AdaCore
The GNAT Pro Company

Safe Parallel Language Extensions for Ada 202X

Tucker Taft, AdaCore

Co-Authors of HILT 2014 paper:
Brad Moore, Luís Miguel Pinho, Stephen Michell

June 2014

HILT 2014 Co-Located with OOPSLA/SPLASH in Portland OR

October 18-21, 2014 — Portland, Oregon (USA)
Pre-conference tutorials: October 18-19
Conference: October 20-21; Co-located with *SPLASH 2014*

Sponsored by ACM SIGAda in cooperation with SIGBED, SIGCSE, SIGPLAN, SIGSOFT, Ada-Europe and the Ada Resource Association

Contact: SIGAda.HILT2014 at acm.org   www.sigada.org/conf/hilt2014

KEYNOTE SPEAKERS

Tom Ball

Christine Anderson

celebrating the 20th Anniversary of completion of Ada9X

Submission	*Deadline*
Technical articles, extended abstracts, experience reports, panel session proposals, or workshop proposals	~~**June 7, 2014**~~ **now July 5!**
Industrial presentation proposals	**July 5, 2014** (overview) **Aug 6, 2014** (extended abstract)
Send **Tutorial** proposals to	~~**June 7, 2014**~~ **now July 5!**

www.sigada.org/conf/hilt2014

The *Right Turn* in Single-Processor Performance

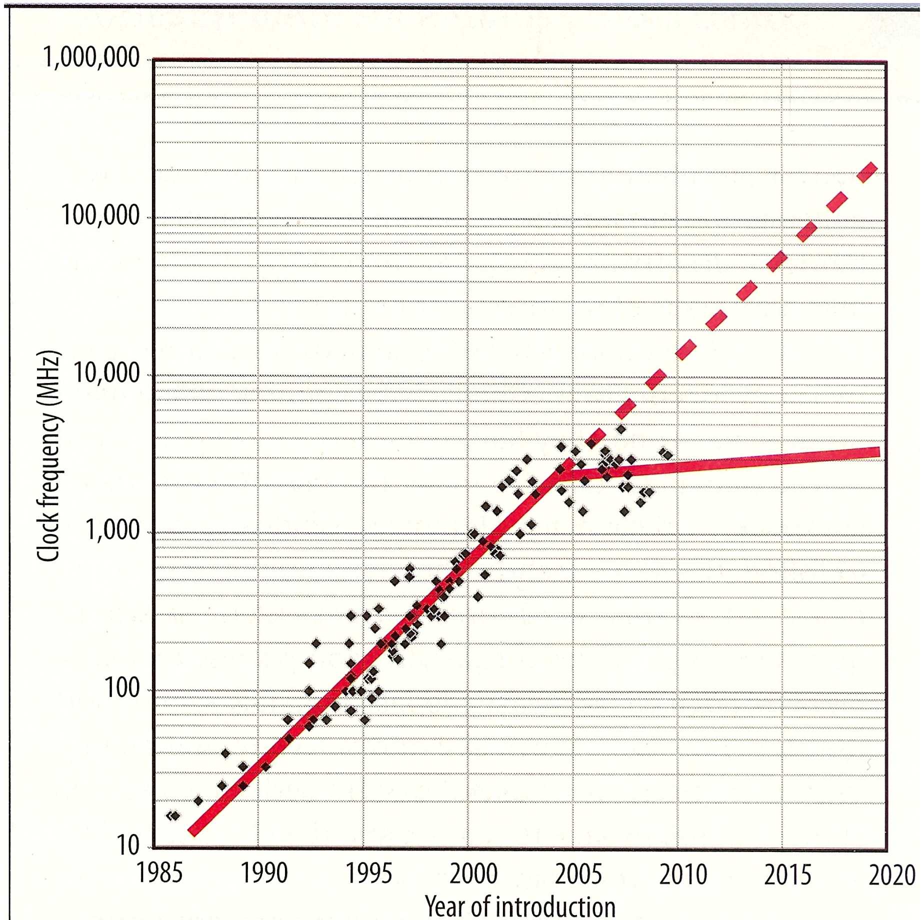
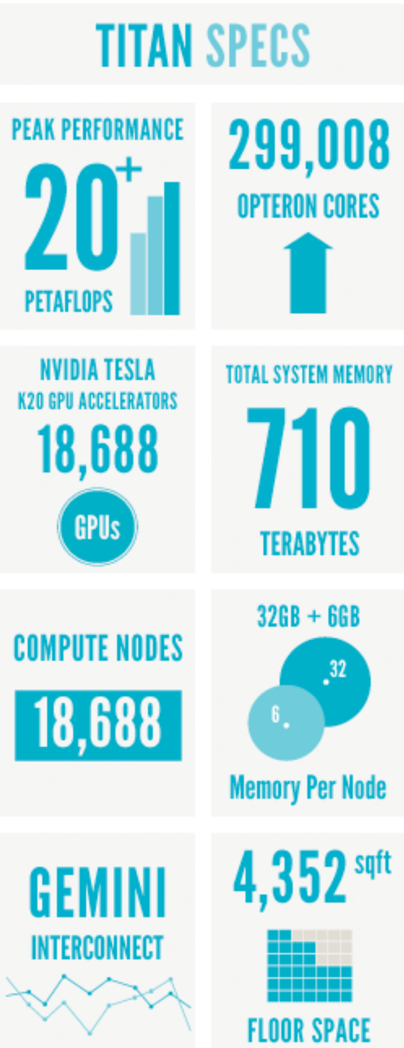
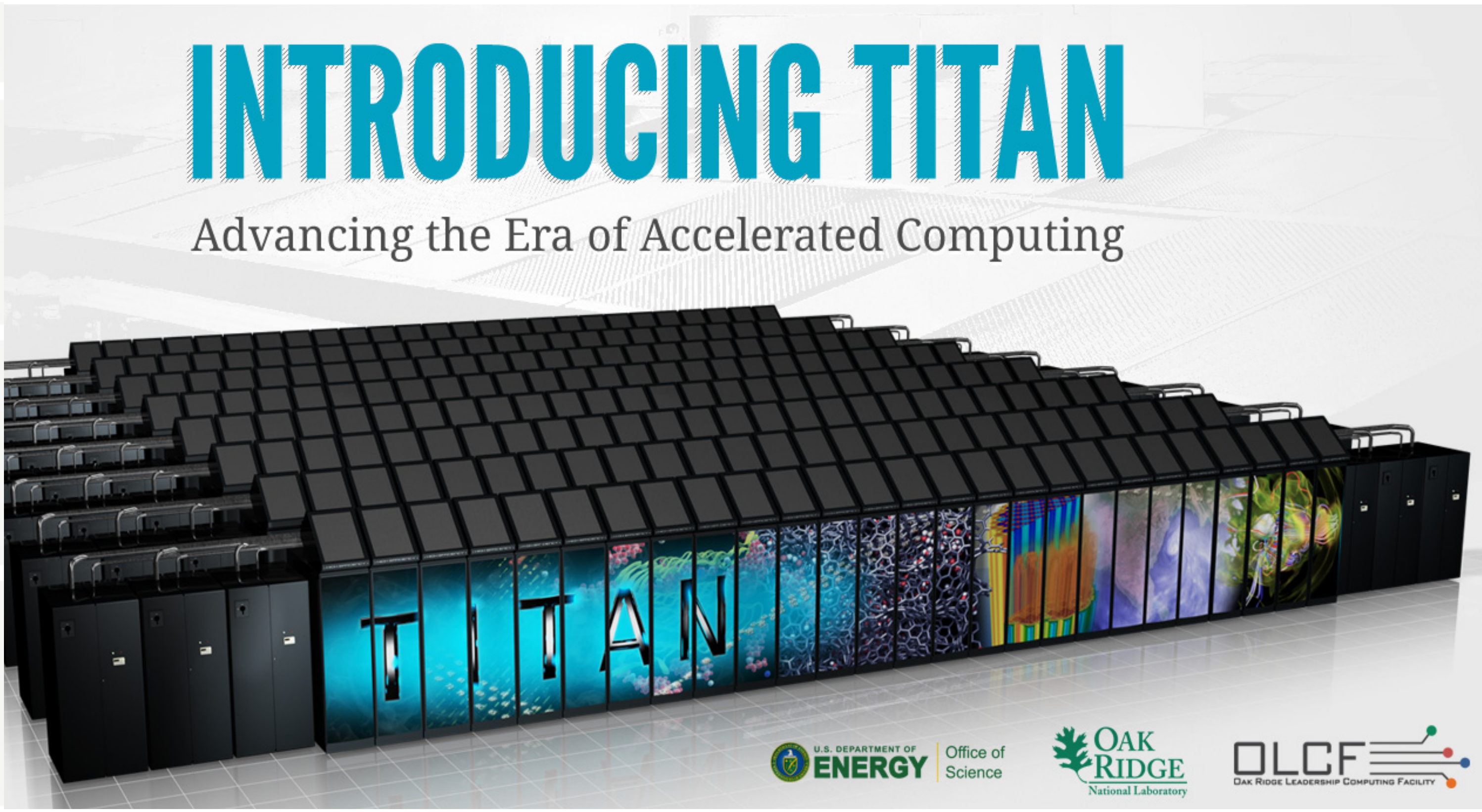


Figure 2. Historical growth in single-processor performance and a forecast of processor performance to 2020, based on the ITRS roadmap. A dashed line represents expectations if single-processor performance had continued its historical trend.

Courtesy IEEE Computer, January 2011, page 33.

Titan Supercomputer at Oak Ridge National Lab in US





- Distributed Computing with 18,688 “nodes”:
  - Multicore (16 cores each) with Vector unit
  - GPU with 64 warps of 32 lanes

AdaCore
The GNAT Pro Company

Example of parallel programming language (ParaSail), with implicit parallelism for divide-and-conquer

Simple cases

Divide and Conquer

```
func Word_Count
  (S : Univ_String; Separators : Countable_Set<Univ_Character> := [' '])
   -> Univ_Integer is
    // Return count of words separated by given set of separators
    case |S| of
      [0] => return 0   // Empty string
      [1] =>
        if S[1] in Separators then
            return 0    // A single separator
        else
            return 1    // A single non-separator
        end if
      [..] =>           // Multi-character string; divide and conquer
        const Half_Len := |S|/2
        const Sum := Word_Count( S[ 1 .. Half_Len ], Separators ) +
          Word_Count( S[ Half_Len <.. |S| ], Separators )
        if S[Half_Len] in Separators
          or else S[Half_Len+1] in Separators then
            return Sum      // At least one separator at border
        else
            return Sum-1    // Combine words at border
        end if
    end case
end func Word_Count
```

AdaCore
The GNAT Pro Company

Count words in a string, given a set of separators , using divide-and-conquer (rather than sequential scan)

S: "This is a test, but | it's a bit boring."

```
    1111111111|2222222222333333333
1234567890123456789|0123456789012345678
```

Separators: [' ', ',', '.']

Word_Count(S, Separators) == ?

```
|S| == 38                // |…| means "magnitude"
Half_Len == 19
Word_Count(S[1 .. 19], Separators) == 5
Word_Count(S[19 <.. 38], Separators) == 4
Sum == 9                 // X <.. Y means (X, Y]
S[19] == 't'             // 't' not in Separators
S[19+1] == ' '           // ' ' is in Separators
return 9
```

Word_Count example in Ada 2012:

Simple cases

Start outer task

```
function Word_Count(S : String; Separators : String) return Natural is
    use Ada.Strings.Maps;
    Seps : constant Character_Set := To_Set(Separators);

    task type TT(First, Last : Natural; Count : access Natural);
    subtype WC_TT is TT;  --  So is visible inside TT
    task body TT is begin
       if First > Last then      --  Empty string
          Count.all := 0;
       elsif First = Last then   --  A single character
          if Is_In(S(First), Seps) then
             Count.all := 0;     --  A single separator
          else
             Count.all := 1;     --  A single non-separator
          end if;
       else  --  Divide and conquer
          ... See next slide
       end if;
    end TT;

    Result : aliased Natural := 0;
 begin
    declare  --  Spawn task to do the computation
       Tsk : TT(S'First, S'Last, Result'Access);
    begin
       null;
    end;  --  Wait for subtask
    return Result;
 end Word_Count;
```

Word_Count example in Ada 2012 (cont'd):

```
function Word_Count(S : String; Separators : String) return Natural is
    use Ada.Strings.Maps;
    Seps : constant Character_Set := To_Set(Separators);
    task type TT(First, Last : Natural; Count : access Natural);
    subtype WC_TT is TT;  --  So is visible inside TT
    task body TT is begin
       if …  --  Simple cases (see previous slide)
       else  --  Divide and conquer
          declare
             Midpoint : constant Positive := (First + Last) / 2;
             Left_Count, Right_Count : aliased Natural := 0;
          begin
             declare  --  Spawn two subtasks for distinct slices
                Left : WC_TT(First, Midpoint, Left_Count'Access);
                Right : WC_TT(Midpoint + 1, Last, Right_Count'Access);
             begin
                null;
             end;  --  Wait for subtasks to complete

             if Is_In(S(Midpoint), Seps) or else
               Is_In(S(Midpoint+1), Seps) then  --  At least one separator at border
                Count.all := Left_Count + Right_Count;
             else  --  Combine words at border
                Count.all := Left_Count + Right_Count - 1;
             end if;
          end;
       end if;
    end TT;
    … See previous slide
end Word_Count;
```

Divide and Conquer

Word_Count example in (hypothetical) Ada 202X:

Simple cases

Divide and Conquer

```
function Word_Count (S : String; Separators : String) return Natural
  with Global => null, Potentially_Blocking => False is
    case S'Length is
     when 0 => return 0; --  Empty string
     when 1 =>           --  A single character
       if Is_In(S(S'First), Seps) then
          return 0;      --  A single separator
       else
          return 1;      --  A single non-separator
       end if;
     when others =>
       declare           --  Divide and conquer
          Midpoint : constant Positive := (S'First + S'Last) / 2;
          Left_Count, Right_Count : Natural;
       begin
          parallel       --  Spawn two tasklets for distinct slices
             Left_Count  := Word_Count (S(S'First .. Midpoint), Separators);
          and
             Right_Count := Word_Count (S(Midpoint+1 .. S'Last), Separators);
          end parallel;  --  Wait for tasklets to complete

          if Is_In(S(Midpoint), Seps) or else
            Is_In(S(Midpoint+1), Seps) then  --  At least one separator at border
             return Left_Count + Right_Count;
          else                               --  Combine words at border
             return Left_Count + Right_Count - 1;
          end if;
       end;
    end case;
end Word_Count;
```

Parallel Block

```
parallel
   sequence_of_statements
{and
   sequence_of_statements}
end parallel;
```

Each alternative is an (explicitly specified) *"parallelism opportunity"* (POp) where the compiler may create a *tasklet,* which can be executed by an *execution server* while still running under the context of the enclosing *task* (same task 'Identity, attributes, etc.). Compiler will complain if any data races or blocking are possible (using Global and Potentially_Blocking aspect information).

cf. ARM 9, Note 1: … *whenever an implementation can determine that the required semantic effects can be achieved when parts of the execution of a given task are performed by different physical processors acting in parallel, it may choose to perform them in this way.*

AdaCore
The GNAT Pro Company

Global (cf. SPARK) and Potentially_Blocking aspects

```
Global => all      -- default within non-pure packages

-- Explicitly identified globals with modes (SPARK 2014)
Global => (Input  => (P1.A, P2.B),
           In_Out => P1.C,
           Output => (P1.D, P2.E))

-- Pkg private, access collection, task/protected/atomic
Global => (In_Out => P3)  -- pkg P3 private data
Global => (In_Out => P1.Acc_Type)  -- acc type
Global => (In_Out => synchronized)

Global => null      -- default within pure packages

Potentially_Blocking [ => True | => False ]
```

Parallel Loop

```
for I in parallel 1 .. 1_000 loop
   A(I) := B(I) + C(I);
end loop;

for Elem of parallel Arr loop
   Elem := Elem * 2;
end loop;
```

Parallel loop is equivalent to parallel block by unrolling loop, with each iteration as a separate alternative of parallel block.

Compiler will complain if iterations are not independent or might block (again, using Global/Potentially_Blocking aspects)

Wonderfully simple and obvious, but what about… ?

- **Exiting the block/loop, or a return statement?**
  - All other tasklets are aborted (need not be preemptive) and awaited, and then, in the case of return with an expression, the expression is evaluated, and finally the exit/return takes place.
  - With multiple concurrent exits/returns, one is chosen arbitrarily, and others are aborted.
- **With a very big range or array to be looped over, wouldn't that create a huge number of tasklets?**
  - Compiler may choose to "chunk" the loop into subloops, each subloop becomes a tasklet (subloop runs sequentially within tasklet).
- **Iterations are not completely independent, but could become so by creating multiple accumulators?**
  - We provide notion of *parallel array* of such accumulators (next slide)

Parallel arrays of accumulators; Map/Reduce

```
declare
   Partial: array (parallel <>) of Float := (others => 0.0);
   Sum_Of_Squares : Float := 0.0;
begin
   for E of parallel Arr loop -- "Map" and partial reduction
      Partial(<>) := Partial(<>) + E ** 2;
   end loop;

   for I in Partial'Range loop  --  Final reduction step
      Sum_Of_Squares := Sum_Of_Squares + Partial (I);
   end loop;

   Put_Line ("Sum of squares of elements of Arr =" &
     Float'Image (Sum_Of_Squares));
end;
```

Parallel array bounds of <> are set to match number of "chunks" of parallel loop in which they are used with (<>) indices. May be specified explicitly.

Map/Reduce short hand

- **Final reduction step will often look the same:**

```
Total := <identity>;
for I in Partial'Range loop
    Total := <op> (Total, Partial);
end loop
```

- **Provide an attribute function 'Reduced to do this:**
  - Total := Partial'Reduced(Reducer => "+", Identity => 0.0);

    or
  - Total := Partial'Reduced; -- *Reducer and Identity defaulted*
- **The 'Reduced attribute may be applied to *any* array when Reducer and Identity are specified explicitly**
- **The 'Reduced attribute may be implemented using a tree of parallel reductions.**

Summary

Summary

- **Parallel programming constructs can simplify taking advantage of modern multi/manycore hardware**
- **Parallel block and Parallel loop constructs are natural solutions for Ada**
- **Global (cf. SPARK 2014) and Potentially_Blocking aspects enable compiler to check for data races and blocking**
- **Parallel arrays and 'Reduced attribute simplify map/reduce sorts of computations.**
- ***Please submit extended abstracts to HILT 2014 by July 5 and come to Portland, OR***