Parallelism in Ada: status and prospects

Luís Miguel Pinho, Brad Moore, Stephen Michell

Ada-Europe 2014, Paris, France

Outline

- Motivation
  - (Almost) nothing new
- Review of the tasklet model
  - Tight(er) semantics
- Proposals in this paper
  - Annotating data types
    - Impact in expressions
  - Parallel blocks
- Open Issues
  - Some still open

Motivation

- Paradigm shift
  - The shift from relying upon increasing processor speed to relying upon increasing parallelism impacts heavily in software development
  - Amdahl's law is clear: the only way to improve Speedup is improving *p*, the percentage of the program which can be parallelized (very low)
  - Virtualization helps but processors tend to be idle
  - It is not just a question of mapping tasks/threads to cores
    - There are more cores than parallel activities in the system
    - And cores can be themselves highly parallel (vectorization)

Motivation

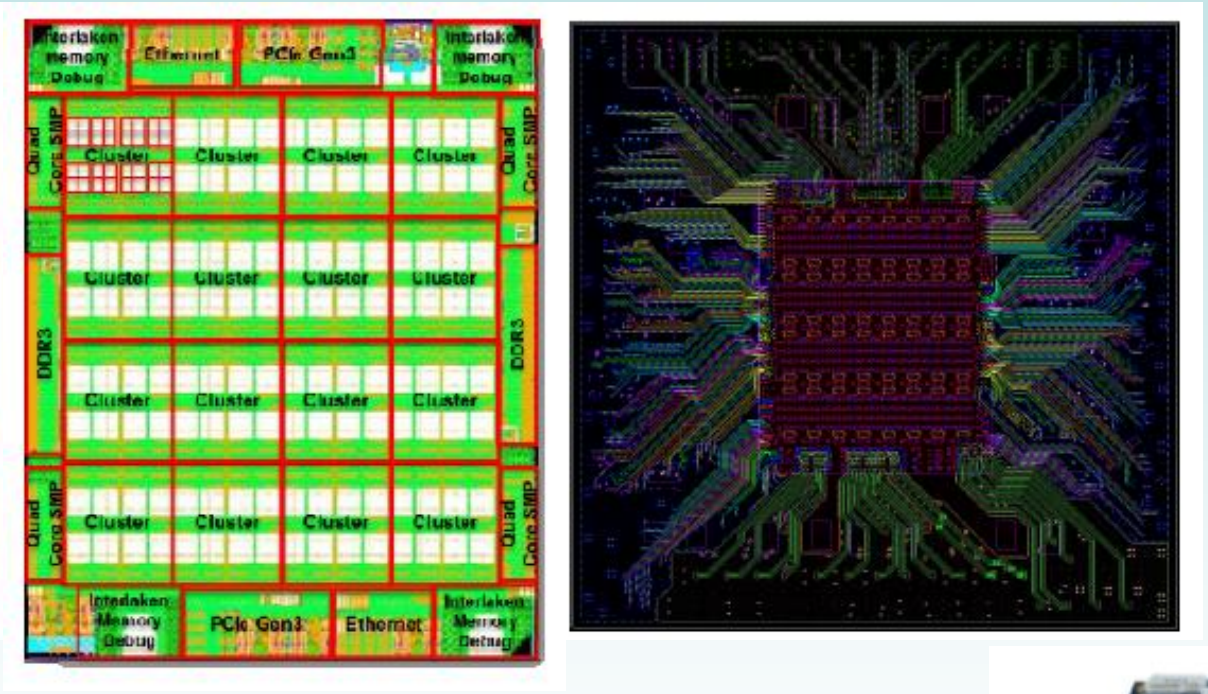
- Software is heavily impacted
  - Needs to adapt and be parallel
    - If not, there is no gain from multi- and many-core
    - But more complex and error prone
    - Compiler-based parallelization is not enough
  - There is no consensus as to programming models
    - Sequential model with automatic parallelization
    - Programming with low level threads interface
    - Task-centric programming
    - Data-flow models

Motivation

- Our view is that concurrency and parallelism should both be in the language semantics
  - Actual syntax is irrelevant …
    - Needs to fit the language model
  - … but it is needed
    - For the general case to help reduce manual re-writes of algorithms
      - Parallel loops, blocks, …
    - For specific cases
      - **Specification of parallelism behavior**

Motivation

- Architectures



Review of the model

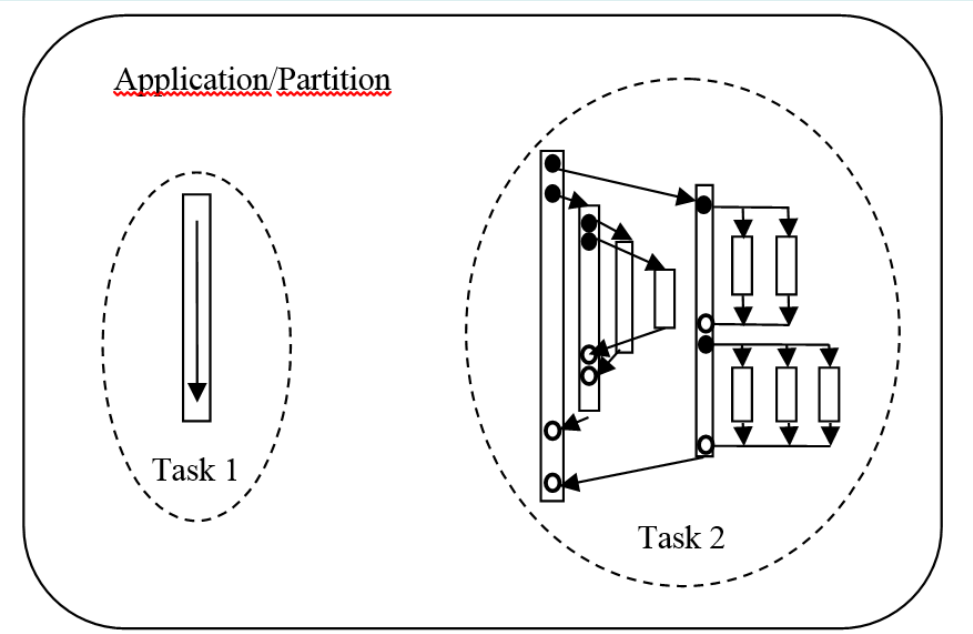
- Based on the notion of a logical unit of **potential** parallelism
  - A lightweight task, denoted *Tasklet*
    - When there is no parallelism, there is an implicit tasklet for the Ada Task
  - Tasklet creation is either explicit
    - The programmer specifies algorithms informing the compiler that *tasklets* should be generated
  - Or implicit
    - The compiler itself generates the *tasklets* (e.g. operating on parallel data types)

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
    - Programmer annotates places in code that are *Parallelism OPportunities* (POP)
      - » Actual execution can be sequential
      - » Compiler may even not generate the code
    - 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
  - Many models allow these logical units to float around in the application
  - Relation between the logical task and the design model or the concurrent model is very loose at most
  - Ada must clearly have a well-defined model
    - Tasklets are within Tasks
    - With a strict fork-join model



Review of the model

- What about syntax
  - The basic approach to the annotation is to use the Ada *aspect* mechanism

    ```
    with parallel => [True| False]
    ```
  - This can be applied to
    - Subprogram specifications
      - Identifies POPs when calling the subprogram
    - For loops
      - Iterations can be in parallel
    - Data types (arrays and records)
      - Operations in the type can be in parallel
    - Blocks
      - Runs in parallel with following code

Review of the model

- We have clarified the semantics a bit
  - Semantics for parallel subprogram calls in standalone statements also applies to parallel blocks and complex expressions
    - Call to a subprogram in a standalone statements and parallel blocks execute in parallel with the following statement(s) in the same scope
    - Calls to parallel subprograms in expressions will execute in parallel with the following subexpression(s)
  - When a subprogram or a block executes in parallel with following statements, the synchronization point for the parallel computations is the earlier of:
    - either the end of the deepest enclosing construct, or
    - the first point where an object updated by the parallel call or block is read or written by the following statements

Review of the model

- Only placing aspects on spec, not on actual call
  - Change only in spec no need to change everywhere it is used
    - Expressions became confusing with aspects in call
  - Parallel => False on specification guarantees that calls to the subprogram are executed by the tasklet that executes the enclosing scope of the call
    - It does not prevent parallelism from being initiated within the body of the subprogram itself
    - It does not prevent parallelization at levels above the immediately enclosing scope of the call.
    - Also, other subprograms in the same enclosing scope may be executing in parallel within that scope, and hence with the subprogram.

Review of the model

- Subprogram call example

```
   -- programmer identifies opportunities:
   -- procedures X and Y can be executed in
   -- parallel with code at place of call
   procedure X with parallel=> true;
   procedure Y with parallel=> true;
begin
   -- …
   X(); -- compiler may create tasklet here
   Y(); -- no need to create tasklet here
end;    -- since enclosing ends here
```

Review of the model

- Recursive subprogram example

```
function Fib( Left : Natural)
   return Natural with Parallel is
begin
   return           -- Expression is the enclosing scope
      Fib(Left-1)   -- Tasklet spawned to execute parallel
                    -- with following sub-expressions
      +             -- consumes results, so sync
                    -- happens before "+"
      Fib(Left-2);-- Nothing left for parallel execution
                    -- in the expression hence no spawning
end Fib;
```

Review of the model

- For loops

```
   S: Integer := 0;
   P: Integer := 1;
begin
   for I in 1 .. 100 -- compiler/runtime may "chunk"
     with Parallel     => True,
          Accumulator => (S, Reduction=> "+",
                             Identity => 0),
          Accumulator => (P, Reduction=> "*",
                             Identity => 1)
   loop
            S := S + I;
            P := P * I;
   end loop;
```

Review of the model

- Specifying behaviour
  - Allow the ability for the programmer to take control of parallel behaviour (e.g. for timing analysis)
  - Aspects added to Parallel => true to refine behaviour

```
Chunk_Size          -- fixes amount of work per tasklet

Worker_Count        -- number of workers

Parallel_Manager    -- programmer own parallel manager

Task_Pool           -- create own pool of workers

Work_Plan           -- strategy for partitioning
                       (e.g. work-sharing, stealing, …)
```

Review of the model

- Example

```
package My_Pool is new My_Pool_Implementation (
                          Number_Of_Workers => 4);
TP: My_Pool.Pool(…);

package Max_Loops is new  Reducing_Loops (
                          Result_Type       => Integer,
                          Reducer           => Integer'Max,
                          Identity          => Integer'First,
package My_Loop is new Max_Loops.Work_Sharing;

for I in Some_Range  with Parallel          => true,
                          Task_Pool         => TP,
                          Accumulator       => Max_Value,
                          Parallel_Manager  => My_Loop.Manager
loop
      Max := Integer'Max (Max_Value, Some_Array (I));
end loop;
```

Parallel Data Types

- A Parallel aspect can be added to data types
  - Inform the compiler that (and how) some of its primitive operations can be parallelized
  - Two new aspects are introduced
    - Parallel_By_Element for arrays
    - Parallel_By_Component for composite types.
  - These aspects specify how the operation on the data type is to be performed
    - based on the composition of its individual elements

Parallel Data Types

- Array example

```
type Par_Arr is array (1..100) of Some_Type
   with Parallel => true;

function "+"(Left, Right: Par_Arr) return Par_Arr
   with Parallel_By_Element =>  "+";
   -- the full specification of the individual
   -- by element +" operation is known to the
   -- compiler so it is only the operation name
   -- that is required
```

Parallel Data Types

- Array example

```
type Par_Arr is array (1..100) of Some_Type
   with Parallel => true;

function "+"(Left, Right: Par_Arr) return Par_Arr
   with Parallel_By_Element =>  "+",
        Chunk_Size          => 10;
   -- programmer my specify the size of each "chunk"
```

Parallel Data Types

- Not all operations may be “By_Component”

```
   function "*"(Left,Right: Par_Arr)
      return Some_Type is
    Result: Some_Type := Id_Value;
begin
    for I in 1 .. 100
        with Parallel => True,
             Accumulator => (Result,
                             Reduction => "+",
                             Identity => Id_Value)
    loop
       Result := Result + Left(I) * Right(I);
    end loop;
    return Result;
end "*";
```

Parallel Data Types

- Record example

```
type Par_Rec is record
   with Parallel => True
   A: Some_Type_A;
   B: Some_Type_B;
end record;

function "+"(Left, Right: Par_Rec) return Par_Rec
   with Parallel_By_Component => (A => "+",
                                  B => Some_Op);
```

Impact in Expressions

- Previous work allowed for programmers to introduce aspects within expressions
  - To control the actual spawning of parallelism
    - However, we now consider that this is a very complex, error prone, and “inelegant” mechanism, which should not be used.
- Instead, we now propose that expressions are parallelized by the compiler
  - Using the knowledge on parallel operations on data types and function calls

Parallel Blocks

- Allowing a block to be annotated (using the **with** Parallel aspect notation) as being possible to execute in parallel
  - Block executes in parallel with the statements immediately following the block end statement.
  - The synchronization point for the parallel block and subsequent statements is the end of the immediately enclosing scope

```
begin
   declare with Parallel => True
      -- …
   begin
      -- this code executes in parallel
   end;
   -- with this code
End;
```

Open Issues

- Implicit synchronization
  - The first model considered the possibility to synchronize implicitly to wait for asynchronous results
    - We are now forbidding such race condition, the compiler rejects code where a potential race condition occurs
    - This applies to both results of parallel call to subprogram and reading variables which are being updated in a parallel block
- Pure subprograms
  - By introducing parallel notations, the cases where the code may be updating the same variable simultaneously increases
    - Compilers can detect many cases of unsafe behavior, but many situations are not detectable.
    - Introducing real pure subprograms in Ada, without side effects, could potentially make for much safer parallelism
  - There [will be] {**is**} an alternative proposal along this line

Open Issues

- Tasklet synchronization
  - Many times we may need to synchronize or communicate between tasklets
  - Using protected objects and barriers may be possible but we still need to analyze this further
    - E.g. if iterations of a loop synchronize, it may deadlock if compiler “chunks” the iterations
    - If actual execution is sequential, runtime must guarantee equivalent semantics as of parallel execution
  - This is still open

Open Issues

- Distribution
  - Some modern many-core architectures can be seen as truly distributed systems
  - The model proposed here can be extended so that tasklets can execute in different partitions
  - However analysis is needed to determine if a different distribution execution model is required
    - Particularly considering communication between partitions
  - This is still open

Open Issues

- Specifying behavior and mapping
  - Aspects are provided for the programmer to specify parallel behavior
    - Still need to further detail how the tasklet model provides analyzability
    - Also need control on/off of implicit parallelism
  - Hardware mapping may also be required
  - And how to address this under the new proposal
  - This is still open

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
  - And syntactic sugar to reduce re-writes
- There is an ongoing effort to produce a proposal
  - This paper presented one of the possibilities
  - More in the future (and in the past ☺)
    - Several open issues (IRTAW-15?)