Real-time Issues in the Ada Parallel Model with OpenMP

Luis Miguel Pinho
Polytechnic Institute of Porto
Portugal
lmp@isep.ipp.pt

Sara Royuela, Eduardo Quiñones
BSC
Spain
{sara.royuela,eduardo.quinones}@bsc.es

Abstract

The current proposal for the next revision of the Ada language considers the possibility to map the language parallel features to an underlying OpenMP runtime. As previously presented, and discussed in previous workshops, the works on fine-grain parallelism in Ada map well to the OpenMP tasking model for parallelism. Nevertheless, and although the general model of integration, and the semantic constructs are already reflected in the proposed revision of the standard, the integration of these new features with the Real-Time Systems Annex of Ada is still not complete. This paper presents an overview of the what is supported and the still open issues.

1 Introduction

The existent proposal to extend Ada with a fine-grained parallelism model is based on the notion of logical threads of control: *A single task, when within the context of a parallel construct, can represent multiple logical threads of control which can proceed in parallel* (LRM 202X, ch. 9) [1]. In this revision of the language, parallel constructs can be found as parallel loops (LRM 5.5) and parallel bocks (LRM 5.6.1), as well as reduction expressions (LRM 4.5.10) and iterators (LRM 5.5.2) [1]. This structured approach to parallelism implements a fully-strict fork-join model, as defined in the tasklet model [2].

In parallel to the development of the Ada language specification, works have been made that demonstrate the suitability of the OpenMP tasking model [1] to support the parallel features of Ada [3], a topic which has been discussed in the last IRTAW workshop [4]. Recently this proposal has gained attraction, and a prototype implementation is validating the possibility [5].

Nevertheless, although the current draft of the Ada 202X standard already includes the parallelism support, and the validation of the prototype implementation, the use of the parallel features together with the Real-Time Systems Annex of Ada is still unclear. In order to allow that the parallel features are used in a real-time application, care must be taken that (i) the language model allows for a correct execution according to the LRM Annex D specifications, and (ii) the OpenMP and Ada runtimes are correctly integrated.

It is already too late to propose changes to the Real-Time Systems Annex to accommodate the use of the fine-grain parallel features, therefore this paper starts a process to analyze how this can be later performed (e.g. via a technical report). It is nevertheless important to determine if it is not necessary to update the annex taking into consideration that a task can now represent multiple logical threads of control. Note that in many places of the standard, the work task is replaced by the logical threads of control, but in Annex D only a note is made (LRM D2.1 4/5) [1].

[1] The term task in OpenMP is not related to Ada tasks, as OpenMP tasks are lightweight parts of the code that can be executed in parallel by worker threads.

On the other hand, the growing demand for high-performance computing in embedded systems (e.g., autonomous driving) is pushing the use of high-level parallel programming models, such as OpenMP, to exploit embedded hardware. For that reason, there is a significant effort in the OpenMP community to extend the use of OpenMP to uses other than High-Performance Computing, such as MPSoCs [6] [7]. In this line, a discussion forum within the OpenMP Architecture Review Board has been made available to tackle the topics regarding real-time systems, considering timing constraints and functional safety.

Overall, the plan is to continue working in the real-time interactions between Ada and OpenMP, in parallel to the real-time OpenMP work, with to propose a technical report to be included in Ada 203X (or before). Note, nevertheless, that the open issues, and potential solutions, are also applicable to non-OpenMP implementations of the Ada parallel model.

2 System Model

The integration of real-time and parallelism in Ada is a complex task, due to the complexity of the Ada concurrency model, and the richness of the real-time features available. Therefore, in the current work we are assuming a simplified Ada runtime, as well as the restriction to use only the OpenMP tasking model (no use of OpenMP threading constructs).

Therefore, the work considers architectures based in both homogenous and heterogenous processors, restricted to:

1. Homogenous processors, with a multicore real-time Operating System (OS), supporting global scheduling, with a single ready queue across processors.
2. Heterogenous processors, with homogenous host processors (using an OS as in 1) and a heterogenous fabric, which is either opaque to OpenMP, and is treated as an independent accelerator (using message passing), or also providing an OpenMP runtime, with a minimal or eventually with no OS.

The plan is to later remove these restrictions, supporting other models, in particular with partitioned scheduling (one fixed-priority scheduler per core, with no thread migration), and EDF and server-based scheduling in the host part of heterogenous processors [2].

3 Current status

The current work on implementing the proposed parallel model for Ada includes the implementation on top of a generalized lightweight parallel model, which can be mapped to OpenMP (or other implementations). Figure 1 shows the high-level architecture of the approach used by AdaCore in their prototype implementation [5]. There, the Ada 202X parallel model is automatically transformed by the compiler into generic calls of an intermediate API implemented in System.Parallelism. This API uses the System.LWT module to initialize and finalize parallel regions, as well as to implement the specific light-weight thread support, currently for OpenMP, a work-stealing approach, and a storage approach (for reducing the heap usage).

[2] To be specified in a “Benicassin” technical report.

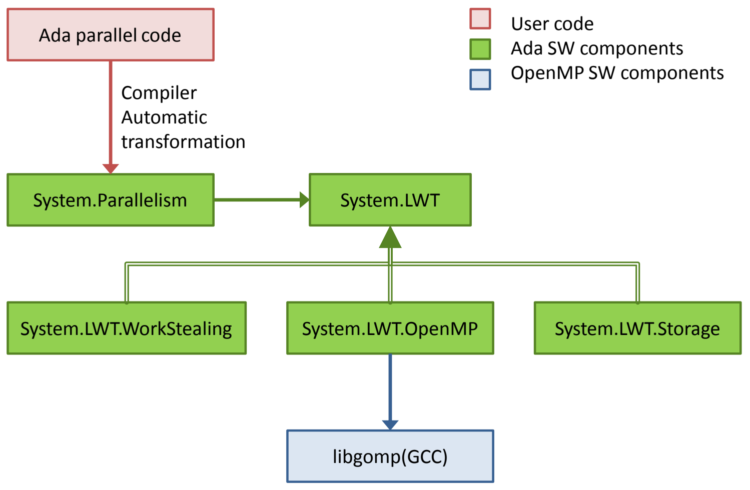

Figure 1. Ada parallel model implementation by AdaCore.

In this approach, Ada tasks with no parallelism are mapped to just one thread, and each Ada task enclosing a parallel region is associated with a thread pool provided by OpenMP to execute the fine-grain parallel constructs. This implementation entails a limitation on the overall scheduling of the application, and this is related to how OpenMP defines parallel regions. In this regard, the team of threads that is associated to each parallel region in OpenMP, constitutes a black box for the rest of teams in OpenMP [8], as well as for the Ada scheduler. This may cause the execution to be non-conforming with specific real-time requirements, such as keeping a work-conserving strategy, or ensuring a priority-driven scheduler. Different solutions have been proposed to tackle this issue, some involving alterations to the OpenMP specification to avoid the black-box nature of OpenMP teams [9] , another enforcing specific implementations of the OpenMP specification regarding the mapping to the underlying resources [10], and finally, and most relevant to this work, another offering a specific templated execution that forces a unique OpenMP team of threads to be accessed from any Ada task [11]. The prototype implementation propagates Ada priorities to the threads created for the OpenMP parallel, which allows to solve this problem.

Additionally, the presented approach assumes that each OpenMP thread is mapped 1-to-1 to operating system threads, which is typically the case (e.g., GCC’s libgomp) since threads are reused only when there is no nested parallelism and between consecutive parallel regions. This mapping allows to simplify analysis and implementation (it makes an Ada task enclosed in a defined set of threads).

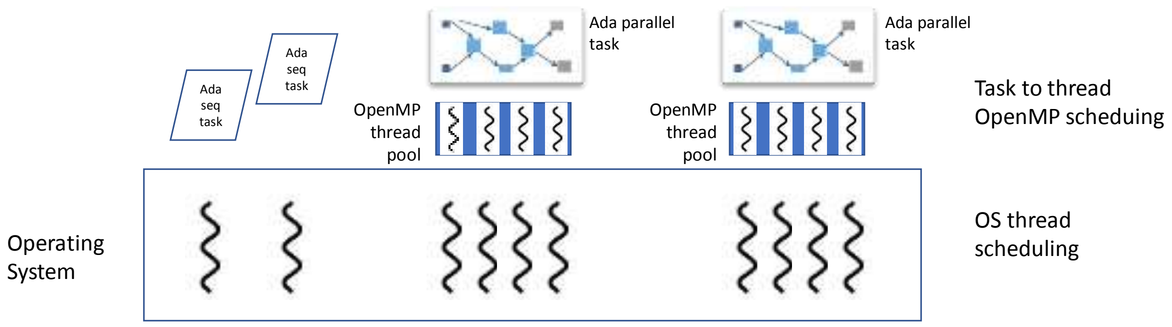

Figure 2. Model of an Ada application with 4 tasks:
2 with no parallel constructs, and 2 with parallel constructs

At the same time, several works have tackled the OpenMP standard considering Real-time restrictions in two directions: time predictability and functional safety. Regarding the former, the suitability of the OpenMP tasking model to derive timing guarantees based on a Task Dependency Graph (TDG) has been proved [12], the timing behavior of both tied and untied tasks has been characterized [13] [14], and the OpenMP specification has been analyzed and provided with augmentations to support the features needed in critical real-time systems. Regarding the latter, the functional safety of the OpenMP 4.5 specification has also been analyzed [15], and several works tackle different aspects such as correctness, including race conditions [16] and deadlocks [17], resiliency [18] and programmability [19] [20].

4 Issues which have been addressed

The impact of fine-grained parallelism in the Ada specification has been discussed for several years, in particular in the context of the IRTAW Workshop [4]. Several of the issues which have been discussed are already addressed in the proposed Ada 2020X standard [1].

Parallelism inside protected actions
One of the recurrent issues was what would happen in parallel code would be executed whilst the Ada task was executing a protected action. This has been addressed, and the solution is to force that all logical threads of control created inside a protected action are executed (in an arbitrary order) by the logical thread of control which is executing the protected action (LRM 9.5.1) [1]. This, in fact, makes the execution sequential.

Potentially blocking operations
The execution of a potentially blocking operation inside parallel code was a subject of intense discussion, and one of the main challenges in supporting fine-grained parallelism. Although a solution was proposed [21], the adopted approach was to disallow these operations, and handle this situation as a bounded error (LRM5.1-18/5) [1]. This makes use of the new Nonblocking aspect in Ada 202X.

Abort and transfer of control
Another issue which was discussed several times was what would happen in the case of task cancelling. The approach (LRM 5.1-16/5 and 17/5) [1] is to attempt to cancel all logical threads of control, and prevent new ones to be

started. Note that the specification allows deferring the cancel further than an abort deferred operation, but not the creation of new parallel constructs. Parallel constructs are not abort-deferred operation, but abort must be made as soon as a abort completion point is reached (LRM 9.8) [1]: for parallel constructs this is where a new logical thread of control would be created or the end of the parallel construct. Both Ada and OpenMP allow for cancelation to be checked in specific predefined points (OpenMP parallel regions can only be cancelled by means of cancellation constructs, i.e., cancel and cancellation point). Hence, in order to be able to stop the parallel computation in an OpenMP region is for the compiler to insert cancellation points in places where it is safe to abort the operation without affecting the correctness of the application.

Shared variables

The parallel access to shared variables is addressed both considering the rules for unsynchronized access to variables (LRM 9.10) [1], and access to Atomic and Volatile objects (LRM C.6) [1]. Ada already had the notion of concurrent activities accessing shared data, which were only necessary to be updated to consider logical threads of control instead of tasks. Note that this also includes access to task attributes, which were already required to use locks (LRM C7.2) [1].

Exceptions

From the point of view of the model presented in this paper, exceptions are in fact handled as a transfer of control issue. If some exception is raised and handled inside the code being executed in one OpenMP thread, then there is no impact, as it is handled inside this thread. If, nevertheless, it is propagated outside of the parallel construct, it is necessary to cancel the execution in the other threads (which is made as noted for the transfer of control). If several exceptions are raised, an arbitrary one can be selected (LRM 5.1 16/5) [1] (the AdaCore implementation selects the "first" that informs the runtime [5]).

5 Real-time systems

Task priorities

The priority model which is specified in Annex D (LRM D.1) [1] specifies that each Ada task is provided with a Priority aspect that defines a degree of urgency, and that, except when explicitly noted, should be used to determine the next task to execute: processors are allocated to the ready task with the highest priority value. OpenMP also has a notion of priority, but that is only used as a hint, and a recommendation for the execution of OpenMP tasks (OpenMP specification, 2.10.1) [22].

Nevertheless, the current model of mapping OpenMP threads 1-to-1 to operating system threads, allows to address this difference, if the priority of the Ada task is propagated to the underlying OS threads. In this case, OpenMP priorities are hidden due to the black-box nature of task-to-thread mapping. All threads from the same thread pool would have the same priority (the priority of the Ada task), thus the semantics of Ada priority would be kept and analysis is possible to guarantee real-time requirements.

It is nevertheless important to note that there are works proposing that OpenMP task priorities are made global to all thread pools, and that priority becomes more than just a recomendation [8]. But in this case, it is possible to propagate the Ada task priorities to OpenMP tasks priorities, thus still maintaining the semantics of Ada priorities.

Dynamic changes to priorities

Priority in Ada may change in two different conditions (LRM D.1 19/3-24) [1]:

- The base priority of a task is changed through a Set_Priority procedure call.
- The active priority of a task changes due to priority inheritance (e.g. when accessing a protected object).

Either changes to the base priority or active priority must be taken into consideration for the execution in a parallel construct. This can happen, for instance, if a task changes its priority (or has its priority changed) with Set_Priority while executing parallel code (thus, actually executing in more than one thread), or if one of the "branches" of the parallel construct executes a call to a protected object with a ceiling priority higher than the current active priority of the task.

For the case of changing the base priority of the task, with Set_Priority, the proposal is to extend the current specification so that the change of the priority is only performed outside of the parallel construct. Currently, in a single processor, Set_Priority needs to be executed as soon as the task is outside of a protected action. For multiprocessors, an implementation needs to document the conditions that may delay the change. Note that this is stronger than the current approach for deferring transfer of control, which cannot be delayed past the creation of new logical threads of control. In this case, it would need to be deferred completely to the end of the parallel construct. This can be specified in the Dynamic priorities section of the LRM (D.5.1).

Priority inheritance is used to guarantee the correctness of access to shared protected objects (other sources of priority inheritance, e.g. task activation or accepting entry calls, cannot exist when executing in a parallel construct). In this case, when accessing a protected object with e ceiling priority higher than its current active priority, the task inherits the ceiling priority of the object. Although this sill needs careful analysis, a potential approach is to change only the active priority of the parallel "branch" executing the protected action, thus allowing a task to have more than one active priority at a time (replacing task by logical thread of control in section D.3 of the LRM, and allowing for multiple active priorities in section D.1). Note that currently a task only has one active priority, therefore if a logical thread of control accesses a protected object, all parallel logical threads need to change.

Other dispatching models

Ada specifies not only preemptive priority-driven scheduling, but other different models:

- Non-preemptive dispatching
- Round-robin dispatching
- Earliest Deadline First (EDF) dispatching

For non-preemptive dispatching, as the Yield procedures are specified as Nonblocking => False, they cannot be called from within a parallel construct. Therefore, parallel code is always non-preemptible [3].

For Round-robin dispatching, the main issue is if a quantum is defined per task, or per logical thread of control. As it is currently specified, it is per task, which means that when the quantum is exhausted, all logical threads of control of that task need to also be moved to the tail of the ready queue. The main issue is how to account for the parallel

[3] Note that it is not explicit, in a multiprocessor setting, if Yield in one of the processors affects that processor only. If global scheduling is being used, there is only one dispatcher, but other processors may be executing non-preemptive tasks. Yield forces a task dispatching point, but it is implicit that only the processor where it is executed is affected.

execution time, and the accuracy of determining it has exhausted. This is similar to execution time timers, therefore the discussion is left for that section.

For EDF dispatching, the same approach as used for priorities can be used. If the underlying operating system supports EDF, the relative deadline of a Ada task needs to be propagated to the OS thread. OpenMP tasks have no deadlines, but again the mapping of OpenMP tasks to threads is treated as a black-box from the OS scheduling point of view.

Moreover, with the addition of parallel capabilities, other dispatching models may be interesting to explore:

- Limited preemptive dispatching: OpenMP executes OpenMP tasks in the threads non-preemptively. This is mainly to reduce preemption overhead, as well as reduce caching issues, as it is assumed that preempting while a computation is taking place increases the risk of invalidating cache lines being used by that computation. Due to that, a specific implementation of a lightweight OpenMP runtime [23] and OS [24] propagates the preemption points of OpenMP tasks to the underlying OS threads so that threads are only preempted at these points.
- For more dynamic parallel real-time systems, server-based scheduling is an interesting approach to support real-time applications. In this case, the need to account for execution budget becomes fundamental.

Execution time timers

Ada Annex D has several capabilities to handle execution time control (D.14), and in particular it allows to set handlers when a task or a group of tasks has used a specific amount of CPU time. When in a parallel setting, a task may be executing in more than one CPU at a time, therefore the implementation of an execution time timer needs to consider the execution time in all CPUs where the task is executing [4], which means that the detection of execution time expiration may not be immediate.

Several different solutions exist (e.g., to account only for execution time in the CPU where the task has set the handler, to disregard execution time accounting for parallel tasks, to consider that the execution time accounting is for each parallel thread independently). Nevertheless, budgets may be required to be per task, and in this case, the execution time needs to accumulate from different CPUs. This can be performed only at specific points in the parallel code (e.g. the same as used for cancelation polling), which means some loss of accuracy.

Asynchronous Task Control and Preemptive Abort

Preemptive Abort (LRM D.6) specifies that in a single processor an abort is completed immediately at the first point that is outside of an abort-deferred operation. This is more restrictive than for the general case (LRM 9.8), but possible in a single processor. For the multiprocessor case, the only requirement is that an implementation document any further delay.

The Asynchronous Task Control (LRM D.11), the behavior is specified in terms of the notion of the *held* priority. Similar to inherited priorities, it is necessary to understand if this reflects in the logical thread of control, or the whole task. The latter case is potentially the preferred one, which means that if it is accepted that a task may have multiple active priorities (see Dynamic changes to priorities above), then the held priority must be made clear as affecting all active priorities of a task.

Multiprocessor dispatching domains

The notion of dispatching domains in used Annex D to specify the set of processors a task may execute. This allows to specify from a fully global scheduling (all tasks may execute in all processors with a single ready queue) to fully

[4] This does not happen for group execution time budgets, since these are per CPU (precisely to avoid the multiprocessor accounting of execution time).

partitioned scheduling (each task is statically assigned to one particular CPU, and each CPU has a separate, even if conceptual, ready queue).

However, when going into a parallel setting, there is another dimension, which is the fact that an Ada task may be executing in several processors, but it may be relevant to disallow migrations between the processors (a computation which starts in a specific core always continues in this core). This model is applicable to the case where the Ada task has a set of threads available for the parallel execution, with each one of these threads pinned to one core. The current support for dispatching domains does not allow such model, since if a task is able to execute in more than one core, there is a single ready queue in the domain, and worker threads may migrate from core to core.

Therefore, a possibility is to extend the current model with a Logical_Thread_of_Control object added to the Assign_Task procedure, which will provide the information on the allocation of underlying threads per CPU, as well as if migration is allowed.

Control of the underlying runtime (number of openmp tasks and worker threads)
Although analysis exists that can extract an extended task dependency graph of a code with OpenMP tasking annotations [25], and that therefore can be applied to Ada parallel code, one important issue for real-time is the possibility to define the number of worker threads of a parallel construct, as well as the number of parallel load in each thread. For the latter, parallel blocks are not an issue (as the number of parallel "braches" is fixed), but in parallel loops it is necessary to specify the amount of parallelism to provide. The current draft of the standard allows to specify the maximum number of chunks (LRM 5.5) [1] of a loop, but not a minimum, or a specific number.

Concerning the former, although not in the draft standard, the current prototype implementation [5] allow to specify the number of threads a task will have available for parallelism. However, this will be a per Ada task number, which will be fixed, and no possibility exists to specify in a particular loop, iterator or block, the actual number of threads to be used.

Acknowledgements

This work was supported by the Spanish Ministry of Science and Innovation under contract TIN2015-65316-P and by the European Union's Horizon 2020 Research and Innovation Programme under grant agreements No 780622 and No 871669.

References

[1] Ada Rapporteur Group, "Ada Reference Manual, 202x Edition, Draft 24," 2020. [Online]. Available: http://www.ada-auth.org/standards/2xrm/html/RM-TTL.html. [Accessed February 2020].

[2] L. M. Pinho, B. Moore and S. Michell, "Parallelism in Ada: status and prospects," in *International Conference on Re-liable Software Technologies – Ada-Europe 2014, LNCS 8454, Springer*, 2014.

[3] S. Royuela, C. Martorell, E. Q. X and L. M. Pinho, "OpenMP tasking model for Ada: safety and correctness," in *22nd International Conference on Reliable Software Technologies (Ada-Europe 2017), pp 184-200. Vienna, Austria,* 2017.

[4] L. M. Pinho and T. Vardanega, "Session Summary: Parallel Programming," in *IRTAW 2018, Ada Lett. 38, 1 (July 2018), 58–60. DOI:https://doi.org/10.1145/3241950.3241960*, 2018.

[5] T. Taft, “Report on Ada 202X light-weight parallelism features,” 2020.

[6] B. Chapman, L. Huang, E. Biscondi, E. Stotzer, A. Shrivastava and A. Gatherer, “Implementing OpenMP on a High-Performance Embedded Multicore MPSoC,” in *International Symposium on Parallel & Distributed Processing,* 2009.

[7] A. Marongiu, P. Burgio and L. Benini, “Supporting OpenMP on a Multi-cluster Embedded MPSoC,” *Microprocessors and Microsystems,* vol. 35, no. 8, pp. 668-682, 2011.

[8] M. A. Serrano, S. Royuela and E. Quiñones, “Towards an OpenMP Specification for Critical Real-Time Systems,” in *International Workshop on OpenMP (IWOMP),* 2018.

[9] M. Garcia, J. Corbalan, R. M. Badia and J. Labarta, “A Dynamic Load Balancing Approach with SMPSuperscalar and MPI,” in *Facing the Multicore-Challenge II,* 2012.

[10] S. Royuela, M. A. G.-G. M. Serrano, S. M. Bellido, J. Labarta and E. Quiñones, “The Cooperative Parallel: A Discussion about Run-time Schedulers for Nested Parallelism,” in *International Workshop on OpenMP (IWOMP),* 2019.

[11] S. Royuela, L. M. Pinho and E. Quiñones, “Enabling Ada and OpenMP Runtimes Interoperability through Template-based Execution,” *Journal of Systems Architecture,* vol. 105, 2020.

[12] R. Vargas, E. Quiñones and A. Maronjiu, “OpenMP and Time Predictability: A Possible Union?,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2015.

[13] M. A. Serra, A. Melani, R. Vargas, A. Marongiu, M. Bertogna and E. Quiñones, “Timing Characterization of OpenMP4 Tasking Model,” in *International Conference on Compilers, Architecture and Synthesis for Embeded Systems (CASES),* 157-166.

[14] J. Sun, N. Guan, Y. Wang, Q. He and W. Yi, “Real-time Scheduling and Analysis of OpenMP Task Systems With Tied Tasks,” in *IEEE Real-Time Systems Symposium (RTSS)*, 2017.

[15] S. Royuela, A. Durán, M. A. Serrano, E. Quiñones and X. Martorell, “A Functional Safety OpenMP* for Critical Real-time Embedded Systems,” in *International Workshop on OpenMP (IWOMP),* 2017.

[16] U. Banerjee, B. Bliss, Z. Ma and P. Petersen, “A Theory of Data Race Detection,” in *Workshop on Parallel and Distributed Systems: Testing and Debugging,* 2006.

[17] D. P. D. S. P. Kroenig and B. Watcher, “Sound Static Analysis for C/Pthreads,” in *IEEE/ACM International Conference on Automated Software Engineering*, 2016.

[18] M. Wong, M. Klemm, A. Duran, T. Mattson, G. Haab, B. R. de Supinski and A. Churbanov, “Towards an Error Model for OpenMP,” in *International Workshop on OpenMP,* 2010.

[19] S. Royuela, A. Duran, C. Liao and D. J. Quinlan, “Auto-scoping for OpenMP Tasks,” in *International Workshop on OpenMP (IWOMP),* 2012.

[20] S. Royuela, A. Durán and X. Martorell, “Compiler Automatic Discovery of OmpSs Task Dependencies,” 2012.

[21] L. M. Pinho, B. Moore, S. Michell and S. T. Taft, “An Execution Model for Fine-Grained Parallelism in Ada,” in *Proceedings of the 20th Ada-Europe International Conference on Reliable Software Technologies, Madrid Spain, June 22-26,* 2015.

[22] OpenMP Architecture Review Board, “OpenMP Application Programming Interface,” 2018.

[A. Marongiu, G. Tagliavini and E. Quiñones, “OpenMP Runtime,” in *High-Performance and Time-*

23] *Predictable Embedded Computing*, 2018.

[24] C. Scordino, E. Guidieri, B. Morelli, A. Marongiu, G. Tagliavini and P. Gai, "Embedded Operating Systems," in *High-Performance and Time-Predictable Embedded Computing*, 2018.

[25] M. A. Serrano, S. Royuela, A. Marongiu and E. Quinones, "Predictable Parallel Programming," in *High-Performance and Time-Predictable Embedded Computing*, 2018.

[26] M. A. Serrano, S. Royuela and E. Quiñones, "Towards an OpenMP Specification for Critical Real-time Systems," in *Internation Workshop on OpenMP (IWOMP)*, 2018.

[27] AdaCore, "GitHub - AdaCore/gnat-llvm: LLVM based GNAT compiler," 2020. [Online]. Available: https://github.com/AdaCore/gnat-llvm.

[28] S. Royuela, R. Ferrer, D. Caballero and X. Martorell, "Compiler analysis for OpenMP tasks correctness," in *International Conference on Computing Frontiers (CF)*, 2015.

[29] B. M. L. M. P. S. Michell, "Tasklettes – a Fine Grained Parallelism for Ada on Multicores," in *International Conference on Reliable Software Technologies – Ada-Europe 2013, LNCS 7896, Springer*, 2013.