An Execution Model for Fine-Grained Parallelism in Ada

Luís Miguel Pinho[1] Brad Moore [2] Stephen Michell [3] S. Tucker Taft [4]

[1] CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto, Portugal, lmp@isep.ipp.pt
[2] General Dynamics, Canada, brad.moore@gdcanada.com
[3] Maurya Software Inc, Canada, stephen.michell@maurya.on.ca
[4] AdaCore, USA, taft@adacore.com

Abstract. This paper extends the authors earlier proposal for providing Ada with support for fine-grained parallelism with an execution model based on the concept of abstract executors, detailing the progress guarantees that these executors must provide and how these can be assured even in the presence of potentially blocking operations. The paper also describes how this execution model can be applied to real-time systems.

1 Introduction

This work is part of an ongoing effort to incorporate fine-grained parallelism models and constructs into existing programming languages, such as CPLEX (for C Parallel Language Extensions) [1], the C++ "Technical Specification for C++ Extensions for Parallelism" [2], or, the topic of this work, the tasklet model for Ada [3,4,5].

The current proposal to extend Ada with a fine-grained parallelism model is based on the notion of tasklets, which are non-schedulable computation units (similar to Cilk [6] or OpenMP [7] tasks). However, in contrast to the C and C++ work, the principle behind this model is that the specification of parallelism is an abstraction that is not fully controlled by the programmer. Instead, parallelism is a notion that is under the control of the compiler and the run-time. The programmer uses special syntax to indicate where parallelism opportunities occur in the code, whilst the compiler and runtime co-operate to provide parallel execution, when possible.

The work in [3] introduced the notion of a Parallelism OPportunity (POP). This is a code fragment or construct that can be executed by processing elements in parallel. This could be a parallel block, parallel iterations of a **for** loop over a structure or container, parallel evaluations of subprogram calls, and so on. That work also introduced the term tasklet to capture the notion of a single execution trace within a POP, which the programmer can express with special syntax, or the compiler can implicitly create.

This model is refined in [4], where each Ada task is seen as an execution graph of execution of multiple control-dependent tasklets using a fork-join model. Tasklets can be spawned by other tasklets (fork), and need to synchronize with the spawning tasklet (join). The concept is that the model allows a complete graph of potential parallel execution to be extracted during the compilation phase. Together with the Global aspects

proposed in [5], it is thus possible to manage the mapping of tasklets and data allocation, as well as prevent unprotected parallel access to shared variables. Although not a topic addressed in this paper, the work considers that issues such as 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 as defined are orthogonal to Ada tasks and execute within the semantic context of the task from which they have been spawned, whilst inheriting the properties of the task such as identification, priority and deadline. The model also specifies that calls by different tasklets of the same task into the same protected object are treated as different calls resulting in distinct protected actions; therefore synchronization between tasklets could be performed using protected operations (in [5] it was restricted to non-blocking operations)[1].

As tasklets compete for the (finite) execution resources, an execution model is then necessary. This paper provides the specification of the execution behavior of tasklets based on the notion of abstract executors, which carry the actual execution of Ada tasks in the platform. The goal of this abstraction is to to provide the ability to specify the progress guarantees that an implementation (compiler and runtime) need to provide to the parallel execution, without constraining how such implementation should be done. This abstraction is then used to demonstrate how tasklet synchronization can be supported using potentially blocking operations.

The paper also shows how the approach can be applied for use in real-time systems (under certain assumptions), and how a finer control of the tasklet execution can be made available to the programmer.

The structure of the paper is as follows. Section 2 provides a short summary of the previously proposed tasklet model for fine-grained parallelization of Ada code. Then, Section 3 proposes the underlying executing model for tasklets, whilst Section 4 shows how this model can be used in real-time systems and mentions some currently open issues. Finally, Section 5 provides some conclusions.

2 The Tasklet Model

This work considers a model where an Ada task is represented as a fork-join Directed Acyclic Graph (DAG) of potentially parallel code block instances (denoted as tasklets). The DAG of a task represents both the set of tasklets of the task, as well as the control-flow dependencies between the executions of the tasklets.

An Ada application can consist of several Ada tasks, each of which can be represented conceptually by a DAG. Therefore, an application might contain multiple (potentially data dependent) DAGs. Dependencies between different DAGs relate to data sharing and synchronization between Ada tasks (e.g. using protected objects, suspension objects, atomic variables, etc.).

---

[1] Note that this is consistent with the current standard which already supports multiple concurrent calls by a single task in the presence of the asynchronous transfer of control capability [8, section 9.7.4].

An Ada task might correspond to several different DAGs, but the actual execution of the task will correspond to only one. Control-flow constructs indicate multiple different paths within the code, each one a different DAG, but during execution a single flow is created.

Within this DAG, a terminology of relation between tasklets is defined, as follows:

- The spawning tasklet is referred to as the *parent* tasklet;
- The spawned tasklets are referred to as the *children*;
- Tasklets spawned by the same tasklet in the same POP are denoted *siblings*;
- *Ancestor* and *descendant* denote the nested hierarchical relationships, since spawned tasklets may themselves spawn new tasklets.

Figure 1 shows code representing the body of execution of an Ada 202X task (according to the syntax proposal in [5]), whilst Figure 2 provides its associated DAG.

```
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
     parallel
        -- D, child of B, descendent of A, sibling of E
     and
        -- E, child of B, descendent of A, sibling of D
     end;
  and
     -- tasklet C, child of A, sibling of B, no relation to D and E
  end;

  -- tasklet A again

  parallel
     -- tasklet F, child of A, no relation to B,C,D and E
  and
     -- tasklet G, child of A, no relation to B,C,D and E
  end;

  -- tasklet A again
end;
```

Figure 1. Task body example (Ada 202X)

This model of tasklet execution is a *fully strict* fork-join, where new tasklets spawned by the execution of a tasklet are required to complete execution before the spawning tasklet is allowed to complete [2]. Note that this model is mandatory for explicit parallel constructs, such as parallel blocks and parallel loops, where the sibling tasklets are required to complete at the parallel construct "**end**" keyword, before the spawning (parent) tasklet is allowed to continue past this construct.

[2] A *terminally* strict fork join for implicit parallelization is left for future work. This is a model where a spawned tasklet is required to join with one ancestor, but not forcibly the parent.

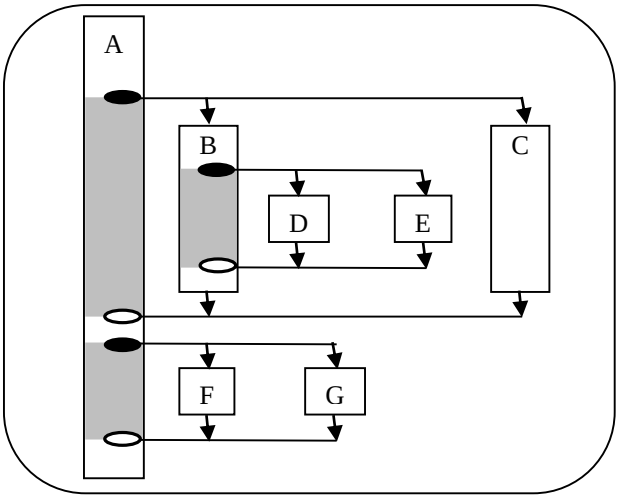

Figure 2. Task DAG example (rectangles denote tasklets, dark circles fork points, and white circles join points)

In Figure 2, although there is no direct relation between F/G and B/C/D/E, the execution of F and G can actually only start after the completion of the latter. Implicit parallelization by the compiler (as proposed in [5]) introduces additional fork-join points (as if an explicit parallel block was being created). The grey parts within the tasklets A and B represent the fact that the tasklet is waiting for the execution of its children. As discussed later in this paper, this does not imply tasklet (or task) blocking.

3 The Tasklet execution model

The proposal of this paper is to define the DAG's execution based on a pool of abstract executors (Figure 3), which are required to serve the execution of tasklets while guaranteeing task *progress*, under certain assumptions.

An *executor* is an entity which is able to carry the execution of code blocks (the notion of an executor entity is common to many systems, and can be traced back to the SunOS lightweight processes [9]). Although we consider that most likely executors would be operating system threads, the definition gives freedom to the implementers to provide other underlying mechanisms to support this model. The justification for this abstraction is that it allows implementations to 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.[3]

[3] Note that an executor cannot explicitly be an Ada Task, as it was proposed in earlier works [10]. This would defeat the separation between the design model of concurrency around Ada Tasks, and the platform model of parallelism around executors.

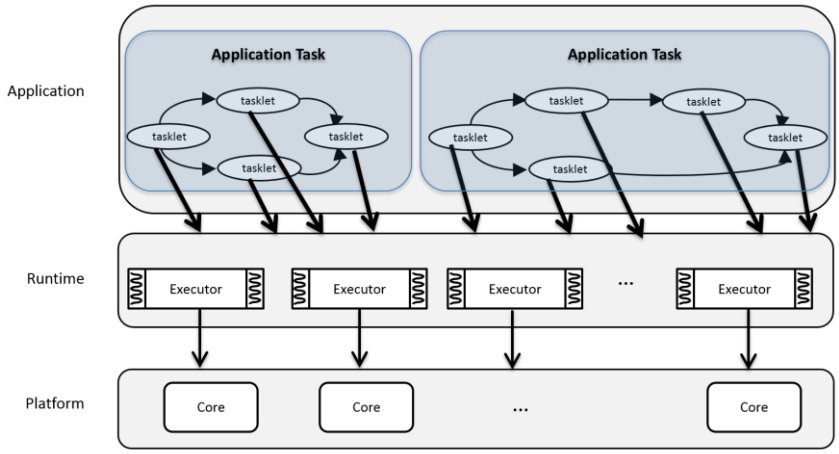

Figure 3. Virtual stack of application, runtime and platform

The model presumes that the allocation of tasklets to executors, and of executors to cores is left to the implementation. 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. Also, in the general case it is left to the implementation whether executor pools are allocated per task or globally to a given dispatching domain or to the entire application.

The model of 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.

3.1 Progress

The progress of a task is defined such that a task progresses if at least one of its tasklets is being executed. Only if all tasklets of a task DAG are not being executed then the task is considered not to be progressing. It might not be blocked, as it might simply be prevented from being executed by other higher priority tasks being executed.

A task is only blocked when all its tasklets are blocked or have self-suspended. Tasklets are considered to be blocked when they are waiting for a resource, which is not an executor nor a core (e.g. executing an entry call) [4].

Considering this, we can identify different progress classes, and use them to define the execution model, rather than the actual implementation [11] (our definition differs from [11] in that we relate progress to tasklet execution and not to thread equivalence). Implementations must guarantee one class of *progress*, as defined below:

- *Immediate progress* – when cores are available, tasklets which are ready to execute can execute to completion in parallel (limited only by the number of free cores);
- *Eventual progress* – when cores are available, ready tasklets might need to wait for the availability of an executor, but it is guaranteed that one will become available so that the tasklet will eventually be executed.
- *Limited progress* – even if cores are available, ready tasklets might need to wait for the availability of an executor, and the runtime does not guarantee that one will be eventually available. This means a bounded number of executors, which may block when tasklets block.

Immediate progress is the strongest progress model, but it might require the implementation to create new executors, even if this would not be needed for progress. The main difference from immediate to eventual progress, is that in the former, if no executor is available, a new executor needs to be created (the implementation needs to be work-conserving). In the latter, the implementation is allowed some freedom, as long as it guarantees that tasklets will not starve.

Limited progress does not guarantee there will be an executor available. It is defined for the cases where a bounded number of executors needs to be pre-determined, and the implementation is such that executors block when tasklets block. In this case, offline static analysis (considering the actual mapping of the tasklet DAG to the underlying executors' implementation) is needed to guarantee tasklets neither starve nor deadlock.

3.2 Use of Potentially Blocking Operations

The work in [5] left as an open issue whether to support potentially blocking operations within tasklets, limiting parallelism to invoking subprograms where the `Potentially_Blocking` aspect was set to `False`. Nevertheless, although the work considers the use of locking in a parallel setting as heavily impacting performance, there are reasons to support (but not recommend) the use of protected operations, such as supporting the need to synchronize "stitching" operations where the results of computations of neighboring regions are combined in the proper order. Moreover, parallel tasks themselves might need to wait for data or delay while within parallel constructs. In any case, lock-free approaches [12] such be considered.

---

[4] There are cases where the compiler is able to guarantee that this blocking is actually bounded (e.g. a delay operation). It may be possible to allow an implementation in this case to consider the tasklet not to be blocked. Nevertheless, in order to be consistent with the Ada standard, these operations are also considered to be potentially blocking [8, section 9.5.1].

The use of blocking synchronization between tasklets introduces further risks of deadlock, which is dependent on the method of tasklet allocation to the underlying executor. For example, the execution of the code in Figure 4 is safe if all iterations are executed in real parallelism, or with some interleaving of iterations, but will deadlock if all iterations of the loop are sequential and executed in order. The code in Figure 5 might appear to be safe for any order of iterations, but in fact it is not safe if the compiler aggregates several iterations in the same tasklet [5]. If, for instance, in a platform with two cores, the compiler generates a DAG of 2 tasklets to execute the loop, both tasklets will call the entry `Wait` in their first iteration, and will wait there indefinitely.

```
protected Obj is
  entry Wait;
  procedure Release;
private
  Open: Boolean := False;
end Obj;
protected body Obj is
  entry Wait when Open is
  begin
     null;
  end Wait;
  procedure Release is
  begin
     Open := True;
  end Release;
end Obj;

-- ...
begin
  for I in parallel 1..10 loop
     -- Phase 1 Work
     if I < 10 then
        Obj.Wait;
     else
        Obj.Release;
     end if;
     -- Phase 2 Work
  end loop;
end;
```

Figure 4. Deadlock example

[5] "Chunking" of several iterations in the same tasklet is an optimization which is usually done to reduce the parallelism overhead when each iteration in isolation is computationally small,

```
protected Obj is
  entry Wait;
private
  Open: Boolean := False;
end Obj;
protected body Obj is
  entry Wait when Wait'Count = 10 or Open is
  begin
     Open := True;
  end Wait;
end Obj;

-- ...
begin
  for I in parallel 1..10 loop
     -- Phase 1 Work
     Obj.Wait;
     -- Phase 2 Work
  end loop;
end;
```

Figure 5. Deadlock example

It is possible for tasklets to synchronize with protected operations (tasklets are the "caller" as specified in the standard [8, section 9.5]), as long as the following conditions are satisfied: (i) the implementation guarantees that all tasklets will eventually be allowed to execute, and (ii) the implementation ensures that the behavior is as if each call to a potentially blocking operation was allocated to a single tasklet.

Condition (i) is satisfied in implementations that provide immediate or eventual progress. In the limited progress model, condition (i) needs to be satisfied by complementing the implementation with offline analysis.

Condition (ii) is satisfied by the compiler that generates an individual tasklet whenever a call could be performed to a potentially blocking operation [8, section 9.5.1] or to a subprogram which has the `Potentially_Blocking` aspect set to `True` (note that according to the rules specified in [5] this is the default value of the aspect). As tasklets can be logical entities only, implementations may provide other means to guarantee (ii), as long as the behavior is equivalent.

3.3 Implementation Issues

The model does not stipulate how an implementation provides these guarantees. Nevertheless, this section discusses a few usage approaches, knowing that experience of use may lead to more efficient mechanisms to implement the model.

The implementation may allocate multiple tasklets to the same executor, allowing these tasklets to be executed sequentially by the executor (under a run-to-completion model). As soon as a tasklet starts to be executed by a specific executor it continues to

be executed by this executor, until it completes, or blocks. It is nevertheless possible that tasklets that are ready and that have not yet begun execution (but queued for one executor), to be re-allocated (e.g. with work-stealing [13]) to a different executor.

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). In the latter case, the implementation releases the executor to execute other tasklets but maintain the state of the blocked tasklet in the executor, for later resumption.

In the immediate and eventual progress models, the implementation must allow a blocked tasklet to resume execution, either by allocating it to an existing or new executor as soon as the tasklet is released, or by resuming it in the original executor (if it is available). In the case of resuming in a different executor than the one that started the computation, the implementation must guarantee that any tasklet-specific state that is saved by the executor is migrated to the new 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.

Note that 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 release it to execute other unrelated tasklets (as described above).

Implementations may also use some form of parent-stealing [13]. 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.

3.4 Tasklets and protected actions

When executing in protected actions, it is possible to allow tasklets to spawn new tasklets, if we are able to guarantee that deadlock will not arise from different executors accessing the same locks.

This is possible if: (i) the executor that is executing the parent tasklet (which owns the lock of the protected action) is only allowed to execute children tasklets, suspending or spinning if none are available for execution (this guarantees that the same executor will not acquire another non-nested lock); and (ii) fully-strict fork-join is used thus all nested children tasklets need to join with the ancestor that entered the protected action,

before it leaves the protected action; and (iii) executors for children tasklets inherit the lock from the executor executing the parent, and do not acquire it again.

Note that protected operations are supposed to be very short. The time needed to spawn tasklets might exceed the recommended time inside a protected operation.

3.5 Use of atomics and programmer specific synchronization

If the programmer uses atomic variables or some specific synchronization code outside of the Ada synchronization features, then no guarantees can be provided as is the case for the use of these mechanisms in the presence of concurrent Ada tasks.

4 Real-time

4.1 Model

We propose a model of real-time parallel programming where real-time tasks map one-to-one with Ada tasks. The execution of the Ada task generates a (potentially recurrent) DAG of tasklets, running on a shared memory multiprocessor. [6]

The use of enhanced parallel programming models such as this one proposed for Ada, will allow for the compiler (with optional parameters/annotations provided by the programmer) to generate the task graphs (similar to the Parallel Control Flow Graphs [14]) which can be used to perform the required schedulability analysis [15].

As specified in the model [4] and presented in the introduction, tasklets run at the priority (and/or with the deadline) of the associated task. We consider that 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.

The executors and the underlying runtime guarantee progress as defined in section 3, and if only limited progress is available, offline analysis is able to determine the minimum number of executors required for each task.

Each task, and therefore its DAG of tasklets, execute within the same dispatching domain [8, section D.16.1]. A dispatching domain is a subset of the processors that is scheduled independently from all other processors. Henceforth we focus on a single dispatching domain, and when we talk of *global* scheduling we mean that the (on-line) scheduling (dispatching) algorithm allows any given tasklet of a task to be scheduled

[6] We recognize that it is difficult if not impossible to scale shared memory to hundreds or thousands of cores. One possibility is to address scalability through clustering, where cores are divided into shared memory clusters (current architectures use 4/8/16 cores per cluster), with clusters communicating through a network on chip. Scalability is dependent on data placement: if the problem is such that the compiler can partition the data into the local memory of each core, the number of cores can actually increase. The communication can be explicit or transparent (depending on design decisions, and the availability of tools to partition the data set and create the required communication patterns).

on any processor within the task's dispatching domain, while fully *partitioned* scheduling means that the on-line scheduling algorithm relies on tasklets being pre-assigned to individual processors by some off-line analysis. We also can consider intermediary strategies where some tasklet migration is permitted, but not necessarily sufficient to ensure an absence of priority inversion within the domain. We consider part of being a *global* scheduling approach that there is no priority inversion within the domain: at any given time, there is never a tasklet running on a processor in the dispatching domain if there are tasklets of tasks with a higher priority (or earlier deadline) awaiting execution.

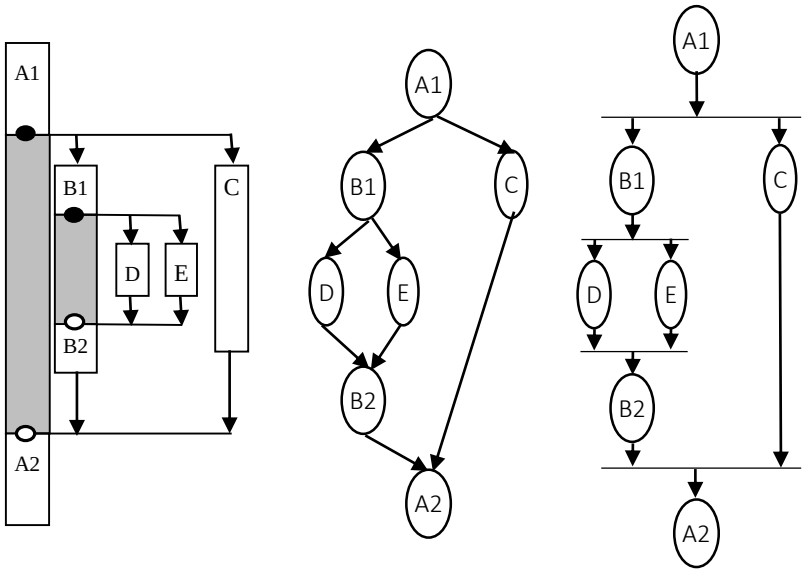

Figure 6. Mapping of tasklet DAG (left) to a general DAG (middle) and synchronous fork-join (right)

This model allows for the tasklet DAG to be converted (Figure 6) to a DAG of subtasks (Figure 6 middle) as commonly used in the real-time systems domain [16, 17, 18]. Response-time analysis techniques can also be used [19, 20] (restricted to non-nested tasklets), as the tasklet model is actually more restricted than the general real-time DAG model, as it considers a fully-strict fork-join, thus a synchronous model can be used [18] (Figure 6 right).

It is important to note that the challenges of providing high-reliability systems with real-time guarantees in the presence of parallelized execution are still far from being solved. When the number of processors increases, it is no longer possible to separate schedulability and timing analyses. Execution time is highly dependent on the interference on accessing hardware resources (e.g. memory) from other parallel tasks, which is dependent on how these are mapped and scheduled. The analysis becomes potentially unfeasible or extreme pessimism must be incorporated.

The work in this paper recognizes this challenge. The path of the work is to allow 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 [21] 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.

We note finally that, as described in section 2, an Ada task might potentially generate several different DAGs (actual execution will only generate one), and therefore the offline schedulability analysis needs to take into consideration all potential DAGs. Works that consider the general workload of a graph can still be used, by taking the maximum workload and critical path length among all the potential graphs (with a less optimal result), or approaches that use a single tighter worst-case graph can be considered [22], which allow one to reduce the complexity of the interference analysis.

4.2 Blocking issues and real-time

Common real-time models assume that tasks execute an infinite number of iterations of the same code (each iteration being called a *job*), with each release of the code being performed with specific time intervals (cyclic or periodic tasks), or triggered by an event (aperiodic or sporadic). It is also common that blocking (or voluntary-suspension) is not allowed inside the iteration of the loop. Blocking calls (such as delay-until or entry calls) are only allowed at each iteration start, to suspend the task until release.

For such cases, parallel execution should follow the same rules as sequential, and parallel code should not block or self-suspend during execution. However, as shown in section 3, the notion of blocking in the parallel model is not as straightforward as in sequential. In particular, calling a closed entry, which blocks a task in a sequential setting, might not block the task in a parallel setting if the call is executed in the context of a tasklet when other tasklets of the same task are progressing in parallel.

Another issue is that spawning tasklets might cause the executor of the parent tasklet to need to wait for some of its children tasklets to finish before being able to proceed. Although this is not considered blocking (see section 3), it is nevertheless a voluntary suspension of execution.

Therefore, in order to simplify, we propose that for these systems, the following additional rules apply: (i) potentially blocking operations are not allowed when executing in a potentially parallel setting (i.e. if more than one tasklet exists); and (ii) 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.

4.3 How to control parallelization

For the general case, the compiler is assumed to have the ability to make the best decisions on how to manage the parallelism associated with each POP. For real-time systems however, it may be necessary to allow the programmer to have more control of

the parallelism, since the analysis might need to consider how the parallelism is implemented in greater detail. Certain types of analysis might not work well with the default choices made by the compiler, but by giving more control to the programmer, the programmer can guide the compiler to produce an implementation that supports the best available analysis methods, in particular for more restricted models.

Table 1 summarizes a set of controls for the programmer to fine tune the parallelism, and control its implementation, while Figure 7 presents examples of their use.

Table 1. Parallelism Controls

Control	Interpretation	Typical Specification
`Executors` aspect of task or task type	Restricts number of executors for task or task type, unless Unbounded is specified	Number of cores plus number of tasklets that might undergo unbounded blocking
`Max_Executors` parameter for dispatching domain `Create` operation	Restricts total number of executors that may be allocated to the domain; sum of `Executors` aspects must not exceed this value	Sum of `Executors` aspects for tasks in domain
`Potentially_Unbounded_Blocking` aspect of subprogram	When False, disallows untimed, unconditional entry calls; defaults to same value as Potentially_Blocking aspect	Specifying False allows a tasklet to perform delays but not potentially unbounded entry calls.
`No_Executor_Migration` restriction	Disallows executor migration from one processor to the next during its execution	Can be specified in a Restrictions pragma to restrict scheduling mechanisms
`Tasklet_Count` aspect of a discrete subtype, an array type, an iterator type, or an iterable container type	Limits number of tasklets that may be spawned in a single iteration	Typically the number of processors, or the number of executors for the enclosing task.
`No_Implicit_Parallelism` restriction	Restricts the compiler from inserting parallelism at places not explicitly identified as **parallel**	Can be specified in a Restrictions pragma to ensure tasklets are only spawned at programmer-defined points
`No_Nested_Parallelism` restriction	Restricts the programmer from using nested explicitly **parallel** constructs, and disallows the compiler from implicitly inserting such constructs	Can be specified in a Restrictions pragma to ensure tasklet DAGs remain one-level structures, to simplify analysis

```
task My_Task with Executors => 4;

function Create (First, Last   : CPU;
                 Max_Executors : Natural) return Dispatching_Domain;
…
My_Domain : Dispatching_Domain := Create (1, 4, Max_Executors => 8);

procedure P with Potentially_Blocking => True,
                 Potentially_Unbounded_Blocking => False;

subtype Loop_Iterator is Natural range 1 .. 1000
  with Tasklet_Count => 10;
…
for I in parallel Loop_Iterator loop
    Array (I) := Array (I) + I;
end loop;

pragma Restrictions (No_Implicit_Parallelism, No_Nested_Parallelism);
```

Figure 7. Examples of Parallelism Controls

4.4 Other Real-Time issues

4.4.1 Mixed Priorities and Per-Task Deadlines

There are approaches that require setting different priorities/deadlines for parallel computation (e.g. some decomposition methods [18]), but the model considers all tasklets to inherit the priority/deadline of the Ada task that contains the POP. It both simplifies the creation and scheduling of tasklets (all tasklets share all attributes of the parent task, including ID and priority), and allow for priority and deadline to represent the relative urgency of the job executing. If priority/deadline boosting is required, it is only the executor that is actually affected that will have this change.

4.4.2 Changing Task Priority and other task attributes

Under the rules proposed above, when a tasklet in a DAG executes `Set_Priority`, it is the base priority of the parent task and all tasklets of that task currently executing that are changed. Under Ada rules [8, section D.5.1], this change happens as soon as the task is outside a protected action. Care should be taken that calls to change a priority or deadline are executed by only a single tasklet, and ideally when it is the only active tasklet. It is likely an error to let multiple tasklets change the priority or deadline, especially if with different values. The same applies to changing other task attributes.

4.4.3 Timing Events

A timing event [8, section D.15] is handled by a protected object, with a protected procedure called if the event occurs, and a protected procedure or entry used to handle the event. Care is needed to ensure that the presence of multiple tasklets does not result in multiple event creations, nor in multiple tasklets attempting to handle the same event.

4.4.4 Execution Time Timers

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 our

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.

5 Conclusion

This paper presented an execution model for the execution of tasklet graphs (previously proposed for fine-grained parallelism in Ada), based on the abstract notion of executors, which allows reasoning about the execution model in terms of progress guarantees, rather than on the actual implementation. The paper also shows how this model can be used for real-time systems, complementing the model with a proposal for mechanisms the programmer can use to explicitly specify the parallel behavior.

Although no implementation exists of the complete proposal, parallelism only makes sense to provide faster computation. 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. For the case of real-time systems, addressing parallelism is still a challenge, and more research and experimentation is needed.

Acknowledgements

The authors would like to thank Ted Baker and the anonymous reviewers for the valuable comments and suggestions. 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) and ref. FCOMP-01-0124-FEDER-020447 (REGAIN); 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] CPLEX, C Parallel Language EXtensions study group, archives at http://www.open-std.org/mailman/listinfo/cplex, last Accessed March 2015.

[2] Working Draft, "Technical Specification for C++ Extensions for Parallelism", available at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3960.pdf, last Accessed March 2015.

[3] 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, LNCS 7896, Springer, 2013.

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

[5] T. Taft, B. Moore, L. M. Pinho, S. Michell, “Safe Parallel Programming in Ada with Language Extensions”. High-Integrity Language Technologies Conference, October 2014.

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

[7] OpenMP Architecture Review Board, “OpenMP Application Program Interface”, Version 4.0, July 2013.

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

[9] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D.Williams, “Beyond Multiprocessing: Multithreading the SunOS Kernel”. Proceedings of the Summer USENIX Conference, June 1992.

[10] B. Moore, S. Michell and L. M. Pinho, “Parallelism in Ada: General Model and Ravenscar”. 16th International Real-Time Ada Workshop, April 2013.

[11] T. Riegel, “Light-Weight Execution Agents”, October 2014, available at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4156.pdf, last accessed March 2015.

[12] G. Bosch, “Lock-free protected types for real-time Ada”. 16th International Real-Time Ada Workshop, April 2013.

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

[14] L. Huang, D. Eachempati, M. W. Hervey, B. Chapman, “Extending Global Optimizations in the OpenUH Compiler for OpenMP”. Open64 Workshop at CGO, 2008.

[15] L. M. Pinho, E. Quiñones, M. Bertogna, A. Marongiu, J. Carlos, C. Scordino, M. Ramponi, “P-SOCRATES: a Parallel Software Framework for Time-Critical Many-Core Systems". Euromicro Conference on Digital System Design, August 2014.

[16] S. Baruah, “Improved multiprocessor global schedulability analysis of sporadic DAG task systems”. 26th Euromicro Conference on Real-Time Systems, July 2014.

[17] J. Li, J. J. Chen, K. Agrawal, C. Lu, C. Gill, and A. Saifullah. "Analysis of Federated and Global Scheduling for Parallel Real-Time Tasks". 26th Euromicro Conference on Real-Time Systems, July 2014.

[18] A. Saifullah, J. Li, K. Agrawal, C. Lu, and C. Gill. "Multi-core real-time scheduling for generalized parallel task models". Real-Time Systems Vol.49 No.4, July 2013.

[19] C. Maia, M. Bertogna, L. Nogueira, and L. M. Pinho. "Response-time analysis of synchronous parallel tasks in multiprocessor systems". 22nd International Conference on Real-Time Networks and Systems, October 2014.

[20] P. Axer, S. Quinton, M. Neukirchner, R. Ernst, B. Dobel, and H. Hartig. "Response-time analysis of parallel fork-join workloads with real-time constraints". 25th Euromicro Conference on Real-Time Systems, July 2013.

[21] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, R. Kegley, “A Predictable Execution Model for COTS-based Embedded Systems”, 17th IEEE Real-Time and Embedded Technology and Applications Symposium, April 2011.

[22] J. Fonseca, V. Nélis, G. Raravi, L. M. Pinho, "A Multi-DAG Model for Real-Time Parallel Applications with Conditional Execution", 30th ACM Symposium on Applied Computing, April 2015.