Library Oriented Approaches for Parallel Loop Constructs
Outline

- Parallelism Intro
  - Loops and Blocks
  - The Challenge of Loop Reduction
- Paraffin
  - Design
  - Capabilities
  - Examples
- Syntax Helpers?
  - Goal: Integrate with libraries
  - What can be done?
Where Ada stands to shine

- Ada's focus on correctness
- Static checking
  - Let compiler find problems when possible
  - Catch bugs earlier in development.
- Parallel Programming is difficult to get right.
  - Let Ada compiler help programmer out as much as possible.
- Ideally Ada would prevent data races
  - Other languages let programmers shoot themselves in the foot more readily.
Parallelism Constructs

- Basically two constructs needed
  - Parallel Blocks
    - Forking two or more actions in parallel.
  - Parallel Loops
    - Simple Iteration loops
    - Reduction loops
    - Container Iteration
Parallel Blocks

- When Two or more lengthy actions can execute at the same time.
  
  Paint_Sistine_Chapel; -- 1502 - 1512
  Paint_Mona_Lisa; -- 1503 - 1506

- Doesn't work so well with just one worker
  - But with two or more workers, works great!

- Same goes for;
  
  Build_Rome; -- Took longer than a day
  - A classic Divide and Conquer problem
Parallel Blocks
Works well with Recursion

- Leonardo Bonacci (c. 1170 – c. 1250)
  - Known also as Leonardo of Pisa
  - You might know him by his other name;
    - Leonardo Fibonacci
      - popularized the Hindu–Arabic numeral system
      - Wrote Liber Abaci in 1202
        - A historic book on Arithmetic
        - Among many other things, introduced the Fibonacci sequence
Recursive Parallel Fibonacci

\[ F_n = F_{n-1} + F_{n-2} \quad \{0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots\} \]

```plaintext
function Fibonacci (N : Natural) return Natural is
begin
  if N < 2 then
    return N;
  end if;

  return Fibonacci (N - 2) + Fibonacci (N - 1);
end Fibonacci;
```

Opportunity for Divide & Conquer
function Fibonacci (N : Natural) return Natural is
  Left, Right : Natural;
begin
  if N < 2 then
    return N;
  end if;
  parallel
    Left := Fibonacci (N - 2);
  and
    Right := Fibonacci (N - 1);
  end parallel;
  return Left + Right;
end Fibonacci;
Parallel Loops

• Same action occurring multiple times

Italian Music Term: **Da Capo (D.C.)**

Italian goto statement

Nested Loop
Middle Bar plays twice

To Coda

D.C. al Coda

CODA

```
for Verse in 1 .. 2 loop
    Play (Bar1, G1, 4s); Play (Bar2, A2, 4s);
    if Verse = 1 then
        for Repeat in 1 .. 2 loop
            Play (Bar3, B2, 4s);
        end loop;
        Play (Bar4, C2, 4s);
    end if;
end loop;
Play (Bar5, C1, 6s);
```

Go back to the Beginning
Make this a parallel loop
(We might get Jazz!)
Biggest challenge for parallelism syntax

- Loop Reductions (by far)
  - Combining parallel results into a single overall result

```
Sum := 0;
for I in 1 .. N loop
  Sum := Sum + I;
end loop;
```

Global result, need to avoid data races

Need to be able to run this loop in parallel, But how?
Benefits of Syntax

- Can be tailored to “suit” a particular problem
  - Has to “fit” in the language, however
- Compiler can have more intimate knowledge
  - eg. Detect data races
- Can be easier to read and write
- Examples of syntactic solution
  - OpenMP (C, C++, FORTRAN)
  - Cilk (C, C++)
Other side of Syntax

- Adds complexity to language definition
- More work for compiler writers
- Danger of unseen problems, or regrets
  - Once something is in Standard, there for good
- Might think of better idea down the road
  - As new hardware and computing platforms arise

- All roads might lead to Rome...
  - but some get us there faster. (Parallelism goal)
Other extreme – Library Approach

- Libraries can be written today using existing syntax (Examples C#, Java)
- Generally easier to implement a library than syntax
- No additional complexity for language definition
- Syntax tends to be generalized
- Libraries can more easily adapt to specific needs
  - Controls, Parameters, Variants, etc
The syntax spectrum

- No need to stick with one extreme or the other
- Might be a middle ground that combines more general syntax with a library approach...
  - The more places new syntax can be used...
    - Generally means more useful
- Other possibility is to provide both
  - Libraries for those who want less “magic”
  - Syntax for those that want ease of expression
Library approach

● How far can we go?
  - To make libraries easy to use
    ● Specifically parallelism libraries
  - Maybe sprinkle on some syntactic sugar?

  - Eg. Ada Containers + Ada 2012 Iterator Syntax

    for Element of Container loop
    Element := Element + 1;
end loop;
Paraffin – A study in parallelism libraries

- Features
  - Written in Ada
  - Parallel Loops
  - Parallel Blocks
  - Parallel subprograms
  - Task Pools (optional)
  - Ravenscar (optional)
  - Non-commutative reduction (optional)
Paraffin – Features (Cont)

- Support for multilanguage use
  - C, C++, C#, Java, FORTRAN, Python, Rust
- Bindings to OpenMP and Cilk
- Native Paraffin implementations as well
- Stack safe parallel recursion
- 3 native load balancing strategies
  - Work Sharing, Work Seeking, Work Stealing
- Supports for Ada 95, Ada 2005, and Ada 2012
- At least two different compiler vendors
  - Adacore + ICC Irvine Compiler
C# Interfacing to Paraffin

class test_paraffin_lib
{
    [ThreadStatic]
    private static int partial_sum;

    static void Main(string[] args)
    {
        int sum = 0;

        paraffin_pkg.parallel_loop
            (from : 1, to: 400000000,
             reset: () => { partial_sum = 0; },
             process: (start, finish) =>
             {
                 for (int i = start; i <= finish; i++)
                     partial_sum += i;
             },
             reduce: () => { sum += partial_sum; });
generic
  type  Loop_Index  is range <>;
  type  Result_Type  is private;
with  function  Reducer
    (Left, Right : Result_Type)
    return  Result_Type;
Identity_Value : Result_Type;
package  Parallel.Generic_Reducing_Loops  is

  function  Parallel_Loop
    (From, To   : Loop_Index;
     Loop_Body  : not null access
    procedure (From, To : Loop_Index;
               Result   : in out Result_Type))
    return  Result_Type;

end  Parallel.Generic_Reducing_Loops;
package Loops is new
  Parallel.Generic_Reducing_Loops
  (Loop_Index  => Integer,
   Result_Type => Integer,
   Identity    => 0,
   Reducer     => "+"); use Loops;

procedure Loop_Body
  (Start, Finish : Integer;
   Partial_Result : in out Integer) is
begin
  for I in Start .. Finish loop
    Partial_Result := Partial_Result + I;
  end loop;
end Loop_Body;

Sum := Parallel_Loop (From => 1, To => N,
                        Loop_Body => Loop_Body'Access);
Idea #1 Lambda Procedures

Sum := Parallel_Loop
(From => 1,
To => N,
Loop_Body => (Start, Finish, Result)
(for I in Start .. Finish loop
 Result := Result + I;
 end loop));
Idea #2 Loop Body Procedures

for (Start, Finish, Result) of Parallel_Loop (From => 1, To => N) loop
    for I in Start .. Finish loop
        Result := Result + I;
    end loop;
end loop;
Idea #3 Stream Function Loops

- Java takes a unique approach with Java Streams
- Functions are pipelined together
  - A library approach

```java
int sum = IntStream.range(1,N).parallel().sum();
```

Delete “Parallel” to get Sequential loop

Collector function terminates Stream
Idea #3 Stream Function Loops

Sum := 0;

for I of Iter(1,N).Parallel.Add(Sum) loop
  Sum := Sum + I;
end loop;
-- Iterating through a map containers keys.
for Pair of Container.Keys loop
  Put_Line(Key_Type'Image(Pair.Key) & " => " & Elem_Type'Image(Pair.Elem));
end loop;

Total : Integer := 0;

for V of Container.Elements.Sum(Total) loop
  Total := Total + V;
end loop;
Summary

- A blend of libraries + general loop syntax can express a parallel loop quite nicely.
- Desire to represent parallel loops as loops.
- Desire to represent functions as functions.
- Which one wins? Maybe we need both?
- Combining Java Stream idea with idea for loop procedure bodies seems like a good way to express parallelism with minimal syntax.
Questions? Comments?

- Thank you!