CISTER
Research Center in
Real-Time & Embedded
Computing Systems

Journal Paper

Real-Time Support in the Proposal for Fine-Grained Parallelism in Ada

Luis Miguel Pinho

Brad Moore

CISTER-TR-151204

Real-Time Support in the Proposal for Fine-Grained Parallelism in Ada

Luis Miguel Pinho, Brad Moore

CISTER Research Center

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail:

http://www.cister.isep.ipp.pt

Abstract

The Ada language has for long provided supportfor the development of reliable real-time systems, with a modelof computation amenable for real-time analysis. Tocomplement the already existent multiprocessor support in thelanguage, an ongoing effort is underway to extend Ada with afine-grained parallel programming model also suitable for realtime systems. This paper overviews the model which is beingproposed, pointing out the main issues still open and road ahead.

© CISTER Research Center
www.cister.isep.ipp.pt

Real-Time Support in the Proposal for Fine-Grained Parallelism in Ada

Luís Miguel Pinho
CISTER Research Centre
Portugal

Brad Moore
General Dynamics
Canada

Stephen Michell
Maurya Software Inc
Canada

S. Tucker Taft
AdaCore
USA

***Abstract*— The Ada language has for long provided support for the development of reliable real-time systems, with a model of computation amenable for real-time analysis. To complement the already existent multiprocessor support in the language, an ongoing effort is underway to extend Ada with a fine-grained parallel programming model also suitable for real-time systems. This paper overviews the model which is being proposed, pointing out the main issues still open and road ahead.**

Keywords—real-time systems; parallel programming model; Ada

V. INTRODUCTION

In the last years there has been a plethora of work on real-time parallel models, mainly focusing in timing and schedulability analyses concerns. At the same time, very little has been done to integrate fine-grained parallelism and real-time in the software platforms used to deploy such systems. In this paper we focus particularly in real-time programming technologies in general and Ada in particular.

Ada [1] is a language of choice to the development of reliable real-time systems, having for long incorporated models of computation which are amenable for real-time analysis. It has intrinsically the notion of tasks which can be used to model recurrent periodic or sporadic computations. A previous revision of the language (Ada 2005) extended the scheduling model with the provisions for mixing fixed priority, EDF, round-robin and non-preemptive scheduling in an integrated hierarchical scheduling framework [2, 3].

The latest version of the language, Ada 2012, also provides support to multiprocessor system programming, with the ability to represent platforms with multiple cores, and control how tasks are pinned to the cores, with the possibility to support global, partitioned and a multitude of in-between scheduling schemes [4]. Nevertheless, multiprocessor support provides only a coarse-grained concurrent and parallel model, using Ada tasks.

More recently, a proposal was put forward to augment the language with a fine-grained parallel programming model, [5], which can be made amenable for real-time analysis [6,7]. This paper provides an overview of this model, presenting some of the current main open issues for real-time computing.

The paper is structured as follows. The next section presents the parallel programming model proposed for Ada, whilst section III discusses its applicability for real-time systems. Section IV then presents the open issues as well as the current status.

VI. PROPOSED PARALLEL MODEL

The proposal to extend Ada with a fine-grained parallelism model is based on allowing an Ada task to execute a graph of non-schedulable computation units (similar to Cilk [8] or OpenMP [9] tasks), denoted tasklets [5]. Graph edges represent control-flow dependencies. An Ada application can consist of several Ada tasks, each of which can be represented conceptually by a graph.

Tasklets may be explicitly created by the programmer (using proposed new syntax for parallel loops and blocks) as well as implicitly by the compiler. In order to support the compiler a separate proposal is to create contract annotations to [10], to prevent unprotected parallel access to shared variables. An important notion is that the model uses strict fork-join. Tasklets can be spawned by other tasklets (fork), and need to synchronize with the spawning tasklet (join). This restriction, which may be seen as a disadvantage, allows safer programming, as scoping guarantees data access correctness.

A goal of the model is to allow a complete graph of potential parallel execution to be extracted during the compilation phase. Data allocation and contention for hardware resources are key challenges for parallel systems, and therefore compilers and tools must have more information on the dependencies between the parallel computations, as well as data, to be able to generate more efficient programs.

Tasklets are not schedulable per se, and always execute within the semantic context of the enclosing task inheriting the properties of the task such as identification, priority and deadline. The proposed model integrates with the resource sharing protected objects of Ada, therefore synchronization between tasklets of the same or different tasks can be performed using Ada protected operations.

```
task body My_Task is
begin

   -- tasklet A, parent of B, C, F and G,
      ancestor of D and E
   parallel
      -- tasklet B, child of A, parent of D and E
      for I in parallel Some_Rage loop
         -- tasklets D, E, …, are
         -- created by compiler/runtime
      end loop;
   and
      -- tasklet C, child of A, sibling of B
   end;

   -- tasklet A again

   parallel
      -- tasklet F, child of A
   and
      -- tasklet G, child of A
   and
      -- tasklet H, child of A
   end;

   -- tasklet A again
end;
```

Figure 1 – Task body example (not yet Ada)

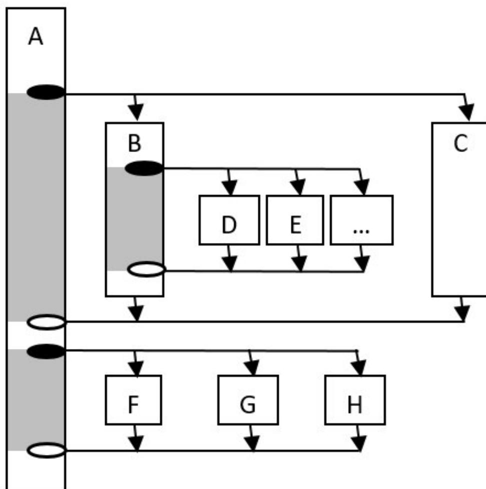

Figure 2 – Task graph example

Figure 1 shows code representing the body of execution of an Ada task (using the parallelism syntax proposal in [10]), whilst Figure 2 provides its associated graph of tasklets (rectangles denote tasklets, dark circles fork points, and white circles join points). The gray area of a tasklet represents tasklet waiting for child joining. The proposed model also specifies the execution behavior of the graphs (as tasklets compete for the finite execution resources), based on a pool of abstract executors (Figure 3), which are required to serve the execution of tasklets while guaranteeing progress [7].

An executor is an entity which is able to carry the execution of code blocks. Although most likely executors will be operating system threads, the definition gives freedom so that other underlying mechanisms can be provided to support this model. An implementation may provide the minimum functionality to execute parallel computation, without requiring the full overhead associated with thread management operation. In an extreme case, an executor can be the core itself, continually executing code blocks placed in a queue.

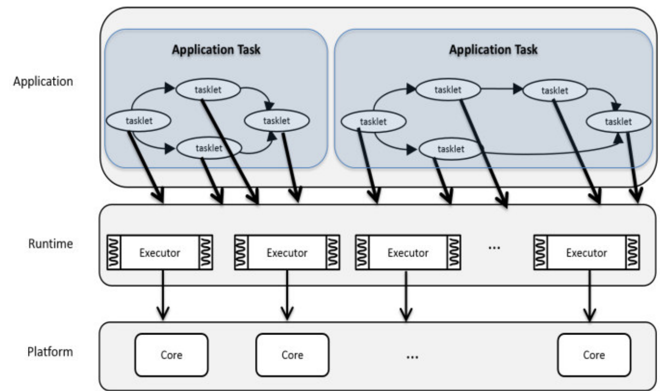

Figure 3 – Execution stack

Tasklet execution by the executors is a limited form of run-to-completion, i.e., when a tasklet starts to be executed by one executor, it is executed by this same executor until the tasklet finishes. Limited because the model allows executing tasklets to migrate to a different executor, but only in the case where the tasklet has performed an operation that would require blocking or suspension. It would be too restrictive to force the executor to also block or suspend. Before starting to execute, tasklet migration is unrestricted.

Note that run-to-completion does not mean that the tasklet will execute uninterruptedly or that it will not dynamically change the core where it is being executed, since the executor itself might be scheduled in a preemptive, or quantum-based scheduler, with global or partitioned scheduling.

The model presumes that the allocation of tasklets to executors, and of executors to cores is left to the implementation [7]. More flexible systems, that are compatible with this model, might decide to implement a dynamic allocation of tasklets to executors, and a flexible scheduling of these in the cores, whilst static approaches might determine an offline-fixed allocation of the tasklets to the executors, and utilize partitioned scheduling approaches for the executors within the cores.

In the general case it is implementation defined whether or not a tasklet, when it blocks, releases the executor. The implementation may also block the executor, creating a new executor, if needed, to serve other tasklets and guarantee the progress of the task, or it may queue the tasklet for later resumption (in the same or different executor).

Note that when a tasklet needs to join with its children (wait for the completion of its children), it is not considered to be blocked, as long as one of its children is executing (forward-progressing). Regardless of the implementation, the executor that was executing the parent tasklet may suspend it and execute one or more of its children, only returning to the parent tasklet when all children have completed.

Furthermore, in a fork-join model it is always safe to suspend a parent tasklet when it forks children, releasing the executor to execute the children tasklets, and resuming the parent tasklet in the same executor when all children tasklets have completed (since the parent can only resume once the children complete). It might happen that other executors take some of the children tasklets. In that case, it might happen that the executor that was executing the parent finishes the execution of children tasklets while other executors are still

executing other children of the same parent. In this case, the parent needs to wait for other children tasklets still being executed in other executors, and the implementation may spin, block or suspend the executor, or re-lease it to execute other unrelated tasklets (as described above).

Implementations may also use some form of parent-stealing [11]. In this case, the suspended parent tasklet might be reallocated to a different executor, or its continuation might be represented by a different tasklet. As before, the implementation must guarantee that tasklet-specific state is also migrated.

VII. Real-time Support

The usual model for real-time programming in Ada is where real-time tasks map one-to-one with Ada tasks. The parallel model follows the same approach, and thus the execution of the Ada task generates a (potentially recurrent) graph of tasklets (on a shared memory multiprocessor). As specified in the model, tasklets run at the priority (and/or with the deadline) of the associated task.

To avoid priority inversion, each Ada task (or priority) is provided with a specific executor pool, where all executors carry the same priority and deadline of the task and share the same budget and quantum. Tasklets run-to-completion in the same executor where they have started execution, although the executor can be preempted by higher-priority (or nearer deadline) executors, or even the same priority/deadline if the task's budget/quantum is exhausted.

This allows for a tasklet graph to be represented (Figure 4) as a Directed Acyclic Graph (DAG) of sub-tasks as commonly used in the real-time systems domain, allowing for the multitude of existent analysis in the community. Moreover, the safety concerns which implied using a strict fork join model have another positive advantage, as allows to use the analysis methods for simpler fork/join and synchronous models.

Each Ada task, and therefore its graph of tasklets, execute within the same dispatching domain, which is a subset of the processors scheduled independently from all other processors. For real-time systems it may be necessary to explicitly control parallelism, since the analysis might need to consider how the parallelism is implemented in greater detail. Therefore, controls are provided [6] for defining, e.g., the number of task executors, the number of tasklets generated in parallel loops, and if executors are allowed to migrate. Together with the dispatching domain model of Ada [4], where platforms can be divided in a set of (disjoint) processor clusters (domains), and Ada tasks allocated to a specific domain (shared or not) this allows sufficient flexibility to support the majority of the real-time models.

The proposed model nevertheless does not support approaches that require setting different priorities/deadlines for individual nodes in the graph (decomposition techniques). In the model, base priorities and deadlines of tasklets remain the same as the parent task, which simplifies creation and scheduling of tasklets, as well as integration with Ada tasking. Note that priority and deadline represent the relative urgency of the job executing; urgency between tasklets of the same graph is not meaningful since it is only the correct and timely completion of the complete graph that matters. Therefore, decomposition techniques must be supported by program restructuring into different Ada tasks.

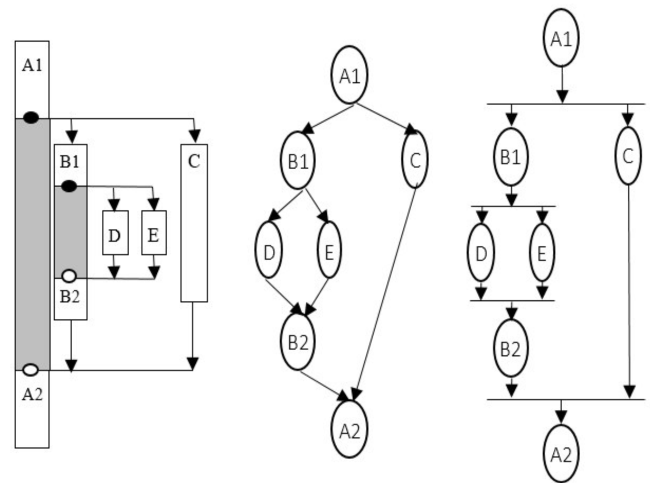

Figure 4 – Representation of a tasklet graph (left) to a general DAG (middle) and synchronous fork-join (right)

To accommodate models where self-suspension is not allowed inside a job (one iteration of the recurrent loop in a real-time task), potentially blocking operations are not allowed when executing in a potentially parallel setting (i.e. if more than one tasklet exists for a given task) and an executor that spawns children tasklets (such as in a parallel block or loop) is required to execute children tasklets, if available, or spin as if executing the parent tasklet.

VIII. Open Issues

The integration of fine-grained parallelism in Ada is a very complex issue, with implications in all mechanisms of the language, from tasks to timing and interrupt handlers, or even exceptions. And supporting real-time computing introduces further complexity. There are therefore a multitude of open issues, which have been recently discussed in a real-time Ada workshop [12]. Of these, several issues impact real-time support, such as:

- **Execution time budget.** In Ada, execution time timers measure the amount of time that a single task (or group of tasks or interrupt routine) uses and notifies a handler if that time is exceeded. Under the current proposal, the execution of a tasklet is reflected in the budget of its task. The overhead of managing the parallel update of the budget may make this unfeasible, except if larger quanta are used or budget updates are not immediate (which may lead to accuracy errors). Specific per core quanta may be used to address this issue.
- **Limited preemption models**. With the introduction of the lightweight tasklet-based programming model (known as task-based programming model in other programming models), it is important to assess if new preemption models are of interest. In particular, the potential small computation effort of tasklets, as well as the fact that potentially variables exist that do not cross the tasklet boundary, it would be possible to implement a model of (limited) preemption only at tasklet boundaries (when tasklets complete). This could eventually reduce overhead and contention, improving efficiency and analyzability.
- **Executor control**. The proposal allows some limited degree of programmer control of tasklets and executors. It is still open if it is possible to explicitly control the

number of executors which are processing a specific parallel region, and if the programmer can use some sort of inter-executor synchronization to control the execution of the tasklets (e.g. by doing computation in phases inside a parallel loop), as this can lead to unsafe computations.

- **Tasklet stealing.** In the real-time model tasklets run-to-completion in the same executor where they have started execution. Therefore, tasklets that have already started cannot be stolen, and parent stealing is disallowed. If executors are not allowed to migrate, any tasklet after starting in a specific core will not leave that core. This is potentially too restrictive.

The hard real-time guarantees of applications executing with the proposed model need to be provided by appropriate timing and schedulability analysis approaches. Although extensive works exist in these topics, and the model described in this paper is fit to be used in these works, it is still not possible to know the feasibility of applying these methods for parallel systems. The complexity and combinatorial explosion of interferences between the parallel executions may prove the timing analysis of parallel computations to be unfeasible. Moreover, the timing analysis requires determinism (and knowledge) of the specific contention mechanisms at the hardware level, something which is more and more difficult to obtain.

This current work allows the compiler, static tools and the underlying runtime to derive statically known tasklet graphs and use this knowledge to guide the mapping and scheduling of parallel computation, reducing the contention at the hardware level. Co-scheduling of communication and computation can further remove contention, and requires knowledge from the application structure. But with the increased complexity and non-determinism of processors, it is not easy to recognize a solution in the near future. For less time-critical firm real-time systems, the model allows for more flexible implementations, using less pessimistic execution time estimates (e.g. measurement-based), and work-conserving scheduling approaches.

Currently no implementation exists of the complete proposal. This work brings to the Ada world models which are widely used in other fine-grained parallelization approaches, where for the general case efficient solutions exist (such as OpenMP or Cilk). At the same time parts of the proposal are implemented in both the ParaSail language [13] and the Paraffin library [14]. The next step needs to be the implementation of the compiler support and executor runtime, even if for a reduced Ada profile.

ACKNOWLEDGMENT

This work was partially supported by General Dynamics, Canada, the Portuguese National Funds through FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within project FCOMP-01-0124-FEDER-037281 (CISTER); by FCT and EU ARTEMIS JU, within project ARTEMIS/0001/2013, JU grant nr. 621429 (EMC2), and European Union Seventh Framework Programme (FP7/2007-2013) grant agreement n° 611016 (P-SOCRATES).

REFERENCES

[1] ISO IEC 8652:2012. Programming Languages and their Environments – Programming Language Ada. International Standards Organization, Geneva, Switzerland, 2012.

[2] A. Burns, A. J. Wellings, "Programming Execution-Time Servers in Ada 2005", Real-Time Systems Symposium - RTSS 2006, Rio de Janeiro, Brazil.

[3] J. A. Pulido, S. Urueña, J. Zamorano, T. Vardanega, J. A. de la Puente, "Hierarchical scheduling with ada 2005", International Conference on Reliable Software Technologies - Ada-Europe 2006, Porto, Portugal.

[4] A. Burns and A. J. Wellings, "Dispatching Domains for Multiprocessor Platforms and their Representation in Ada," International Conference on Reliable Software Technologies - Ada-Europe 2010, Valencia, Spain.

[5] S. Michell, B. Moore, L. M. Pinho, "Tasklettes – a Fine Grained Parallelism for Ada on Multicores". International Conference on Reliable Software Technologies - Ada-Europe 2013, Berlin, Germany.

[6] L. M. Pinho, B. Moore, S. Michell, S. T. Taft, "Real-Time Fine Grained Parallelism in Ada", International Real-Time Ada Workshop – IRTAW 2015, ACM Ada Letters (to appear).

[7] L. M. Pinho, B. Moore, S. Michell, S. T. Taft, "An Execution Model for Fine-Grained Parallelism in Ada", International Conference on Reliable Software Technologies - Ada-Europe 2015, Madrid, Spain.

[8] Intel Corporation, Cilk Plus, https://software.intel.com/en-us/intel-cilk-plus, last Accessed September 2015.

[9] OpenMP Architecture Review Board, "OpenMP Application Program Interface", Version 4.0, July 2013.

[10] T. Taft, B. Moore, L. M. Pinho, S. Michell, "Safe Parallel Programming in Ada with Language Extensions". High-Integrity Language Technologies Conference 2014, Portland, Oregon, USA.

[11] L. M. Pinho, B. Moore, S. Michell, "Session Summary: Fine-grained paralllelism", International Real-Time Ada Workshop – IRTAW 2015, ACM Ada Letters (to appear).

[12] R. D. Blumofe, C. E. Leiserson. "Scheduling multithreaded computations by work stealing". J. ACM, 46:720-748, September 1999.

[13] S. T. Taft, ParaSail – Parallel Specification and Implementation Language, http://parasail-programming-language.blogspot.com, last Accessed September 2015.

[14] B. Moore, Paraffin libraries. http://sourceforge.net/projects/paraffin/, last Accessed September 2015.