The OpenMP API for High Integrity Systems: Moving Responsibility from Users to Vendors

Michael Klemm
michael.klemm@openmp.org
OpenMP ARB
Germany

Eduardo Quiñones
eduardo.quinones@bsc.es
Barcelona Supercomputing Center
Barcelona, Spain

Tucker Taft
taft@adacore.com
AdaCore
Lexington, MA, USA

Dirk Ziegenbein
dirk.ziegenbein@de.bosch.com
Bosch
Renningen, Germany

Sara Royuela[*]
sara.royuela@bsc.es
Barcelona Supercomputing Center
Barcelona, Spain

ABSTRACT

OpenMP is traditionally focused on boosting performance in HPC systems. However, other domains are showing an increasing interest in the use of OpenMP by virtue of key aspects introduced in recent versions of the specification: the tasking model, the accelerator model, and other features like the `requires` and the `assumes` directives, which allow defining certain contracts. One example is the safety-critical embedded domain, where several efforts have been initiated towards the adoption of OpenMP. However, the OpenMP specification states that *"application developers are responsible for correctly using the OpenMP API to produce a conforming program"*, being not acceptable in high integrity systems, where aspects such as reliability and resiliency have to be ensured at different levels of criticality. In this scope, programming languages like Ada propose a different paradigm by exposing fewer features to the user, and leaving the responsibility of safely exploiting the full underlying architecture to the compiler and the runtime systems, instead. The philosophy behind this kind of model is to move the responsibility of producing correct parallel programs from users to vendors.

In this panel, actors from different domains involved in the use of parallel programming models for the development of high-integrity systems share their thoughts about this topic.

CCS CONCEPTS

• Computer systems organization → Embedded and cyber-physical systems; **Real-time systems**; **• Software and its engineering → Compilers**; **System description languages**; **• Computing methodologies → Parallel programming languages**.

KEYWORDS

CPS, Safety, Productivity, OpenMP, Ada

1 PARALLELISM IN HIGH-INTEGRITY SYSTEMS

There is a dramatic increase of the required performance in Cyber-Physical Systems (CPSs) and Real-Time systems, such as those implementing advanced automotive applications. This pushes more demanding designs, which integrate components with multiple levels of criticality into heterogeneous platforms featuring multiple cores and accelerators like GPUs and FPGAs [7]. In this context, the use of parallel programming models to effectively exploit the underlying resources is of paramount importance.

Putting questions about functional safety aside, we can identify the three 'P's that target different aspects of developing software for embedded systems. *Productivity* is an important aspect to consider when integrating a parallel model into a high-integrity system. Equally important are *performance* and *programmability* to achieve the best possible solution.

In addition, the following aspects are relevant to different roles in the (software) product development cycle:

– **High Level:** For the domain expert, it is important to describe the behavior of the system in a deterministic and portable way, decoupling the functional development from the final deployment. For this purpose, the system model design is usually based on Model-Driven Engineering (MDE) techniques that include Domain Specific Modeling Languages (like AMALTHEA [3]). These languages provide an understandable model that matches the specific domain, but they are unaware of the specific parallel Application Program Interface (API) underneath.
– **Middle Level:** At the implementation level, programming languages targeting high-integrity systems, like Ada [5], provide mechanisms for parallelism but leave the orchestration of the parallel execution to the compiler and the runtime. At this level, the compiler has to provide enough intelligence to automatically optimize the code without exposing too many low-level details to the programmer. However, for maximum efficiency these languages should be able to take advantage of low-level APIs if needed. The implementation of the Ada202X parallel model on top of OpenMP [8] is an example of a high-level programming language exploiting the lightweight thread scheduling capabilities of a lower-level API without exposing its unsafe features. Overall, this is a suitable approach for tools that aim at being certifiable at some level.
– **Low Level**: For the performance expert, languages like the OpenMP API [4] expose many features to control the details of the execution while still being easier to apply than other low-level parallel APIs like CUDA and OpenCL. However, these models are typically geared towards High-Performance Computing (HPC), as it is the case of the OpenMP API. As a consequence, it does not (yet) support resilience mechanisms that

[*]Panel moderator.

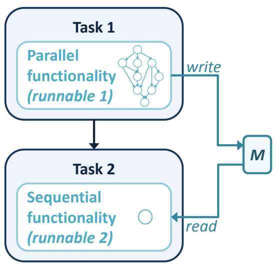
are needed to handle execution errors properly. The latest specifications, however, do include features that can help in the development of safer OpenMP programs, such as the `assumes` and the `requires` directives, allowing the programmer to define certain contracts.

A holistic development environment would be desirable to provide transformations from the highest level to the lowest level. Alas, today, there is a gap between the system description provided at the higher level, and the capabilities provided by current parallel APIs such as the OpenMP API. Research initiatives such as the AMPERE EU H2020 project [6] are exploring the (semi-)automatic transformation of DSMLs to OpenMP directives, in order to orchestrate the parallel execution of CPSs from the automotive and the railway domains in heterogeneous systems, including many-cores, GPUs and FPGAs.

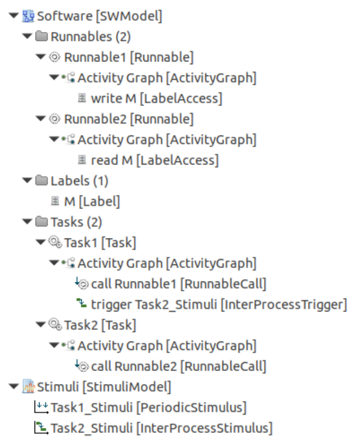
1.1 From the DSML to the parallel API

To illustrate how the tools used in each level of the development cycle are adapted to the specific needs, we use the application design represented in Figure 1a. The application contains two tasks, *Task1* and *Task2*, where *Task2* is triggered by *Task1*. *Task1* is further decomposed in several functionalities that expose parallelism, while *Task2* describes a unique sequential functionality.

A DSML such as AMALTHEA, captures the system description as a series of processes, or *tasks*. Tasks contain an activity graph defining the functionalities, or *runnables*, and synchronizations, such as inter-process events, performed within the task. Finally, runnables can also define accesses to data, or *labels*, among others. Figure 1b shows the AMALTHEA model corresponding to the system description in Figure 1a.


Task 1
Parallel functionality (runnable 1)
write
M
Task 2
Sequential functionality (runnable 2)
read

(a) Application design.



```
▼ Software [SWModel]
  ▼ Runnables (2)
    ▼ Runnable1 [Runnable]
      ▼ Activity Graph [ActivityGraph]
          write M [LabelAccess]
    ▼ Runnable2 [Runnable]
      ▼ Activity Graph [ActivityGraph]
          read M [LabelAccess]
  ▼ Labels (1)
      M [Label]
  ▼ Tasks (2)
    ▼ Task1 [Task]
      ▼ Activity Graph [ActivityGraph]
          call Runnable1 [RunnableCall]
          trigger Task2_Stimuli [InterProcessTrigger]
    ▼ Task2 [Task]
      ▼ Activity Graph [ActivityGraph]
          call Runnable2 [RunnableCall]
▼ Stimuli [StimuliModel]
    Task1_Stimuli [PeriodicStimulus]
    Task2_Stimuli [InterProcessStimulus]
```

(b) AMALTHEA software model.

Figure 1: Application modelling with AMALTHEA

The description level provided by AMALTHEA matches the coarse-grained concurrency features exposed by high-level languages like Ada, by means of tasking and synchronization features. Furthermore, the parallel model proposed for Ada 202X allows also to exploit fine-grained structured parallelism. These capabilities are shown in Figure 2, including an Ada implementation of the model presented in Figure 1b.

```
task body Task1 is
  Next : Calendar.Time := Start;
  procedure runnable1 is
    -- Only structured parallelism
    parallel
    for I in M'Range loop
      ...
    end loop;
  end runnable1;
begin
  delay (Start - Calendar.Clock);
  loop
    runnable1;
    Task2.runnable2;
    Next := Next + Period;
    delay (Next - Calendar.Clock);
  end loop;
end Task1;
```

```
task body Task2 is
  procedure runnable2 is
    -- Sequential execution
    ...
  end runnable2;
begin
  loop
    select
      accept runnable2 do
        ...
      end runnable2;
    or
      terminate;
    end select;
  end loop;
end Task2;
```

Figure 2: Ada coarse- and fine-grained parallelism.

```
// Structured example
void runnable1 () {
  #pragma omp taskloop num_tasks(NT) shared(M)
  for (int i=0; i<Msize; ++i) {
    ...
  }
}

// Unstructured example
void runnable1 () {
  for (int i=0; i<Msize; i++) {
    #pragma omp task depend(inout:M[i]) \
                     shared(M) firstprivate(i)
    ...
  }
  for (int i=0; i<Msize; i++) {
    for (int j=i; j<Msize; j++) {
      #pragma omp task depend(in:M[i]) depend(out:M[j]) \
                       shared(M) firstprivate(i,j)
      ...
    }
  }
}
```

Figure 3: OpenMP fine-grained descriptive parallelism.

Nonetheless, some functionalities may expose dynamic and unstructured behaviors that cannot be represented with the constrained parallel model proposed for Ada. In such cases, the use of flexible APIs like OpenMP allows the definition of more complex parallel structures. Figure 3 shows an example of structured and unstructured parallel functionalities described with the OpenMP tasking model in C. The structured version of *runnable1* mimics the behavior implemented with Ada in Figure 2. On the contrary, the unstructured OpenMP version cannot be implemented with Ada. Furthermore, characteristics like the data-sharing attributes of the variables (i.e., the `shared` and `firstprivate` clauses) or the number of concurrent entities (i.e., the `num_tasks` clause) to be spawned in a parallel loop, can only be defined in OpenMP, while Ada leaves this responsibility in the compiler and the runtime.

2 SUITABILITY OF THE ABSTRACTION LAYERS TO SUPPORT SAFE PARALLELISM

In the recent years, there have been several initiatives to facilitate the development of safety and high-integrity systems targeting parallel architectures.

At the higher level, DSMLs allow describing the system behavior using an easily understandable and deterministic model that fits each specific domain. As an example, the Logical Execution Time

(LET) abstraction has been used as an underlying deterministic model of computation in the DSMLs targeting the automotive domain, as it nicely decouples the functional behavior description from the detailed deployment onto multi-core platforms [10]. Furthermore, the use of automatic code generators transforming the model descriptions into code increases productivity and eases the verification and validation processes in parallel architectures.

At the middle level, SPARK is a well-defined subset of Ada intended for the development of applications demanding safety and security. Interestingly, AdaCore recently released a qualifiable code generator from Simulink to SPARK for formal verification [1]. Although it is not yet supported, the intention of the tool is to incorporate information from the system model level into contracts at the SPARK level, with the objective of enhancing the detection of *data races* and *deadlocks*, two of the most important sources of errors in parallel execution. For detecting data races, contracts include characteristics such as *mode* of access to any global data (input vs. output vs. in-out) as well as *atomicity* of access; for detecting deadlocks, contracts indicate whether an operation is *nonblocking*.

The programming model can provide relevant information to the compiler in order to perform *conflict checking*. Languages such as SPARK are built following this philosophy and, as a result, they are being used in high-integrity systems, including a steer by wire application and NVIDIA firmware modules. However, general programming languages like C and C++ limit the ability of the compiler to perform conflict checking, due to the use of pointers and other complexities. These languages are nonetheless wide-spread in the automotive domain, which uses models like AUTOSAR to represent relevant information about e.g. the task-level parallelism as meta-data, enabling some verification of the system.

The automotive industry is particularly interested in the coarse-grained parallelism at the system-design level. This is because individual components usually cannot be modified as they are legacy code. Nonetheless, it is quite common to reuse components that typically run on accelerator devices. Two major aspects to consider about parallel programming are: a) the productivity of the parallel framework, including its effectiveness in exploiting heterogeneous environments, and b) the capability of the parallel programming model to match the model described at design level.

At the lower level, the OpenMP API is a good candidate to implement automotive software for many reasons, including its tasking and accelerator models, its proven time-predictability, as well as its internal functional safety. However, several features are missing for it to be adopted in high-integrity systems. One reason is that the OpenMP API was never intended to be used in an environment where functional safety at the application level was one of the primary design goals. There have been attempts to include an error model on top of OpenMP [2, 9], but there are restrictions determined by the base languages, i.e., C/C++ and Fortran, on what mechanisms can be used for error handling.

An important challenge when moving the OpenMP API to the embedded domain is to show the clear benefits that compensate the potential risk of losing performance due to the embedded requirements, without conflicting with the use of OpenMP in its primary HPC domain. Fortunately, there is a differentiation between the parallel programming model and its implementation. The Ada parallelism model implemented on top of the OpenMP runtime is defining a subset of the features of the OpenMP API that can be used. Nonetheless, complexity is problematic at any level, and every line of code is another line to prove that is safe. So, for the OpenMP API to be used in safety critical systems, there is a need to identify a subset of OpenMP that is rich enough to be useful and small enough such that it can be certified. This will further allow interoperability and portability across applications and platforms as well as aid composability of software components. The OpenMP specification already contains the foundation to support restricted versions of the language by means of the `requires` (version 5.0) and the `assumes` (version 5.1) directives, among others.

ACKNOWLEDGMENTS

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871669. We would also like to express our gratitude to the organizers of the HILT workshop.

REFERENCES

[1] AdaCore. 2020. QGen. https://www.adacore.com/qgen. (2020).

[2] A. Duran, R. Ferrer, J.J. Costa, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta. 2007. A Proposal for Error Handling in OpenMP. *Intl. Journal of Parallel Programming* 35, 4 (August 2007), 393–416.

[3] Eclipse. 2020. APP4MC. https://www.eclipse.org/app4mc/. (2020).

[4] OpenMP ARB. 2020. OpenMP Application Program Interface v5.1. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf. (2020).

[5] Luís Miguel Pinho, Brad Moore, Stephen Michell, and S Tucker Taft. 2015. An Execution Model for Fine-Grained Parallelism in Ada. In *Ada-Europe International Conference on Reliable Software Technologies*. Springer, 196–211.

[6] Eduardo Quiñones, Sara Royuela, Claudio Scordino, Paolo Gai, Luis Miguel Pinho, Luis Nogueira, Jan Rollo, Tommaso Cucinotta, Alessandro Biondi, Arne Hamann, et al. 2020. The AMPERE Project: A Model-driven development framework for highly Parallel and EneRgy-Efficient computation supporting multi-criteria optimization. In *23rd International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 201–206.

[7] S. Saidi, S. Steinhorst, A. Hamann, D. Ziegenbein, and M. Wolf. 2018. Special Session: Future Automotive Systems Design: Research Challenges and Opportunities. In *2018 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. Torino, Italy. https://doi.org/10.1109/CODESISSS.2018.8525873

[8] S Tucker Taft. 2020. A Layered Mapping of Ada 202X to OpenMP. https://2020.splashcon.org/details/hilt-2020-papers/2/A-Layered-Mapping-of-Ada-202X-to-OpenMP. In *HILT Workshop on Safe Languages and Technologies for Structured and Efficient Parallel and Distributed/Cloud Computing*.

[9] M. Wong, M. Klemm, A. Duran, T. Mattson, G. Haab, B.R. de Supinski, and A. Churbanov. 2010. Towards an Error Model for OpenMP. In *Proceedings of the 6th International Workshop on OpenMP*. Tsukuba, Japan, 70–82. LNCS 6132.

[10] D. Ziegenbein and A. Hamann. 2015. Timing-aware control software design for automotive systems. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. San Francisco, USA. https://doi.org/10.1145/2744769.2747947