An Execution Model for Fine-Grained Parallelism in Ada

Luís Miguel Pinho, Brad Moore, Stephen Michell, S. Tucker Taft

Ada-Europe 2015, Madrid, Spain

Outline

- Review of the tasklet model
  - History and status
- Tasklet execution model(s)
  - Blocking issues
  - To progress or not to progress
- Real-time
  - The model
  - Finer control of parallelization
- Open Issues
  - After a few sessions at IRTAW

Review of the model

- Based on the notion of a logical unit of **potential** parallelism
  - A lightweight task, denoted *Tasklet*
    - When there is no parallelism, there is an implicit tasklet for the Ada Task
  - Tasklet creation is either explicit
    - The programmer specifies algorithms with explicit parallel constructs
  - Or implicit
    - The compiler itself generates the *tasklets* (e.g. parallelizing subprogram calls)

Review of the model

- Separate the design of parallelism from the implementation of parallel execution
  - Allow parallelism design during the development process without the need for profiling
  - Compiler and runtime (with assisted profiling) knows best how to map to the underlying HW
    - Implicit parallelism
    - More complex algorithms may require re-write
      - Programmer writes code under a potentially parallel assumption
        - » Actual execution can be sequential
    - However, also consider the need of a model, where the programmer specifies the details of the mapping, for analyzability

Review of the model

- Restrictions on what the logical unit can be
  - Several parallel frameworks use similar concepts
    - Some provide a complete asynchronous approach
    - Others are loosely defined, with a weak semantic or execution model
  - Ada must clearly have a well-defined model
    - Tasklets are within Tasks
    - With a strict fork-join model
    - Tasklets inherit task attributes
      - Priority, deadline
    - Execution needs also to have a model

Review of the model

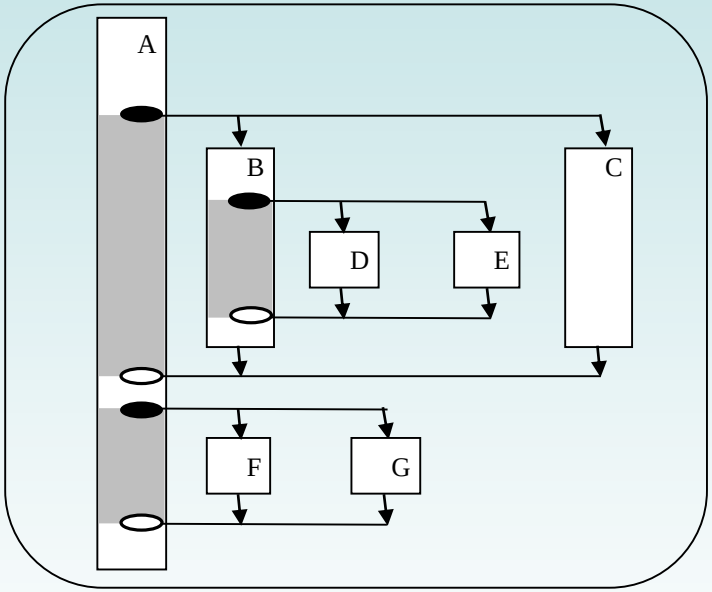
```
task body My_Task is
begin
  -- tasklet A, parent of B, C, F and G,
     ancestor of D and E

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

  -- tasklet A again

  for I in parallel 1..2 loop
    -- compiler creates tasklets F and G,
       child of A, no relation to B,C,D and E
  end loop;

  -- tasklet A again
end My_Task;
```



Review of the model

- Transfer of control out of one parallel sequence (eg. return, exit, goto, raise)
  - initiates the aborting of the parallel sequences not yet completed
    - Once all other parallel sequences complete normally or abort, the transfer of control takes place
  - If multiple parallel sequences attempt a transfer of control before completing
    - one is chosen arbitrarily and the others are aborted
    - Exceptions follow the same model
  - Aborting a tasklet need not be preemptive, but
    - Prevents initiation of tasklets which have not started and of further nested parallel blocks/loops

Review of the model

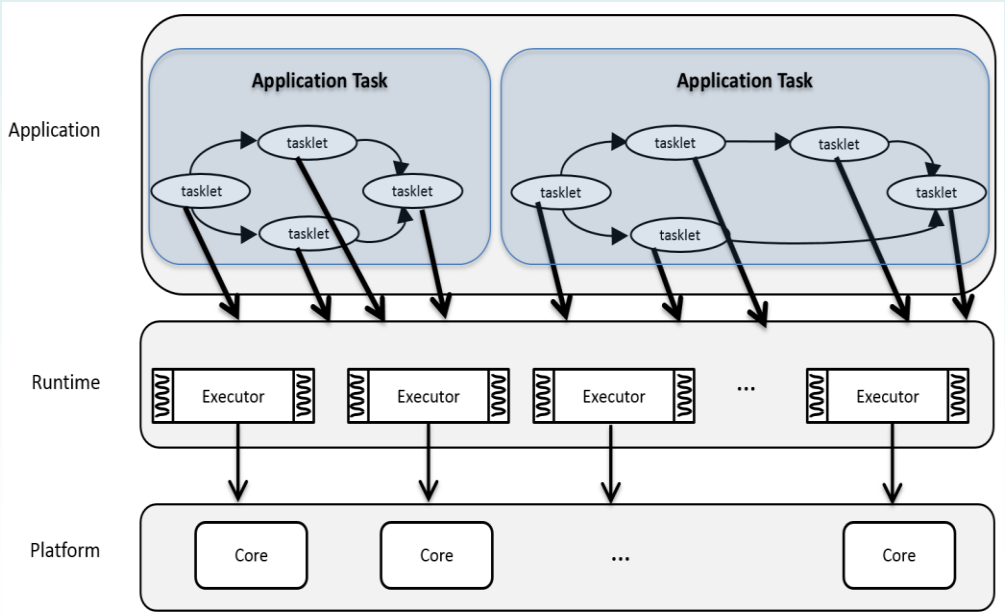
- Shared variables
  - Two actions occurring within two different parallel sequences of the same parallel block/loop are *not* automatically sequential
    - Execution can be erroneous if these actions access the same object (or a neighboring object that is not independently addressable from the first object)
  - New aspects may be specified to enable the static detection of such problems at compile time
    - (proposed) Global aspect to identify usage of global variables and (proposed) Potentially_Blocking aspect to identify subprograms that use constructs that are potentially blocking
    - Compiler has information to determine whether two constructs can be safely executed in parallel

Execution model

- Tasklets compete for the (finite) execution resources
  - Necessary to provide the semantics of the execution model
  - Particularly important if blocking is to be allowed
- Implementations and HW differ significantly
  - Goal is to provide a model which is independent of a particular implementation
  - The focus is then in specifying the perceived behavior that an implementation (compiler and runtime) needs to provide so that Task progress is guaranteed
    - Without constraining how such implementation should be done

Execution model

- An *executor* is an entity which is able to carry the execution of tasklets
  - Most likely executors would be operating system threads, but other approaches may be used
    - Freedom to the implementers, allowing minimum functionality, without full overhead associated with thread management



Execution model

- Tasklet execution is a limited form of run-to-completion
  - When a tasklet starts to be executed by one executor, it is executed by this same executor until the tasklet finishes
    - Simplicity, data locality
    - Exception is only when tasklets block (more later)
  - Run-to-completion does not mean that the tasklet will execute uninterruptedly or that it will not dynamically change core
    - The executor itself might be scheduled in a preemptive global approach

Execution model

- The progress of a Task is defined based on the execution of the tasklets
  - A Task is progressing if at least one of its tasklet is being executed, or ready to be executed
    - Tasklets are considered to be blocked when they are waiting for a resource, which is not an executor nor a core (e.g. waiting in an entry call)
    - Higher priority / earlier deadline tasklets being executed do not impact the model
- Important to understand the conditions which may lead to a otherwise “correct” program to behave in a wrong way

Execution model

- Potentially blocking operations
  - Some parallel algorithms require co-ordination between the potentially parallel tasklets
    - Wavefront, pipelining, etc.
    - Tasklets down the chain require data output by upstream tasklets
  - Allowing tasklets to block may lead to wrong behavior
    - Programmers write the parallel algorithms assuming infinite parallelism
    - Compiler may aggregate parallel computations in “chunks”
    - Runtime may map tasklets to the same or different executors
      - Risks of deadlock, depending on the tasklet allocation to the underlying executor

Execution model

- Potentially blocking operations

```
parallel
   -- some operations
   barrier.wait(3);
   -- some operations
and
   -- some operations
   barrier.wait(3);
   -- some operations
and
   -- some operations
   barrier.wait(3);
   -- some operations
end parallel;
```

If the tasklets are mapped to the same executor, then it deadlocks

```
for I in parallel Range loop
    if … then
       obj.wait_for_my_data;
    end if;
    -- work on my data
end loop ;
```

If the compiler “chunks” in X tasklets < Range then it deadlocks

Execution model

- Implementations must provide one class of behavior
  - *Immediate* – immediate availability of an executor: if a tasklet is ready and a core is available to execute it (either idle or executing lower priority tasklets) then an executor is immediately provided
  - *Eventual* – availability of an executor: if a tasklet is ready and a core is available to execute it (either idle or executing lower priority tasklets) it is guaranteed that an executor will eventually be provided
  - *Limited* – even if cores are available, ready tasklets might need to wait for an executor, and the runtime does not guarantee that one will be eventually available

Execution model

- Blocking operations may be allowed if:
  - Implementation guarantees that all tasklets will be allowed to execute
    - Immediate and eventual behaviors fulfil the requirement
    - Limited model requires offline analysis to guarantee that the number of available executors is always sufficient for the maximum simultaneous demand
  - Implementation ensures that the behavior is as if each call to a potentially blocking operation was allocated to a single tasklet
    - The compiler generates individual tasklets when potentially blocking operations are used

Execution model

- The model does not stipulate how an implementation provides these guarantees
  - Immediately providing an executor may requiring create new executors unboundedly
  - Blocked tasklets may require blocking the executor, or saving and restoring tasklet state
  - Implementations may use bounded or unbounded executor pools
  - Parent waiting for children executing may suspend or spin

Real-time

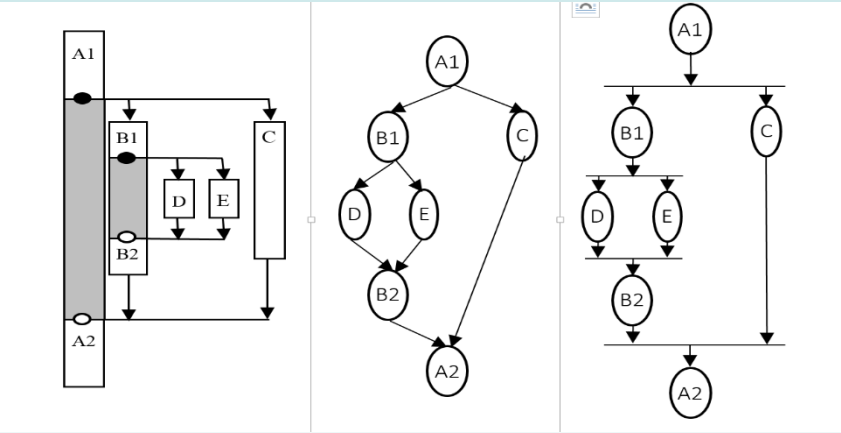
- As usual, real-time tasks map one-to-one with Ada tasks
  - The execution of the Ada task generates a recurrent graph of tasklets

Real-time model

- Each Ada task has its specific executor pool
  - Executors carry the same priority/deadline of the task and (potentially) share budget quantum
    - Offline analysis is used to determine the size of the executor pool
  - Each task (and its DAG of tasklets) execute within the same dispatching domain
  - If priority/deadline boosting is required, e.g. within a protected action, affects only the executor that is actually executing in the action
    - All other executors of the same task will continue at its base priority/deadline

Real-time model

- The use of this model allows the application of state-of-the-art schedulability analysis



  - Applicability to high-reliability hard real-time systems is still far away
    - Timing analysis no longer independent of scheduling
    - HW interference analysis is a major challenge

Real-time model

- Parallelism control
  - In some situations (e.g. real-time) it may be necessary to specify more structure and behavior of execution
  - A set of mechanisms available to provide that control:
    - *Executors* aspect of task/task type to size the pool
    - *Max_Executors* parameter for dispatching domain Create operation
    - *Tasklet_Count* aspect of a discrete subtype, an array type, an iterator type, or an iterable container type
    - *Potentially_Unbounded_Blocking* aspect of subprogram
    - *No_Executor_Migration* restriction
    - *No_Implicit_Parallelism* restriction
    - *No_Nested_Parallelism* restriction

Real-time model

- Open issues
  - Many issues have been thoroughly discussed at the last Real-Time Ada Workshop
  - Some have been closed
  - Some are still open
    - Multiple parallel updates to task attributes
    - Arbitrary selection for transfer of control needs to be further analyzed
    - How to efficiently determine if code is executing in parallel
    - Methods to efficiently track and implement execution timers

Summary

- There is a need to support parallel programming
  - Effort being done in all languages, new and existing
- Ada needs to be augmented with parallel programming facilities
  - With a strong semantic model
  - Reducing the burden from the programmer
  - And syntactic sugar to reduce re-writes
- There is an ongoing effort to produce a proposal
  - Still issues to be addressed
- No full implementation
  - But partial examples from other domains or models

Thank you!

- Questions?
- Do you want to help?