Tasklettes – a Fine Grained Parallelism for Ada on Multicores

Stephen Michell

Brad Moore

Miguel Pinho

Basic Problem

- Processors today no faster (in clock speed) than those 5 years ago
- Energy consumption and heat dissipation a problem for further CPU clock speed increases.
- Lower energy consumption can be maintained with use of multicore
- Onset of massively parallel computers is here
- Advances must come from restructuring code to take advantage of Parallel OPportunities (POP)

Need for explicit syntax

- For the general case
- To help reduce manual rewrites to make POPs explicit
- For the intersection of tasklettes and
    - Hardware
    - Data placement
    - Work management strategies

What this means

- Must try to use every parallel opportunity
- Loops (obviously)
- Subprograms (|| with each other)
  - Watch the data overlaps
- Whole-structure operations & expressions
  - Including initialization
- Cannot expect compiler to recognize and parallelize everything for us (Inefficient small loops, user defined reductions, etc)
- Sequential algorithms abound
  - Guidance (and rewrites) is needed by the programmer

Parallel Syntax

- New aspect “with parallel”

```
function Fib(I: Integer) return Integer
   with Parallel;

for I in D'Range with Parallel,
loop ...
```

A non-obvious example

- Histogram
  - Basic sequential code and problem

```
Cumulative(1) := Histogram(1);
for I in 2 .. N loop
 Cumulative(I) :=
   Cumulative(I-1) + Histogram(I);
end loop;
```

=> this cannot be automatically parallelized, since calculating for any I requires that the result value for I-1 is already available.

  - Solution after we give syntax

Advanced Syntax

- New aspect “with parallel”

```
function Fib(I: Integer) return Integer
  with Parallel;

for I in D'Range
   with Parallel,
        Chunk_Size => N,
        Reduction => “+”,
        Workers => W,
        -- inferred from type sum
        Accumulator => Sum,
        Identity => 0
loop ...
```

Histogram using syntax

```
-- declarations
Barrier : Synchronous_Barrier(Worker_Count);
Intermediate : array (1 .. Worker_Count) of Integer;

-- code
for I in 1 .. Histogram'Length
      with Parallel => True, Workers => Worker_Count
loop
   -- parallel inclusive scan phase
   Cumulative(I'Chunk_Start) := Histogram(I'Chunk_Start);

   for J in (I'Chunk_Start + 1) .. I'Chunk_Finish loop
     Cumulative (J) := Cumulative (J-1) + Histogram (J);
   end loop;

   Intermediate (I'Worker_Index)
       := Cumulative (I'Chunk_Finish);

   -- first barrier
   Wait_For_Release (Barrier, Notified);
```

Histogram parallel- cont

```
   -- sequential exclusive scan phase, only one task does it
   if Notified then
     Sum := Intermediate (1);
     for J in 2 .. Worker_Count loop
        Sum := Sum + Intermediate (J);
        Intermediate (J) := Sum;
     end loop;
   end if;

   -- second barrier
   Wait_For_Release (Barrier, Notified);

   -- first chunk there is nothing to do
   if I'Worker_Index > 1 then

      -- parallel final phase
      for J in I'Chunk_Start .. I'Chunk_Finish loop
        Cumulative (J) := Cumulative (J) +
            Intermediate (I'Worker_Index - 1);
      end loop;
   end if;
end loop;
```

Histogram parallel – comments

- this only works if implementation is based on tasks and there is a 1-to-1 mapping between strands and tasks
- the 1-to-1 does not need to be *always*. Only in this loop.

- note that if parallelism is not available and the code executes sequential, it will work as the serial one

Issues

- Need for advanced control
- Need something to schedule cores
- Have a unit we call a Tasklette
  - Could also be called a Strand, Fibre
  - One tasklette maps to one parallelization unit

Tasklettes and Tasks

- Somehow we need to interoperate with tasks
  - Tasks will be running on cores
  - Serious issues with blocking, data protection, schedulability
  - Integrating models seems reasonable
- Other solutions (C++'s Cilk, OpenMP) use threads to run strands
- Our proposal can use tasks when you want roll-your-own back end

Roll-your-own

- We propose a plugable back-end to control how tasklettes deliver their service
    - Work sharing, work stealing, work seeking
    - Satisfying program's priority model
- Ada.parallel
- Ada.parallel.(generics)
- Tasklettes run **as if** executed by a tasking back-end (but could be simplified by compiler and runtime if possible)

Issues that we considered (cont)

- Goal - parallel model be compatible with
  - Ada Tasks as a design unit of concurrency
  - Ada Tasks as an implementation unit of execution

Ada Tasks as a design unit of concurrency

- An Ada Task can be parallelized with multiple logical tasks cooperatively performing a previously sequential algorithm
- These multiple logical tasks are components of the Ada Task (as a design notion), therefore must relate to it

Ada Tasks as an implementation unit of execution

- It is not possible to forget that Ada Tasks are the unit of execution
  - They have priorities, affinities; dispatching domains and scheduling policies are done in terms of tasks
- Execution of the logical tasks must be compatible with Ada tasks
  - We could add a new language unit of execution, but this would
    - make the language more complex
    - Introduce potential errors in the specification of semantics
    - Make more difficult implementations of Ada 2020

Our conclusion

- Tasklettes must be tightly integrated with Ada tasks, and usually tasks will be the concurrency unit executing tasklettes

Other Issues

- How to handle potentially blocking operations & PO's?
  - *No issue if tasklette code carried by tasks*
- Priority between tasks & tasklettes, how to manage?
  - *Task execute tasklette code -> same priority as client task*
- Accumulating results
  - *Must synch before results consumed or at block end*
- Real Time Scheduling of tasklettes and between tasks and tasklettes
  - *No consensus on a multicore model for concurrent tasks executing – let alone tasklettes*
    - Parallel case worse – no models and no consensus
    - Starting to emerge

Questions?