the tasklet does not change executor, it is not mandatory for it to run uninterruptedly or to execute in the same core, since executors may be scheduled in a preemptive scheduler.

The concept of tasklet is very similar to an OpenMP task[2]: a) both are containers that enable fine-grain parallelism, b) the existence of the container is limited to the work it encloses, and c) OpenMP tasks, as Ada tasklets, can be prioritized and preempted. In that regard, OpenMP defines *task scheduling points* (TSPs) as the moments at which a thread can stop executing a specific task and start executing a different one[3]. It is 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. Similarly to Ada, both tied and

[1]Ada blocking operations are: external calls to protected objects, entry calls, Ada task creation or activation, calls to a subprogram containing blocking operations, and `select`, `accept`, `delay` and `abort` statements.

[2]The equivalence between the Ada tasklet and the OpenMP thread-centric model is not straight-forward for two main reasons: 1) OpenMP maps logical concurrent units of work to threads directly, and 2) neither the specification nor the runtime provide any feature for preempting work-sharings.

[3]OpenMP associates *Task Scheduling Points* to different points in a program, e.g. after the generation of an explicit task. The language also defines the directive `taskyield` to explicitly introduce a TSP.

untied tasks are not forced to run uninterruptedly. The main difference between Ada tasklets and OpenMP tasks is that OpenMP allows users to explicitly define tasks whereas in Ada, tasklets are transparent.

C. Use of resources

OpenMP allows programmers to define the amount of computing resources to be used in a parallel region by means of the `num_threads` clause attached to the `parallel` construct. If none is defined, then the number is implementation defined (although the number of cores is commonly considered).

The Ada tasklet model does not yet define whether the programmer can control the number of executors assigned to a parallel region. In this direction, Ada denotes that the parallel execution *progresses* if at least one of the spawned tasklets is being executed by an executor. The tasklet model defines three classes of progression as defined below:

– *Immediate progress*. Ready tasklets can always execute if there are available cores.
– *Eventual progress*. Ready tasklets may require to wait for the availability of an executor even if cores are available, but it is guaranteed that one executor will become available so that the tasklet will eventually be executed.
– *Limited progress*. Ready tasklets may require to wait for the availability of an executor even if cores are available, and it is not guaranteed that one executor will eventually become available. This may happen when there is a limited number of executors and all are blocked.

Runtime implementations must guarantee one class of progress. The two first classes guarantee progression for any program, even if the runtime does not support tasklet migration between executors when tasklets block. The third class requires static analysis to determine the tasks neither starve nor deadlock, and it is suitable when the resources of the program and the runtime structures are statically determined.

The OpenMP specification does not impose any model of progression, as it is responsibility of the programmer to guarantee that the execution does not stall or starve. However, the execution model enables to mimic progression as defined for Ada tasklets (see details in Section IV-B).

D. Memory model

OpenMP and Ada tasklets define a relaxed-consistency memory model where the visibility of the variables may vary within parallel regions. For safety reasons, Ada delegates the responsibility of defining this visibility to the compiler. On the contrary, OpenMP allows three different possibilities: 1) apply the default data-sharing attributes defined in the specification and based in the storage of the variables; 2) manually define the visibility be means of data-scoping clauses (i.e. `shared`, `firstprivate`, `lastprivate` and `private`); and 3) use auto-scoping techniques [13] to automatically determine the visibility based on the usage and liveness of the variables.

IV. Supporting Ada tasklets with OpenMP

This section further analyses the execution model of OpenMP and Ada tasklets, and demonstrates that OpenMP is a firm candidate to implement parallel blocks and loops.

A. Preemption

The limited form of run-to-completion implemented in the tasklet model is mappable to the OpenMP tasking model (see details in Section III-B). The points where a tasklet can be preempted (at blocking or suspension) can be implemented using the OpenMP `taskyield` operation.

Considering the tasking model, untied tasks are more suitable to implement tasklets, because tasks can migrate between threads. Moreover, untied tasks have better time predictability than tied tasks, due to their work-conserving nature [14].

Considering the thread-centric model, the work-sharing constructs can implement the same semantics as Ada parallel blocks and parallel loops do. Despite this, work-sharing entities cannot be preempted by the runtime. Therefore, the thread model is not suitable to support the Ada completion model.

B. Progression Model

The OpenMP specification does not impose any model of progression, however it supports progress as defined for Ada tasklets. Although the OpenMP runtime cannot dynamically modify the number of threads in a team (and therefore it cannot create a new thread when a task blocks), it can move blocked tasks to a waiting queue and reuse threads to execute other tasks. To implement immediate progress, the OpenMP runtime must enforce a work-conserving scheduler, and the number of threads assigned to parallel regions must be bigger or equal than the number of cores. This way, whenever there are resources available, tasks will be scheduled.

OpenMP tied tasks are not suitable to implement immediate progress due to the non-work-conserving nature of the scheduler. Oppositely, these are convenient for eventual progress as long as threads are reused when tasks block. The same happens if the number of threads is smaller than the number of cores.

C. Fork-join Model

The fully strict fork-join model required by the tasklet 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, explicit synchronizations may be needed. This is the case when implementing nested parallelism in Ada. Fig.2a presents a code snippet with nested parallelism using nested Ada parallel blocks, which spawn and distribute twice (at lines 1 and 3). This code can be transformed in two ways using the OpenMP tasking model. The first one, shown in Fig.2b, uses nested parallel regions, which supposes spawning parallelism twice as well (lines 1 and 7). The second one, shown in Fig.2c, uses nested tasks, and supposes spawning parallelism just once (line 1). It needs a taskwait before *code 4* to force the synchronization of the inner block.

The Ada tasklet model does not specify how the runtime manages resources of parallel executions, therefore both transformations are possible. The version shown in Fig.2c may reduce the overhead of creating and destroying an extra team of threads. However, it is interesting to have the possibility of exploiting two different levels of parallelism for those cases where the parallelism is not exposed at the same level, or where there are load balancing problems.

```
1  parallel
2      -- code 1
3      parallel
4          -- code 2
5      and
6          -- code 3
7      end parallel
8      -- code 4
9  and
10     -- code 5
11 end parallel;
```

(a) Ada

```
1  pragma OMP (parallel);
2  pragma OMP (single);
3  begin
4    pragma OMP (task, untied);
5    begin
6        -- code 1
7        pragma OMP (parallel);
8        pragma OMP (single);
9        begin
10           pragma OMP (task, untied);
11               -- code 2
12           pragma OMP (task, untied);
13               -- code 3
14       end;
15       -- code 4
16   end;
17   pragma OMP (task, untied);
18       -- code 5
19 end;
```

(b) OpenMP with nested parallels

```
1  pragma OMP (parallel);
2  pragma OMP (single);
3  begin
4    pragma OMP (task, untied);
5    begin
6        -- code 1
7        pragma OMP (task, untied);
8            -- code 2
9        pragma OMP (task, untied);
10           -- code 3
11       pragma OMP (taskwait);
12       -- code 4
13   end;
14   pragma OMP (task, untied);
15       -- code 5
16 end;
```

(c) OpenMP with nested tasks

Fig. 2: Mapping nested parallelism between Ada and OpenMP

V. Supporting the OpenMP Tasking Model in Ada

Previous sections of this paper propose OpenMP as an implementation for Ada tasklets. This section evaluates the direct use of OpenMP in Ada to increase its parallel features.

In that respect, OpenMP supports point-to-point synchronizations 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 dependences is used to drive the execution. The use of dependences can significantly improve performance of parallel Ada programs, as will be shown in Section VII.

A fundamental requirement of Ada systems is safety, which can be certified at different levels by means of particular standards (e.g. the ISO26262 [15] for automotive, the DO178C [16] for avionics or the IEC61508 [17] for industry). Problems with certification might be due to error-prone features (compromising reliability) or features with complex semantics (complicating analyzability). In that regard, OpenMP has been proven to provide the safety requirements imposed by such systems [7] 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 to enable whole program analysis when third-party components are used. These are similar to the `Global` and `Potentially_Blocking` Ada aspects and needed only to cover the lack of such support in C and Fortran, thus not needed if using Ada.
- Extensions to include error-handling techniques.
- Runtime implementation guidelines to avoid faulty results.

VI. Interoperability of OpenMP and Ada Runtimes

Supporting OpenMP in Ada or using it to implement tasklets requires integrating the OpenMP and Ada runtimes, ensuring their interoperability does not compromise compliance with the respective specifications. This section analyzes three levels of interplay: 1) Ada tasks scheduling, 2) Ada tasks synchronization and 3) Ada and OpenMP control structures.

A. 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 can be preempted in favor of other. If this occurs, the Ada runtime must inform the OpenMP runtime so any parallel execution is stopped, and save the context of the task. However, if only resources are redistributed, then 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 executed in the cores to be stopped reach a *task scheduling point*. Preempted tasks are put back into the *task ready queue* to resume its execution when an OpenMP thread becomes available.

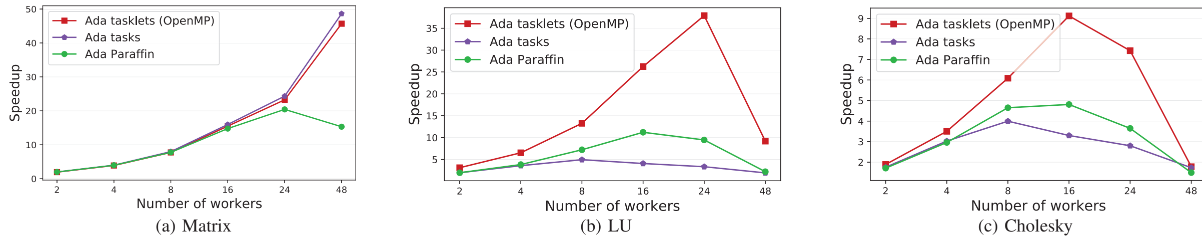
B. Ada Task Synchronization: Protected Objects

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

C. Ada Task Attributes

When executing an OpenMP parallel region (corresponding to either the lowering of an Ada parallel code or a `pragma OMP(parallel)` call), threads must have access to some information of the Ada task (e.g. task id). To do so, OpenMP control structures must include information about the Ada task, so any thread in the parallel region can have access to



(a) Matrix (b) LU (c) Cholesky

Fig. 3: Performance speedup of Ada tasklets (implemented with OpenMP), Ada tasks and Paraffin

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).

VII. Evaluation

This section evaluates the integration of OpenMP into Ada from three different angles: 1) the benefits of OpenMP compared to other implementations that exploit parallelism in Ada, i.e. native Ada tasks [1] and Paraffin [8]; 2) the benefit of OpenMP regardless the base language considered (C and Ada); and 3) the interplay between Ada and OpenMP runtimes.

A. Experimental setup

Runtimes. We use three runtime implementations that support parallelism: 1) the GNU libgomp library for OpenMP from GCC 7.1 [18] 2) the GNAT runtime library for Ada from GCC 7.1 [19], and 3) the Paraffin suit for Ada [8].

Applications. We consider four applications: 1) a matrix intensive computation resembling image processing algorithms (*Matrix*), 2) the LU factorization (*LU*), 3) the Cholesky decomposition (*Cholesky*), 4) a synthetic application that combines several OpenMP constructs and Ada tasks (*Synthetic*). All, the Matrix, LU and Cholesky benchmarks have been parallelized using the Ada tasklet model (implemented with OpenMP), native Ada tasks, Paraffin and C/OpenMP. Additionally, Cholesky has been parallelized using OpenMP task dependences as well, in order to demonstrate the benefits of fully integrating OpenMP into Ada by exploiting unstructured parallelism. Finally, Synthetic is used to demonstrate how Ada and OpenMP runtimes coexist by combining OpenMP tasks called within tasks, and managed by the OpenMP and Ada runtimes respectively.

Platform. We run our experiments in a computing node from the MareNostrum IV [20]. It consists of a 2 sockets Intel Xeon Platinum 8160 CPU with 24 cores each. The processor operates at 2.10GHz, and features a 33MB L3 cache.

Libraries. We use two instrumentation libraries to analyze the correct interoperability between the Ada and OpenMP runtimes: 1) Extrae [21], a tool that gathers information about the performance of parallel applications and generates traces in textual files, and 2) Paraver [22], a performance visualization and analysis tool that uses Extrae traces.

B. Structured parallelism: Ada tasklets, Ada tasks and Paraffin

This section compares the performance speedup of the Ada tasklet model (implemented with OpenMP[4]) with the use of Ada tasks and Paraffin. For such purpose, we use the Matrix, LU and Cholesky benchmarks. Fig.3 shows the speedup obtained for the three benchmarks, considering the three implementations.

[4]There is yet no implementation of the Ada tasklet model.

In the Matrix example (Fig.3a), Ada tasklets and Ada tasks produce equivalent speedups. The regular nature of the algorithm can be efficiently mapped to both Ada and OpenMP tasks. In LU and Cholesky (Fig.s3b and c), Ada tasklets clearly outperform the other implementations because the fine-grain synchronization mechanisms provided by OpenMP are more efficient than the manual mapping of parallelism into Ada tasks and the parallelism management performed by Paraffin. Performance drops down when increasing the number of cores up to 48 because of two reasons: the NUMA effect of the machine, and the small amount of work of the tasks.

C. Unstructured parallelism: Ada tasklets and OpenMP task dependences

OpenMP allows the definition of point-to-point synchronizations to extract parallelism out of highly unstructured applications by means of task dependence clauses.

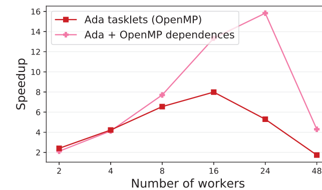


Fig. 4: Speedup of Cholesky using Ada tasklets implemented with OpenMP (structured parallelism) and Ada with raw OpenMP dependences (unstructured parallelism)

Fig.4 shows the results obtained with the Cholesky benchmark parallelized with Ada tasklets, using OpenMP taskwaits to synchronize tasks, and Ada with raw OpenMP, using dependences. The version with dependences 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-grain synchronizations that limit parallelism.

D. Performance benefit of OpenMP: Ada and C

The OpenMP API efficiently supports the development of parallel applications written in C and Fortran. This section proves worth the effort to integrate OpenMP into Ada on account of the performance gains. To this end, the section compares the speedups obtained with C and Ada, and establishes that OpenMP provides tantamount performance, regardless of the differences between the two base languages (evaluating the differences of C and Ada is out of the scope of this paper).

Fig.5 shows the performance obtained for Matrix, LU and Cholesky implemented with Ada and C, using the same

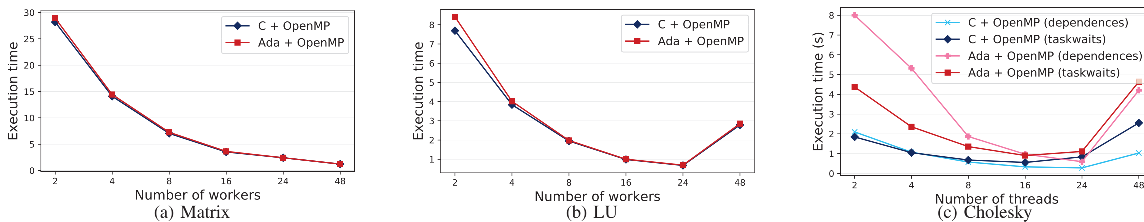


Fig. 5: Execution time of OpenMP running with Ada and C

OpenMP parallelization. Ada scales similarly to C in all cases, proving that OpenMP can be used to satisfactorily exploit parallelism in Ada applications. Furthermore, OpenMP reduces the effect of the differences in the underlying languages (C and Ada) when executed sequentially, delivering a similar execution time for the best parallel versions of both languages.

E. *Interplay of Ada and OpenMP runtimes*

We use a synthetic application to show the coexistence of Ada and OpenMP tasks. The algorithm contains two Ada tasks (one executing periodically every 200ns, and one executing sporadically), and two OpenMP tasks (one performing the intensive computation of Matrix, and one performing light arithmetic computations). OpenMP parallelism is executed within Ada tasks, and the Ada sporadic tasks are released by calling Ada protected objects from within OpenMP tasks.

Fig.6 shows a trace of the execution of this algorithm. The x axis represents time, and the y axis represents available workers. The horizontal bars contain a unit of execution run in a given period in a given worker, where each color represents a different conceptual unit: the Ada sporadic tasks in yellow (executed in threads 1 and 2), the Ada periodic tasks in turquoise (executed periodically in thread 1), the OpenMP heavy tasks in lilac, and the OpenMP light tasks in pink (the last two executed in all threads). The trace shows how Ada and OpenMP tasks share resources and interplay correctly.

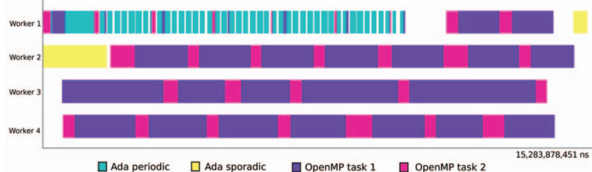


Fig. 6: Execution trace of the Synthetic benchmark mixing OpenMP and Ada tasks

VIII. Conclusion

This paper addresses the integration of OpenMP into Ada, converging the HPC and the safety-critical domains. By comparing the two language specifications, we state that the OpenMP runtime can be used to implement the recently proposed Ada tasklet model, and thus exploit structured fine-grain parallelism in Ada applications. Concretely, we analyze how the OpenMP tasking model, using tied tasks, supports all the preemption model, the progression model and the memory model defined for Ada tasklets.

There are though other implementations that exploit parallelism in Ada, such as Ada tasks and Paraffin. So as to motivate the use of OpenMP to implement tasklets, we compare the three implementations in several benchmarks. Our results show the important benefit obtained with OpenMP, mainly because of the efficiency of its fine-grain synchronization mechanisms in front of those of Ada tasks and Paraffin.

Furthermore, we propose to introduce OpenMP directly into Ada, and thus allow the exploitation of unstructured fine-grain parallelism with the use of task dependence clauses. Our results demonstrate the benefits of this kind of point-to-point synchronization against the use of full barrier synchronizations (implemented with the use of taskwaits).

Overall, and given that previous works already proved that OpenMP keeps Ada safety requirements, we propose OpenMP as a firm candidate to express fine-grain parallelism in Ada.

References

[1] IEC, “8652: 2012 Programming Languages and their Environments– Programming Language Ada,” *International Standards Organization*.
[2] S. T. Taft, B. Moore, L. M. Pinho, and S. Michell, “Safe parallel programming in Ada with language extensions,” *Ada Letters*, 2014.
[3] L. M. Pinho, B. Moore, S. Michell, and S. T. Taft, “An Execution Model for Fine-Grained Parallelism in Ada,” in *Ada-Europe*, 2015.
[4] J. Reinders, *Intel Threading Building Blocks*. O’Reilly & Associates, Inc., 2007.
[5] J. E. Stone, D. Gohara, and G. Shi, “OpenCL: A parallel programming standard for heterogeneous computing systems,” *CS&E*, vol. 12, no. 3, pp. 66–73, 2010.
[6] D. R. Butenhof, *Programming with POSIX threads*. Addison-Wesley Professional, 1997.
[7] S. Royuela, X. Martorell, E. Quinones, and L. M. Pinho, “OpenMP tasking model for Ada: safety and correctness,” in *Ada-Europe*, 2017.
[8] B. J. Moore, “Parallelism generics for Ada 2005 and beyond,” in *Ada Letters*, 2010.
[9] “Paraffin,” 2017. [Online]. Available: http://paraffin.sourceforge.net
[10] B. Moore, “Paraffin: a parallelism api for multiple languages,” *Ada User Journal*, 2016.
[11] S. Royuela, A. Duran, M. A. Serrano, E. Quiñones, and X. Martorell, “A Functional Safety OpenMP for Critical Real-Time Embedded Systems,” in *IWOMP*, 2017.
[12] A. Podobas and S. Karlsson, “Towards Unifying OpenMP Under the Task-Parallel Paradigm,” in *IWOMP*, 2016.
[13] S. Royuela, A. Duran, C. Liao, and D. J. Quinlan, “Auto-scoping for OpenMP tasks,” in *IWOMP*, 2012.
[14] M. A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna, and E. Quinones, “Timing characterization of OpenMP4 tasking model,” in *CASES*, 2015.
[15] *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.
[16] R. DO, “178C,” *Software considerations in airborne systems and equipment certification*, 2011.
[17] *IEC 61508, Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems, Edition 2.0*, 2009.
[18] GNU, “The GOMP project,” 2017. [Online]. Available: https://gcc.gnu.org/projects/gomp
[19] AdaCore, “GNAT User’s Guide for Native Platform,” 2017. [Online]. Available: https://gcc.gnu.org/onlinedocs/gnat_ugn.pdf
[20] BSC, “Marenostrum IV,” 2017. [Online]. Available: https://www.bsc.es/support/MareNostrum4-ug.pdf
[21] ——, “Extrae,” 2017. [Online]. Available: https://tools.bsc.es/extrae
[22] ——, “Paraver,” 2017. [Online]. Available: https://tools.bsc.es/paraver