Real-Time 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

The approach for fine-grained parallelism in Ada presented at the last Real-Time Ada Workshop has been revised, both in terms of the language mechanisms to specify parallelism, as well as in terms of the underlying execution model. This paper summarizes the current state of the proposal, further detailing how programmers can control the behavior of the parallel execution, and discussing the issues which are still open.

Note: To ease understanding, this submitted version of the paper includes more than a summary, repeating in particular the main contributions proposed in [3,4] [1] (sections 2 to 5). Sections 6 and 7 then extend the discussion on the applicability of this proposal for real-time applications, detailing the parallelism controls behavior and the open issues.

1 Introduction

The current proposal to extend Ada with a fine-grained parallelism model [1,2,3,4] is based on the notion of tasklets, which are non-schedulable computation units (similar to Cilk [6] or OpenMP [7] tasks).

The work in [1] 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 [2], 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). Tasklets are defined to be orthogonal to Ada tasks and need to 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.

In [3] the semantic model of the proposal is re-defined. 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 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 (though in [3] this was restricted to non-blocking operations).[2]

In [4], the model was extended with a proposal for the underlying tasklet execution behavior, 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 model is

---

[1] Available at: http://www.cister.isep.ipp.pt/private/HILT2014.pdf and at: http://www.cister.isep.ipp.pt/private/AE2015final.pdf

[2] 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 [5, section 9.7.4].

then used to demonstrate how synchronization between tasklets can also be made using potentially blocking operations and how it can be used for real-time systems.

This paper summarizes the current status of this proposal, further detailing how the programmer can perform explicit control of parallelism, and discussing several open issues.

2 Explicit and implicit parallelization

As specified in [3], there is no syntax for the explicit parallelization of individual subprogram calls, since such parallelization can be performed implicitly by the compiler, when it knows that the calls are free of side-effects. This is facilitated by annotations identifying global variable usage on subprogram specifications [3].

Explicit indications of potential parallel code is given by two constructs:

- parallel blocks

```
declare
   X, Y : Integer;
   Z : Float;
begin
   parallel
      X := Foo(100);
   and
      Z := Sqrt(3.14) / 2.0;
      Y := Bar(Z);
   end parallel;

   Put_Line(“X + Y=” &
            Integer'Image(X + Y));
end;
```

- parallel loops

```
declare
   Partial_Sum : array (parallel <>)
                 of Float
               := (others => 0.0);
   Sum : Float := 0.0;
begin
   for I in parallel Arr'Range loop
     Partial_Sum(<>) := Partial_Sum(<>) +
                        Arr(I);
   end loop;

   for J in Partial_Sum'Range loop
     Sum := Sum + Partial_Sum(J);
   end loop;
   Put_Line ("Sum over Arr = " &
             Float'Image (Sum));
end;
```

Attributes are also proposed for implicit reduction operations when parallel processing of arrays:

```
Put_Line ("Sum over Arr = " &
    Float'Image (Partial_Sum'Reduced(
                 Reducer => "+",
                 Identity => 0.0)));
```

Then, in [3], two aspects are proposed to give the compiler sufficient information on subprograms usage of variables and synchronization, allowing for implicit parallelization:

- A `Global` aspect to identify which global variables and access-value dereferences a subprogram might read or update, where by default, the global aspect is `(In_Out => all)` for normal subprograms (the subprogram may access all global variables); and
- A `Potentially_Blocking` aspect that can be applied to subprogram specifications to indicate whether they use constructs that are potentially blocking (or call other subprograms that have the `Potentially_Blocking` aspect with a value of `True`) where the default value for the `Potentially_Blocking` aspect is `True`.

Given the information in the `Global` and `Potentially_Blocking` aspects (as well as appropriate use of `Overlaps_Storage`), the compiler now has enough information to determine whether two constructs can be safely executed in parallel. When the programmer explicitly specifies that two constructs should be executed in parallel, the compiler can use this knowledge to give appropriate warnings wherever data races are possible. However, it can be a burden on the programmer to add explicitly parallel constructs everywhere in a large program where parallel execution is safe. Therefore, this proposal is designed to enable safe *implicit* parallelization of suitably annotated Ada programs.

3 The Tasklet DAG Model

In [2], a model is defined 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).

Figure 1 shows code representing the body of execution of an Ada 202X task (according to the parallelism syntax proposal in [3]), 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)

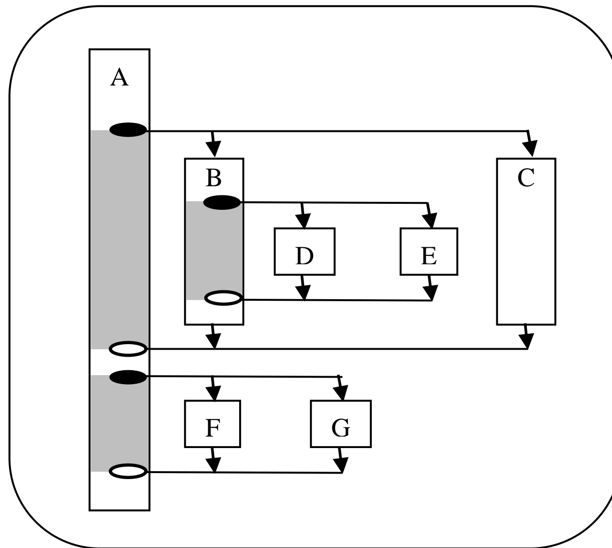

Figure 2. Task DAG example (rectangles denote tasklets, dark circles fork points, and white circles join points)

This model of tasklet execution is a *fully strict* fork-join [8], where new tasklets spawned by the execution of a tasklet are required to complete execution before the spawning tasklet is allowed to complete.

4 The Tasklet execution model

The DAGs execution is based on a pool of abstract executors (Figure 3), which are required to serve the execution of tasklets subject to guaranteeing task *progress*, under certain assumptions.

An *executor* is an entity that is able to carry the execution of code blocks. The definition allows for an implementation that maps executors to operating system threads, but also allows other implementations to be provided. The justification for this separation of executors and threads 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.

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.

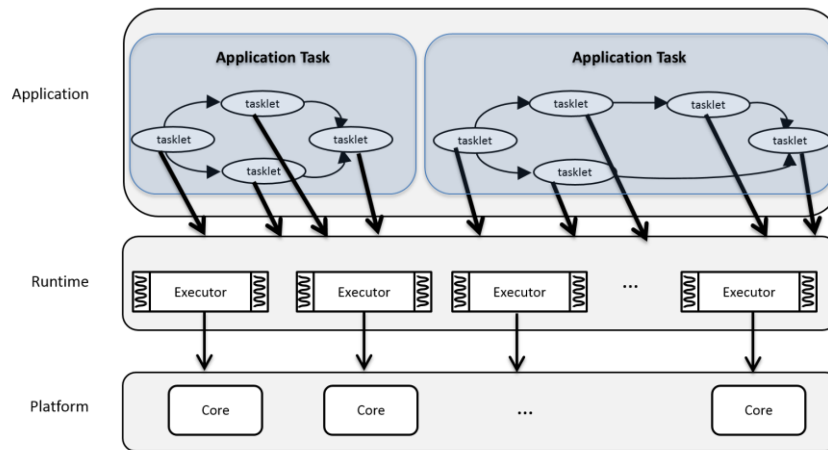

Figure 3. Vertical stack of application, runtime and platform

The default 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, under certain conditions (see [4]) tasklets are allowed to be executed by more than one executor. Note that at any time the executor itself might be preempted by another executor executing a higher-priority tasklet. Therefore, run-to-completion does not mean non-preemptible.

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 executing on an executor is the task then 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. It is nevertheless allowed for the implementation to have a limited number of executors, thus not guaranteeing immediate progress, but it must guarantee that eventually a task that is not blocked will progress.

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), and that cannot be guaranteed to be available in a bounded time.

Considering this, different forms of progress are proposed in [4]:

- *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.

Limited progress is defined for the cases where it is required that a bounded number of executors is pre-determined, and the implementation is such that executors block when tasklets block.

The implementation may allocate multiple tasklets to the same executor, allowing for these tasklets to be executed sequentially by the executor (under a run-to-completion model). It is nevertheless possible that tasklets that have not yet started to be executed and are queued for one executor, can be re-allocated (e.g. with work-stealing [8]) to a different executor.

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. Note that this 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.

In the general case it is implementation defined whether or not a tasklet, when it blocks, releases the executor. For eventual progress, 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). The implementation might release the executor but maintain the state of the blocked tasklet in the executor, for later resumption (by using, e.g., a cactus stack).

This means that when a tasklet being executed by an executor performs a blocking operation, either

a. the executor saves the tasklet state and proceeds with executing another tasklet; or
b. a new executor is spawned if there is no free executor to continue executing tasklets (for that task or for other tasks in the application).

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.

In order to guarantee progress, 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 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 [6]. 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.

In [4] we identify the conditions in which tasklets may synchronize with protected operations (tasklets are the "caller" as specified in the standard [5, section 9.5]):

1. The implementation guarantees that all tasklets will eventually be allowed to execute, and
2. The implementation ensures that each call to a potentially blocking operation is allocated to a single tasklet.

Condition 1 is satisfied in implementations that provide immediate or eventual progress, and condition 2 is satisfied by the compiler that generates an individual tasklet whenever a call is performed to a potentially blocking operation [5, 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 [3] this is the default value of the aspect).

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:

1. 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
2. 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
3. executors that execute the children tasklets inherit the lock from the executor executing the parent, and do not try to 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.

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

5 Real-time model

[4] proposes 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 (varying because of control-flow), running on a shared memory multiprocessor (multi/manycore). The use of enhanced parallel programming models such as the one proposed here for Ada, will allow for the compiler (with optional parameters and annotations provided by the programmer) to automatically generate the task graphs. These graphs can then be made available for analysis. An alternative approach is to give the programmer with more control of the tasklet generation (this is detailed in section 6).

Tasklets run at the priority (and/or with the deadline) of the associated task [3]. 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 [4] and quantum (budget issues are nevertheless discussed in open issues). 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 [5, 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 executor to be scheduled on any processor within the task's dispatching domain, while fully *partitioned* scheduling means that the on-line scheduling algorithm relies on executors being pre-assigned to individual processors by some off-line analysis. We also can consider intermediary strategies where some executor 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 is that there is no priority inversion within the domain, namely that 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.

This model allows using current real-time systems methods for parallel tasks to guarantee the schedulability of the application as the tasklet DAG can be converted to a DAG of sub-tasks or a synchronous fork-join structure [4]. It is important to note that the timing analysis of parallel execution is still an open challenge in real-time systems [4]. The current work addresses this challenge by allowing 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 (or even co-scheduling of computation and communication), reducing the contention at the hardware level.

To accommodate models where blocking (or voluntary-suspension) is not allowed inside a job (one iteration of the recurrent loop in a real-time task), the following additional rules are applied:

- 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);
- 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.

---

[3] While there are approaches that requires setting different deadlines for individual nodes in the graph (decomposition techniques), in our proposed model base priorities and deadlines of tasklets remain the same as the parent task, for the following reasons:

- To simplify the creation and scheduling of tasklets, all tasklets share all attributes of the parent task, including ID and priority;
- Priority and deadline represent the relative urgency of the job executing. Urgency between tasklets of the same DAG is not meaningful since it is only the correct and timely completion of the complete DAG that matters.

Decomposition techniques can be supported by program restructuring into different Ada tasks.

If priority/deadline boosting is required, e.g. within a protected action, it is only the executor that is actually running inside, e.g., the protected action that will have this change. All other executors of the same task will continue at its base priority/deadline.

[4] Execution time timers measure the amount of time that a single task uses, that a group of tasks use, or that an interrupt routine uses and notifies a handler if that time is exceeded. Under our proposals, the execution of a tasklet is reflected in the budget of its task.

6 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. [4] provides a summary of potential controls to be added, which are explained here in detail.

6.1 Executor Count

The proposed model where each task has its own unique pool of executors, provides useful properties such as improving the temporal and spatial isolation from other tasks.

An important parameter of this approach that for analysis purposes is the need to understand the bounds on the number of executors associated with a given task. For a constrained real-time system, it may be desirable to specify this bound, and so we propose that there be a new task aspect called `Executors`, be considered for addition to the Ada standard. E.g.:

```
task My_Task with Executors => 4;
```

The value of the aspect could either be specified as a static integer value indicating a specific number of executors, or it could be specified to have the special value, Unbounded, meaning that the implementation starts off with one executor per core initially, but each tasklet of the task that makes an unbounded blocking call involves the dynamic creation of a new executor, if all executors associated with the task are already busy. If a specific number of executors is to be specified, static analysis of the code may be needed to determine the appropriate value for this setting that is high enough guarantee that deadlock cannot happen.

If all tasklets associated with a task do not involve potentially unbounded blocking, then a reasonable default for the executor count bound could be the number of available cores, since all executors will be kept busy processing tasklets. However, if some of the tasklets involve unbounded blocking calls, then the tasklets associated with each such blocking call within a POP should have a dedicated executor, in order to avoid deadlock, and individual tasklets should be created to manage each such blocking call. It is expected that there are real-time environments where such dynamic behaviour would be undesirable, and so it would be necessary that there be a way to specify that a task is not allowed to make potentially blocking calls that are unbounded in duration.

6.2 Maximum Executors for a Dispatching Domain

In addition to specifying the number of executors allowed for a task, it may also be desirable to specify the total number of executors allowed for the dispatching domain.

We propose that another version of Create be added to the package `System.Multiprocessors.Dispatching_Domains`, as follows;

```
function Create
   (First, Last : CPU;
    Max_Executors : Natural) return Dispatching_Domain;
```

This version of `Create` specifies the maximum number of executors that may be allocated to the dispatching domain. The total number of executors for all tasks specified by the `Executors` aspect may not exceed the value of `Max_Executors` for the dispatching domain.

6.3 Potentially_Unbounded_Blocking

In [3], we proposed a `Potentially_Blocking` aspect could be applied to subprogram and package specifications that indicate which calls can potentially block. Here we extend that capability by also defining an aspect, `Potentially_Unbounded_Blocking`. Whereas Potentially_Blocking identifies subprograms that involve potentially

blocking constructs as defined in the standard, `Potentially_Unbounded_Blocking` only identifies a subset of the potentially blocking constructs, and in particular does not include delay statements, since delay statements are a form of bounded blocking. Delay statements cannot cause deadlock, because they have guaranteed forward progress consistent with the properties of time. It is conceivable that a real-time system may want to allow calls that have `Potentially_Blocking` specified as True, but disallow calls for which `Potentially_Unbounded_Blocking` is True. Any tasklets generated by such a call would require executors to be generated dynamically to handle the call. Real-time analysis of the task could in theory determine the number of dynamic executors needed by each task in the system.

Similarly, stricter real-time systems may want to disallow all calls for which `Potentially_Blocking` is `True`, which would imply that the number of executors associated with the task corresponds to the number of cores in the dispatching domain of that task. This eliminates the dynamic behavior and simplifies analysis.

In addition to being able to specify the `Potentially_Unbounded_Blocking` aspect on subprograms, we propose that the aspect can be specified as a configuration pragma, and also allowed to specify with task declarations. Such a specification for a task with the value False implies that the task guarantees forward progress, which includes any of the tasklets spawned by the task. The `Potentially_Blocking` aspect however is not applicable to tasks since all real-time tasks typically involve at least a single delay.

6.4 Executor Migration

Another capability that the programmer may want to control is whether executors are allowed to migrate to other cores. If executors are pinned to a core then it might allow for a simpler analysis, depending on what type of real-time analysis is needed. It may also allow for a simpler run-time, since executor migration may impose a heavier overhead on the runtime, as the runtime would need to track which cores are the most suitable to execute the next available tasklet, and the migration activity itself is likely to involve some overhead. On the other hand, executor migration potentially offers better core utilization and minimizes priority inversions, since the runtime if striving to maintain the highest available tasklets across all available cores as much as possible. It is not clear whether executor migration offers better processor utilization. It would depend on whether any additional overhead does not fully offset the gains in performance.

Since it is not clear whether executor migration would always be desired or not, we propose a restriction (`No_Executor_Migration`) that allows the programmer to specify this.

6.5 Tasklet_Count

Another need is be able to specify the number of tasklets that are created for a given parallel loop. This may be needed for real-time analysis, to know the number of parallel tasklets in a given timeframe. Also, this can be important for overhead control. It seems clear that the greater the number of tasklets generated, the greater the overhead for the loop. On the other hand, the higher number of tasklets, the better potential for improved core utilization, since the work is being divided into a finer grain size, which theoretically would allow for the work to be more evenly spread across the processors.

To control the number of tasklets generated for a parallel loop, we propose that a new aspect be allowed that may be specified on the declaration of a loop iterator type.

In particular, the `Tasklet_Count` aspect is associated with an integer value, which indicates the maximum number of tasklets generated when that iterator subtype is used as an iterator in a for loop involving loop parameter specifications.

The form for these loops in Ada is;

```
for defining_identifier in [reverse] discrete_subtype_definition
```

For this form of for loop, the expectation would be that the programmer would declare a specific subtype that specifies the range of iterations, and indicate the corresponding `Tasklet_Count` aspect associated with that range.

For example:

```
subtype Loop_Iterator is Natural range 1 .. 1000 with
   Tasklet_Count => 10;
```

```
for I in parallel Loop_Iterator'Range loop
    Array (I) := Array (I) + I;
end loop;
```

6.6 No_Implicit_Parallelism Restriction

For real-time analysis, it is important to understand where parallelism is being used, and potentially such usage should be explicitly indicated by the programmer. For the general model, it is desirable that the compiler also be able to implicitly generate parallelism when it can determine that such parallelism will benefit performance. For real-time environments however, it may be desirable to restrict parallelism to those POP's explicitly annotated by the programmer. We propose that the restriction, `No_Implicit_Parallelism` be added to the set of restrictions already defined by the standard.

6.7 No_Nested_Parallelism Restriction

Allowing nested parallelism can introduce complications for analysis. Generally, the tasklets higher up in the ancestry chain represent lower overhead, and bigger chunks of work, while nested tasklets typically offer less benefits from parallelism due to nested overhead, and smaller chunks of work. It may even be that the overhead for the nested tasklets exceeds the performance benefits.

```
--  Large Benefit, low overhead
for I in parallel 1 .. 1000 loop

   -- Medium Benefit, Med overhead
   for J in parallel 1 .. 100 loop

       -- Small Benefit, high overlead
       for K in parallel 1 .. 10 loop
          Process_Array(Arr, I, J, K);
       end loop;
   end loop;
end loop;
```

Similar effects can occur for nested parallelism in the form of parallel blocks, and recursive parallelism.

From an analysis perspective, it can make analysis difficult, because it can be complicated to determine which sized tasklets will end up being executed, particularly in a work stealing scenario where tasklets may get arbitrarily stolen from one executor to another. For these reasons, it is expected that there will be cases where it will be desirable that nested parallelism will not be desired. To answer this need, we propose that the restriction, `No_Nested_Parallelism`, be added to the list of Restrictions that the programmer may specify in the standard.

7 Open Issues

Several issues are still open. This section provides a list of the ones which are under discussion (extended from [4]).

7.1 Mixed Priorities and Per-Task Deadlines

There are approaches that require setting different priorities/deadlines for parallel computation, 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.

7.2 Changing task and protected object priorities (and other attributes)

As per the model, when a tasklet executes Set_Priority, it is the base priority of the task that is changed, affecting all tasklets of that task. But it is open how to deal with multiple parallel calls from different tasklets. Care should be taken to ensure that calls to change a priority or deadline are executed by only a single tasklet (or the change deferred until outside of any POP), and ideally when it is the only active tasklet. Although a serial equivalence may exist, it is a potential error to let multiple tasklets change the priority or deadline, especially if such changes reflect different values. The same applies to changing other task attributes and to protected objects priorities.

7.3 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.

7.4 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.

7.5 Parallelizing inside interrupt/timing event handlers

Since interrupts and timing events are expressed as protected operations in Ada, and the proposed model allows for tasklets to be spawned inside protected actions, this means that parallelizing the code handling an interrupt is allowed. Although it is not clear why this would be useful (as interrupt handlers are in principle short actions), it is also not clear if it should be forbidden.

One issue which is not yet determined is the relation of these with executors. Ada does not determine which run-time stack is used for the execution of interrupt handlers. It could be interrupt handler executor pool, or it could execute in the stack of the currently executing executor. In the latter case, parallelizing would imply using more executors from the same task.

7.6 Relation with Set_CPU /Get_CPU

The Set_CPU call (or the aspect CPU) is used to constrain a task to a single CPU within the execution domain to which the CPU belongs. If the programmer specifies a single CPU for the task it might mean that all executors of the task are pinned to this CPU, thus all tasklets would be executed in the same CPU. This could eventually be considered for the case where there is a single task in the CPU, and parallelization can be used for "software-based hyperthreading". Or it be that any use of parallel syntax by a task that has the CPU aspect specified as something other than Not_A_Specific_CPU, should result in an implementation generating a compiler warning.

Alternatives would be for Set_CPU to pin only the first tasklet of the task or to add a new version of Set_CPU that assigns a set of CPUs to a task, instead of a single one. This would allow the restriction of the task to execute in a subset of the CPUs in the domain.

Get_CPU can return the CPU where the calling tasklet is being executed.

7.7 Tasklet stealing

As specified in [4] (and in section 5 of this paper), 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 (also means that the main/first tasklet cannot be stolen because it has already been started). So, if No_Executor_Migration is specified, any tasklet after starting in a specific core will not leave that core. This is potentially too restrictive.

7.8 Distinguishing between number of allowed and active executors

Section 6 introduces mechanisms to limit the number of executors per task or on a domain. This limit could be either on the number of available executors, or the number of simultaneously allowed active executors (this differentiation makes sense when executors block on tasklet blocking). Supporting both would also be an option.

7.9 Explicitly control executors

If the programmer is able to explicitly specify the number of executors which are processing a specific POP, it is then possible for the programmer to 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). It is not clear if this should be allowed, and, if so, if a model based on language constructs or on a library should be used.Tasklet minimum execution time

One of the important specifiable parameters affecting the separation into tasklets is the minimum execution time permitted for any given tasklet created by the compiler. This minimum time should be at least as great as the overhead of initiating and waiting for a tasklet (plus increased hardware contention due to parallel execution), to ensure that the critical path execution time for the overall task does not increase as a result of breaking a task into multiple tasklets. This could potentially be a compiler switch or a configuration pragma.

7.10 Ada tasks as executors

The current model abstracts from the actual implementation of the underlying executors, but it is necessary to consider if an executor could eventually be an Ada Task, although differently from what was proposed in an earlier work [9], since the current model separates between the design model of concurrency around Ada Tasks, and the platform model of parallelism around executors.

An implementation may eventually use Ada tasks to execute the tasklets, if when application code denotes any task attribute it refers to the logical task, and it is not directed to the executing task. For instance, a call to task id returns the id of the logical task common to all tasklets of the same DAG, and a priority change reflects in all tasks that can execute tasklets of the same DAG.

7.11 Relation with simpler runtimes

In principle it would be possible to implement a system with the current proposal, also adhering to a set of restrictions used for simpler runtimes such as Ravenscar (or some variant of it). Should we specify a parallel simpler model that can be used with these simpler runtimes? This also has relation with any discussion the workshop may that may have for other profiles than Ravenscar.

7.12 Other 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.

7.13 Applicability to high-reliability hard real-time systems

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 analysis requires determinism (and knowledge) of the specific contention mechanisms at the hardware level, something which is more and more difficult to obtain.

This 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.

8 Summary

This paper has presented the status of a proposal to support fine-grained parallelism in Ada. This proposal is a revision of what was presented at the last Real-Time Ada Workshop, with new language mechanisms and execution model. This paper summarizes the current status of the approach, detailing in particular how the programmer can control the underlying parallel behavior, and discussing some open issues.

Acknowledgements

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] 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.

[2] 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.

[3] T. Taft, B. Moore, L. M. Pinho, S. Michell, "Safe Parallel Programming in Ada with Language Extensions", High-Integrity Language Technologies conference (HILT 2014), October 2014. Available at: http://www.cister.isep.ipp.pt/private/HILT2014.pdf

[4] L. M. Pinho, B. Moore, S. Michell, T.Taft, "An Execution Model for Fine-Grained Parallelism in Ada", accepted at Ada-Europe 2015. Available at: http://www.cister.isep.ipp.pt/private/AE2015final.pdf

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

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

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

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

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