CISTER
Research Center in
Real-Time & Embedded
Computing Systems

Technical Report

Safe Parallel Programming in Ada with Language Extensions

Brad Moore
Luis Miguel Pinho*
Stephen Michell

CISTER-TR-141010
10-20-2014

Safe Parallel Programming in Ada with Language Extensions

Brad Moore, Luis Miguel Pinho*, Stephen Michell

* CISTER Research Center

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail: lmp@isep.ipp.pt,

http://www.cister.isep.ipp.pt

Abstract

The increased presence of parallel computing platforms bringsconcerns to the general purpose domain that were previouslyprevalent only in the specific niche of high-performancecomputing. As parallel programming technologies become moreprevalent in the form of new emerging programming languagesand extensions of existing languages, additional safety concernsarise as part of the paradigm shift from sequential to parallelbehaviour.In this paper, we propose various syntax extensions to the Adalanguage, which provide mechanisms whereby the compiler isgiven the necessary semantic information to enable the implicitand explicit parallelization of code. The model is based on earlierwork, which separates parallelism specification from concurrencyimplementation, but proposes an updated syntax with additionalmechanisms to facilitate the development of safer parallelprograms.

© CISTER Research Center
www.cister.isep.ipp.pt

Safe Parallel Programming in Ada with Language Extensions

S. Tucker Taft
AdaCore,
USA
taft@adacore.com

Brad Moore
General Dynamics
Canada
brad.moore@gdcanada.com

Luís Miguel Pinho
CISTER, ISEP
Portugal
lmp@isep.ipp.pt

Stephen Michell
Maurya Software, Inc.
Canada
Stephen.michell@maurya.on.ca

ABSTRACT
The increased presence of parallel computing platforms brings concerns to the general purpose domain that were previously prevalent only in the specific niche of high-performance computing. As parallel programming technologies become more prevalent in the form of new emerging programming languages and extensions of existing languages, additional safety concerns arise as part of the paradigm shift from sequential to parallel behaviour.

In this paper, we propose various syntax extensions to the Ada language, which provide mechanisms whereby the compiler is given the necessary semantic information to enable the implicit and explicit parallelization of code. The model is based on earlier work, which separates parallelism specification from concurrency implementation, but proposes an updated syntax with additional mechanisms to facilitate the development of safer parallel programs.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features – *Concurrent programming structures.*

General Terms
Performance, Standardization, Languages.

Keywords
Multi-core; programming language; Ada; safe parallelism.

1. INTRODUCTION
There is a continuing trend of exponential growth of computational elements embedded on a single chip. This has led to significant challenges for software designers and implementers. Prior to 2005, the increase in transistor count corresponded to a similar increase in CPU clock speed, which boosted performance of sequential algorithms. Since then, CPU clock speeds have leveled off largely due to power concerns, and chip manufacturers have instead moved towards multicore technologies as a means of achieving performance increases.

This increased parallel execution capability challenges software developers that want to exploit parallelism opportunities that are inherent in algorithms where high performance is critical. To maximize performance, Amdahl's law [1] suggests that it is necessary to minimize the sequential processing in an algorithm, however most mainstream programming languages and development environments lack adequate support for parallelism. In a world of many cores on a single chip executing a single algorithm, the effective use of such capabilities requires a paradigm shift that lets parallel behavior be exposed and captured wherever possible. It is a paradigm that has been for a long time contained within a specialized domain of high-performance computing, but that is now required in all domains of computing systems.

Parallel programming features have a long history. Dijkstra [2], Per Brinch Hansen [3], C.A.R. Hoare [4], and others proposed programming features such as parbegin/parend, cobegin/coend, etc., many years ago. The notion of a light-weight concurrent programming capability remains a key facet of any modern parallel programming capability. In some sense the distinction between "concurrent programming" and "parallel programming" is somewhat arbitrary, but in practice the main difference is that the focus with concurrent programming is to structure a complex program as a set of relatively independent activities to simplify the overall logic, while the primary focus of parallel programming is to divide a compute-intensive problem up to allow it to make better use of parallel hardware. A concurrent restructuring of a sequential program is successful if it simplifies the logic of the program, whereas a parallel restructuring is only successful if it actually speeds up the program on parallel hardware. Concurrent programming has evolved since these early proposals, tending toward explicit task or process constructs, while allowing more explicit control over scheduling of these independent activities. On the other hand, parallel programming has tended to preserve the notion of light-weight, anonymous parallel activities, and to augment the basic cobegin/coend with parallel loops and other data-parallel constructs.

Various parallel programming extensions have been proposed for Ada itself in the past. Mayer and Jahnichen [5] introduce a parallel keyword, which applied to "**for**" loops, allowing a compiler to optimize loop iterations when targeted to a multiprocessor platform. Hind and Schonberg [6] also targeted the optimization of parallel loops, introducing the concept of lightweight (mini) tasks, to reduce the overhead of using tasks for parallelism. Thornley [7] proposed two extension keywords to standard Ada: parallel and single, where parallel was used for declaring that a block or a "**for**" loop would be executed in parallel. Again the emphasis was on keeping the constructs light-weight while introducing parallelism, to ensure that there was a net savings in using the parallel programming features relative to the original sequential program. These efforts occurred in an era when parallel hardware was more the exception than the rule, and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

HILT 2014, October 18 - 21 2014, Portland, OR, USA
Copyright 2014 ACM 978-1-4503-3217-0/14/10...$15.00
http://dx.doi.org/10.1145/2663171.2663181

these largely academic investigations never reached a stage of widespread consideration for adoption into the language standard.

This paper presents a proposal that combines and extends earlier work into concrete syntax and semantics, allowing parallel programs to be expressed safely and naturally in Ada; this work has been performed in the context of the ongoing evolution of the Ada language standard. This work reuses the notion of a *Parallel OPportunity* (POP) and *Tasklet* from [8,9,10]. POPs are places in an algorithm (code) where work can be spawned to parallel executing *workers* that work in concert to correctly execute the algorithm. Tasklets are the notational (logical) units within a task that are executed in parallel with each other. The goals of this proposal are:

1 To permit existing programs to benefit from parallelism through minimal restructuring of sequential programs into ones that permit more parallel execution;
2 To support the development of complete programs that maximize parallelism;
3 To efficiently guide the compiler in the creation of effective parallelism (without oversubscription) with minimal input from the programmer;
4 To support the parallel reduction of results calculated by parallel computation;
5 To avoid syntax that would make a POP erroneous or produce noticeably different results if executed sequentially vs in parallel.

Earlier approaches [8,9,10] sought to avoid changes to Ada syntax and instead provide parallelism hints to the compiler via aspect and pragma annotations. However, it was recognized that such annotations would alter the semantics currently defined in the standard, and support for such aspects would need to be allowed in places not currently allowed in Ada (for example, aspects are not currently allowed to be specified on loop statements). Since the changes needed involved are more significant than simply defining new aspects, we decided to explore possibilities for new syntax that can be directly tied to the new semantics, leaving the semantics for existing syntax as it was previously for the most part.

Furthermore, an alternative paradigm to implicit parallelization is proposed. Introducing parallel notations can increase the likelihood of data races, which can lead to erroneousness. We address this issue by defining `Global` and `Potentially_Blocking` aspects to enable the compiler to provide better static detection of such problems so that they may be eliminated during development.

This paper is organized as follows. Section 2 presents related work while section 3 presents the fundamental model of a tasklet. Section 4 presents a new parallel block construct, while Section 5 addresses parallelization of loops. Section 6 then proposes ways to provide safety of parallel computation, and to enable safe implicit parallelization by the compiler. Finally, Section 7 presents some conclusions and open issues.

2. RELATED WORK

Several new programming languages have been developed recently with parallel programming in mind, and a number of existing languages are investigating how best to add support for parallel programming. Some of the "new" languages are in fact special-purpose *extensions* of existing languages, often with an augmented run-time library, while others are *completely* new designs.

Notable examples of the new languages that are defined as extensions of existing languages are Cilk+ [11] (based on C++), OpenMP [12] (with variants based on C, C++, and Fortran), and OpenCL [13] and CUDA [14] (both being languages that are based on C or C++ and used with Graphics Processing Units, GPUs, and other similar accelerators for general purpose parallel computing). All of these languages make no attempt to improve the semantics of the underlying base language to support parallelism, but add new capabilities on top, which are specifically designed to take advantage of parallel hardware of various sorts. Cilk and OpenMP both use a *fork/join,* divide-and-conquer model where light weight threads can be spawned to perform parts of a larger calculation. In Cilk the light-weight threads are scheduled using a *work-stealing* model where heavier-weight *server* threads, roughly one per physical processor, each serve a queue of light-weight threads, stealing from other servers' queues when their own queue is empty. OpenMP provides a number of different scheduling approaches, providing the programmer more low-level control of how the light-weight threads are mapped to the physical processors. In the GPU languages OpenCL and CUDA, the model is more data driven, where a body of code is identified as a *kernel*, which is to be applied to every item in an array or other data aggregate. OpenCL and CUDA are targeted to environments where there is often separate memory for the main processor(s) and the accelerator(s), and so extra control is provided in placing data in particular parts of memory.

In all of these languages, the emphasis is on giving control to the programmer, with more or less of an attempt to provide a level of portability and abstraction, with Cilk providing the highest level model, and CUDA providing the lowest-level model. There is relatively little left to the compiler to decide, as the programmer determines where new threads are spawned, what code is to be run on the accelerator versus the main processor, etc.

These language extensions have to some degree been the inspiration for efforts to extend the C, C++ and Ada language standards themselves. These efforts are still in their early stages, with the C effort being named CPLEX (for C Parallel Language Extensions) [15], and the C++ effort being documented in a "Technical Specification for C++ Extensions for Parallelism" [16]. As with the other language extensions based on C and C++, no particular effort is made to enhance the underlying semantics of the languages to integrate parallelism. The goal is to give programmers an ability to direct how the compiler could insert parallelism. The programmer remains in control, and the compiler has very little leeway to insert parallelism beyond that which is authorized by the programmer, in large part because the compiler rarely has enough information to perform safe automatic parallelization. These extensions rely on the programmer to worry about data races, and provide few constructs beyond thread-local storage [17] to help identify or minimize such races.

A previous effort to extend Ada [8,10] proposed a fine-grain parallel model, based on the notion of tasklets, which are non-schedulable computation units (similar to Cilk [11] or OpenMP [12] *tasks*). However, in contrast to the C and C++ works, the principle behind this model is that the specification of parallelism is an abstraction that is not fully controlled by the programmer. Instead, parallelism is a notion that is under the control of the compiler and the run-time.

There are a handful of new languages that are not merely extensions of existing languages, but are rather completely new designs. Three notable examples of these are *Go* [18] from Google, *Rust* [19] from Mozilla Research, and *ParaSail* [20] from AdaCore. These languages are built around the notion that all computations will be structured as the coordinated execution of multiple light-weight threads, and each provides constructs specifically designed to simplify the safe interaction of these threads. *Go* provides light-weight *goroutines,* with *channels* for safe communication between them. Go does not prevent data races due to unsynchronized access to shared data, but it makes it relatively easy to structure the program using only goroutines and channels, and thereby avoid the need for directly sharing data.

The *Rust* language provides light-weight *tasks,* with a set of library-based mechanisms for them to interact and communicate, including *futures* [21] and channels. Rust goes further to disallow direct use of shared data between tasks, by enforcing unique pointer *ownership* on global data, while providing more conventional garbage-collected pointer semantics for task-local data. Pointer ownership means that only one pointer is pointing at any given piece of global data, and that pointer is accessible only from one task. The value of a pointer may be *moved* or *sent* from one place or one task to another, leaving a null pointer value behind, to ensure that the uniqueness of each such pointer is preserved. By contrast, pointers into task-local memory may be copied to other task-local variables, meaning that multiple pointers to the same local memory are possible. There are explicitly *unsafe* features which allow these pointer rules to be violated, but so long as these features are avoided, Rust ensures there are no data races.

The *ParaSail* language provides a pervasively parallel model, where the compiler creates light-weight *picothreads* (also called *work items)* as it sees fit, as well as under programmer control. There are no pointers and no global variables, meaning that functions may only update variables passed to them via **var** (in-out) parameters. The compiler treats parameter passing using *hand-off* semantics, similar to that pioneered in the Hermes language [22], where when a variable is passed as a **var** parameter to one function, it is no longer available to be passed to any other function until the original function returns. Similarly, if a variable is passed as a read-only parameter to a function, then the variable may not be passed as a **var** parameter to any other function until the first one returns, though it may be passed to other functions as a read-only parameter. This approach ensures that any ParaSail expression may be evaluated in parallel, so that the ParaSail compiler may insert parallelism where it deems it would be worthwhile. In addition to this implicit parallelism, ParaSail allows programmers to explicitly identify places where parallelism can be inserted, and the compiler will verify that there are no data races introduced by performing the specified code sections concurrently. The compiler might still decide not to actually perform the sections in parallel, but it will always verify the programmer's claim that the sections have no data interdependences. ParaSail also allows the definition of explicitly *concurrent* variables; such variables require the use of software or hardware locks to ensure that concurrent access is properly synchronized. Concurrent variables are allowed to be manipulated concurrently in parallel threads, with no restrictions.

The Ada extensions proposed in this paper, although reusing the tasklet model of [8,10], are closest in spirit to those of Rust and ParaSail, where the compiler has sufficient knowledge to identify all possible data races, and to insert parallelism implicitly where it sees fit. Rather than eliminating global variables and pointers, we have chosen to allow global variable access and pointer dereferences to be specified via the `Global` aspect of a subprogram declaration, to help the compiler determine whether two computations could be safely performed in parallel. Ada's existing synchronization mechanisms based on protected objects, tasks, and atomic objects, provide the equivalent of Rust's library-based synchronization and communication mechanisms, and ParaSail's concurrent objects. Note that support for potentially blocking operations within tasklets is still an open issue (see section 7 below).

Subprograms without any `Global` aspect specified are presumed to update an unspecified number of global variables, and hence cannot be verified to be safe to run in parallel with any subprogram that reads or writes unsynchronized global variables. The overall intent is that introducing explicit parallel constructs into an Ada program will not introduce data races, and that the compiler will also have enough knowledge to introduce parallelism implicitly, when it can identify parallel opportunities that arise in code without explicitly parallel constructs. This is further detailed in section 6.

3. THE TASKLET MODEL

The work in [8] introduced the notion of a *Parallel OPportunity* (POP). This is a code fragment that appears sequential but which can be executed by processing elements in parallel. This could be by-element operations on an array, parallel iterations of a **for** loop over a structure or container, parallel evaluations of subprogram calls, and so on. That work also introduced the notion of a *tasklet* to capture the notion of a single execution trace within a POP, which the programmer can express with special syntax, or the compiler can implicitly create.

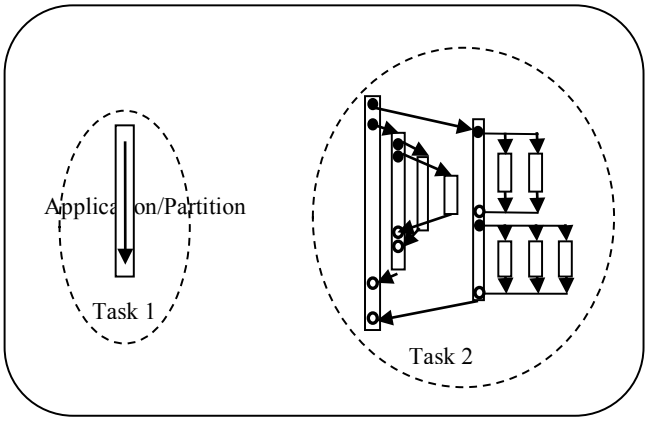

Figure 1. The Tasklet model [10].

As in [10], each Ada task is seen as a graph of execution of multiple control-dependent tasklets (Figure 1), with a fork-join model. Tasklets can be spawned by other tasklets (fork), and need to synchronize with the spawning tasklet (join). In Figure 1, Task 1 denotes the current model of an Ada task where a single thread of control is executing the body of the task; Task 2 denotes the new model, where an Ada task can execute a graph, where rectangles denote tasklets, dark circles fork points, and white circles join points.

An important part of the model is that if the compiler is not able to verify that the parallel computations are independent, then a warning will be issued at compile time (see section 6).
Note that in this model the compiler will identify any code where a potential data race occurs (following the rules for concurrent access to objects as specified in the Language Reference Manual [23, section 9.10]), and point out where objects cannot be guaranteed to be independently addressable. If not determinable at compile-time, the compiler may insert run-time checks to detect data overlap.

Another issue is the underlying run-time. In the proposed model, tasklets are orthogonal to tasks. Regardless of implementation, tasklets are considered to execute in the semantic context of the task where they have been spawned, which means that any operation that identifies a task, such as those in `Task_Identification`, will identify the task in which the tasklet is spawned. This is a major distinction to previous work that left this as an implementation issue. On the other hand, calls by different tasklets of the same task into the same protected object are treated as different calls resulting in distinct protected actions; therefore synchronization between tasklets can be performed using non-blocking protected operations. Note that this is consistent with the current standard which already supports multiple concurrent calls by a single task in the presence of the asynchronous transfer of control capability [23, section 9.7.4].

Our proposed model does not define syntax for the explicit parallelization of individual subprogram calls, since such parallelization can be performed implicitly by the compiler, when it knows that the calls are free of side-effects. This is facilitated by annotations identifying global variable usage on subprogram specifications, a proposal which is detailed in section 6. In sections 4 and 5 we focus on constructs based on explicit specification by the programmer: parallel blocks and loops.

4. PARALLEL BLOCKS

A common parallel language capability is to specify that two or more parts of an algorithm can be executed in parallel with each other. We propose the following syntax for Ada:

```
parallel_block_statement ::=
    parallel
      sequence_of_statements
    and
      sequence_of_statements
   {and
      sequence_of_statements}
    end parallel;
```

Example:

```
declare
   X, Y : Integer;
   Z : Float;
begin
   parallel
      X := Foo(100);
   and
      Z := Sqrt(3.14) / 2.0;
      Y := Bar(Z);
   end parallel;

   Put_Line("X + Y=" &
            Integer'Image(X + Y));
end;
```

In this example, the calculation of Z and Y occur sequentially with respect to each other, but in parallel with the calculation of X. Note that the compiler, using the rules specified in Section 6, may complain if the parallel sequences might have conflicting global side-effects. In this particular case, this means that, at a minimum, either `Foo` or both `Sqrt` and `Bar`, need to be annotated with the `Global` aspect. If only one branch of the construct has `Global` aspects, then they must indicate that that branch does not involve any access to non-synchronized globals; alternatively, both branches must be annotated with non-conflicting `Global` aspects.

The parallel block construct is flexible enough to support recursive usage as well, such as:

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

   parallel
     X := Fibonacci (N - 2);
   and
     Y := Fibonacci (N - 1);
   end parallel;

   return X + Y;
exception
   when others =>
      Log ("Unexpected Error");
end Fibonacci;
```

4.1 Parallel Block Semantics

A parallel block statement encloses two or more sequences of statements (two or more "parallel sequences") separated by the reserved word "**and**". Each parallel sequence represents a separate tasklet, but all within a single Ada task. Task identity remains that of the enclosing Ada task, and a single set of task attributes is shared between the tasklets.

With respect to the rules for shared variables (see section 9.10 in the Ada reference manual [23]), two actions occurring within two different parallel sequences of the same parallel block are *not* automatically sequential, so execution can be erroneous if one such action assigns to an object, and the other reads or updates the same object or a neighboring object that is not independently addressable from the first object. The appropriate use of atomic, protected, or task objects (which as a group we will call *synchronized* objects) can be used to avoid erroneous execution. In addition, the new `Global` and `Potentially_Blocking` aspects may be specified to enable the static detection of such problems at compile time (see section 6).

Any transfer of control out of one parallel sequence will initiate the aborting of the other 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.

If an exception is raised by any of the parallel sequences, it is treated similarly to a transfer of control, with the exception being propagated only after all the other sequences complete normally

or due to abortion. If multiple parallel sequences raise an exception before completing, one is chosen arbitrarily and the others are aborted.

The parallel block completes when all of the parallel sequences complete, either normally or by being aborted. Note that aborting a tasklet need not be preemptive, but should prevent the initiation of further nested parallel blocks or parallel loops.

We considered allowing the parallel block to be preceded with an optional declare part, and followed with optional exception handlers, but it was observed that it was more likely to be useful to have objects that are shared across multiple parallel sequences to outlive the parallel block, and that having exception handlers after the last parallel sequence could easily be misconstrued as applying only to the last sequence. Therefore we reverted to the simpler syntax proposed above. This simpler syntax is also more congruous with the syntax for select statements.

5. PARALLEL LOOPS

In most compute-intensive applications, a significant proportion of the computation time is spent in loops, either iterating over arrays/container data structures, or systematically searching a large solution space. To benefit from parallel hardware, the computation associated with a loop should be spread across the available processors. One approach, presuming the iterations of the loop have no data dependences between them, is to treat each iteration of the loop as a separate tasklet, and then have the processors work away on the set of tasklets in parallel. However, this introduces overhead from the queuing and de-queuing of work items, and the communication of results from each work item. Furthermore, there often are data dependences between iterations, and creating a separate work item for each iteration can introduce excessive synchronization overhead to deal safely with these interdependences. Therefore, it is common to break large arrays, and/or the loops that iterate over them, into *chunks* (or *slices* or *tiles*), where each chunk is processed sequentially, but multiple chunks can be processed in parallel with one another. Fig. 2 shows how a compiler run-time might decide to break a specific loop into chunks to allow up to four parallel workers to process the loop. Although the chunks are all equal size in this example, the run-time may choose different chunk sizes for each chunk, which would be needed if the number of chunks did not divide evenly into the number of iterations, for example.

```
for I in parallel 1 .. 1000 loop
   Process (I);
end loop;
```

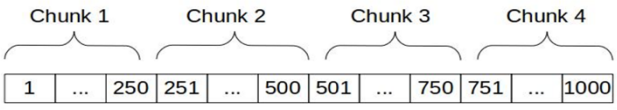

Figure 2. Example of chunking a loop (for 4 parallel workers).

For Ada, we propose giving the programmer some degree of control over the parallelization of **for** loops[1] into appropriately sized chunks, but without requiring that they specify the exact chunk size or the number of chunks. In addition, to deal with data dependences, we would like to provide support for per-thread copies of the relevant data, and a mechanism of reducing these multiple copies down to a final result at the end of the computation.

To indicate that a loop is a candidate for parallelization, the reserved word "**parallel**" may be inserted immediately after the word "**in**" or "**of**" in a "**for**" loop, at the point where the "**reverse**" reserved word is allowed. Such a loop will be broken into chunks, where each chunk is processed sequentially. For data that is to be updated within such a parallelized loop, the notion of a *parallel array* is provided, which corresponds to an array with one element per chunk of a parallel loop. For example, here is a simple use of a parallelized loop, with a parallel array of partial sums (with one element per chunk), which are then summed together (sequentially) to compute an overall sum for the array:

```
declare
   Partial_Sum : array (parallel <>)
                 of Float
               := (others => 0.0);
   Sum : Float := 0.0;
begin
   for I in parallel Arr'Range loop
     Partial_Sum(<>) := Partial_Sum(<>) +
                        Arr(I);
   end loop;

   for J in Partial_Sum'Range loop
     Sum := Sum + Partial_Sum(J);
   end loop;
   Put_Line ("Sum over Arr = " &
             Float'Image (Sum));
end;
```

In this example, the programmer has merely specified that the `Partial_Sum` array is to be a parallel array (with each element initialized to 0.0), but has not specified the actual bounds of the array, using "<>" instead of an explicit range such as "`1 .. Num_Chunks`". In this case, the compiler will automatically select the appropriate bounds for the array, depending on the number of chunks chosen for the parallelized loops in which the parallel array is used.

When a parallel array is used in a parallelized loop, the programmer is not allowed to specify the specific index, but rather uses "<>" to indicate the "current" element of the parallel array, appropriate to the particular chunk being processed. In the above case, we see "`Partial_Sum(<>)`" indicating we are accumulating the sum into a different element of the `Partial_Sum` in each distinct chunk of the loop. In this example, if the loop were to be processed in two chunks then the Partial_Sum array would contain two elements, where the first element could contain the sum for the first half of the array, and the second element would then contain the sum for the last half of the array.

The user may explicitly control the number of chunks into which a parallelized loop is divided by specifying the bounds of the parallel array(s) used in the loop. All parallel arrays used within a given loop must necessarily have the same bounds. If parallel arrays with the same bounds are used in two consecutive parallelized loops over the same container or range, then the two loops will be chunked in the same way. Hence, it is possible to pass data across consecutive loops through the elements of a parallel array that is common across the loops. For example, here is a pair of parallelized loops that produce a new array that is the cumulative sum of the elements of an initial array. The parallel

[1] While loops cannot be easily parallelized, because the control variables are inevitably global to the loop.

arrays Partial_Sum and Adjust are used to carry data from the first parallelized loop to the second parallelized loop:

```
declare
   Partial_Sum: array (parallel <>) of Float
              := (others => 0.0);
   Adjust: array(parallel Partial_Sum'Range)
           of Float
         := (others => 0.0);
   Cumulative_Sum: array (Arr'Range)
                   of Float
                 := (others => 0.0);
begin
   --  Produce cumulative sums within chunks
   for I in parallel Arr'Range loop
     Partial_Sum(<>) := Partial_Sum(<>) +
                        Arr(I);
     Cumulative_Sum(I) := Partial_Sum(<>);
   end loop;

   --  Compute adjustment for each chunk
   for J in Partial_Sum'First..
                    Partial_Sum'Last-1 loop
     Adjust(J+1) := Adjust(J) +
                    Partial_Sum(J);
   end loop;

   --  Adjust elements of each chunk
   for I in parallel Arr'Range loop
     Cumulative_Sum(I):= Cumulative_Sum(I)+
                         Adjust(<>);
   end loop;

   --  Display result
   Put_Line("Arr, Cumulative_Sum");

   for I in Cumulative_Sum'Range loop
      Put_Line(Float'Image(Arr(I)) & ", " &
            Float'Image(Cumulative_Sum(I)));
   end loop;
end;
```

Note that this feature eliminated the need to reference two different elements of the same array (element I and element I – 1) within any of the parallel loop bodies. This reduces expression complexity and eliminates data race issues at chunk boundaries, where the I – 1th element could refer to an element of another chunk.

Note also that chunking is not explicit in parallelized loops, and in the above example, the compiler is free to use as few or as many chunks as it decides is best, though it must use the same number of chunks in the two consecutive parallelized loops because they share parallel arrays with common bounds.

If Arr had been declared as;

```
Arr : array (1 .. 10) of Float
    :=  (1 => 1.0, 2 => 2.0, 3 => 3.0,
         4 => 4.0, 5 => 5.0, 6 => 6.0,
         7 => 7.0, 8 => 8.0, 9 => 9.0,
         10 => 10.0);
```

and the loops were processed as two chunks of 5 iterations each, then after the first loop the following values would have been stored:

```
Cumulative_Sum:
   (1 => 1.0, 2 => 3.0, 3 => 6.0, 4 => 10.0,
    5 => 15.0, 6 => 6.0, 7 => 13.0,
    8 => 21.0, 9 => 29.0, 10 => 39.0)
Partial_Sum:
   (1 => 15.0, 2 => 39.0)
```

After processing the second and remaining loops, the following values would have been stored:

```
Adjust:
   (1 => 0.0, 2 => 15.0)
Cumulative_Sum:
   (1 => 1.0, 2=> 3.0, 3 => 6.0, 4 => 10.0,
    5 => 15.0, 6 => 21.0, 7 => 34.0,
    8 => 36.0, 9 => 44.0, 10 => 54.0)
```

The programmer could exercise more control over the chunking by explicitly specifying the bounds of Partial_Sum, rather than allowing it to default. For example, if the programmer wanted these parallelized loops to be broken into "N" chunks, then the declarations could have been:

```
declare
   Partial_Sum : array (parallel 1..N)
                 of Float
               := (others => 0.0);
   Adjust: array(parallel Partial_Sum'Range)
           of Float := (others => 0.0);
   ...
```

Parallel arrays are similar to normal arrays, except that they are always indexed by Standard.Integer, and they are likely to be allocated more widely spaced than strictly necessary to satisfy the algorithm, to avoid sharing cache lines between adjacent elements. This wide spacing means that two parallel arrays might be interspersed, effectively turning a set of separate parallel arrays with common bounds, into an array of records, with one record per loop chunk, from a storage layout point of view.

Note that the same rules presented for parallel blocks (subsection 4.1) apply to the update of shared variables and the transfer of control to a point outside of the loop, and for this purpose each iteration (or chunk) is treated as equivalent to a separate sequence of a parallel block.

5.1 Automatic Reduction of a Parallel Array

As is illustrated above by the first example, it will be common for the values of a parallel array to be combined at the end of processing, using an appropriate *reduction* operator. In this case, the Partial_Sum parallel array is reduced by "+" into the single Sum value. Because this is a common operation, we are providing a language-defined attribute which will do this reduction, called "Reduced." This can eliminate the need to write the final reduction loop in the first example, and instead we could have written simply:

```
Put_Line ("Sum over Arr = " &
        Float'Image (Partial_Sum'Reduced));
```

The Reduced operator will automatically reduce the specified parallel array using the operator that was used in the assignment statement that computed its value -- in this case the "+" operator appearing in the statement:

```
Partial_Sum(<>) := Partial_Sum(<>) +
                   Arr(I);
```

For large parallel arrays, this reduction can itself be performed in parallel, using a tree of computations. The reduction operator to be used can also be specified explicitly when invoking the `Reduced` attribute, using a `Reducer` and optionally an `Identity` parameter. For example:

```
Put_Line ("Sum over Arr = " &
    Float'Image (Partial_Sum'Reduced(
                 Reducer => "+",
                 Identity => 0.0)));
```

The parameter names are optional, so this could have been:

```
Put_Line("Sum over Arr = " &
    Float'Image (Partial_Sum'Reduced(
                                "+", 0.0)));
```

Note that an explicit Reducer parameter is *required* when the parallelized loop contains multiple operations on the parallel array. More generally, the parameterized `Reduced` attribute with an explicit Reducer parameter may be applied to any array, and then the entire parallel reduction operation will be performed. Hence the first example could have been completely replaced with simply:

```
Put_Line ("Sum over Arr = " &
   Float'Image (Arr'Reduced("+", 0.0)));
```

The examples shown here involve simple elementary types, but the `Reduced` attribute can similarly be applied to complex user-defined types such as record types, private types, and tagged types. The Reducer parameter of the Reduced attribute simply identifies the subprogram to use for the reduction operation.

6. PARALLELISM AND CONCURRENCY SAFETY

One of the strengths of Ada is that it was carefully designed to allow the compiler to detect many problems at compile time, rather than at run time. Programming for parallel execution in particular is an activity that requires care to prevent data races and deadlocks. It is desirable that any new capabilities added to the language to support parallelism also allow the compiler to detect as many such problems as possible, as an aid to the programmer in arriving at a reliable solution without sacrificing performance benefits.

A common source of erroneousness in languages that support concurrency and parallelism are data races, which occur when one thread of execution attempts to read or write a variable while another thread of execution is updating that same variable. Such a variable is *global* in the sense that it is globally accessible from multiple threads of execution. In the current Ada standard, threads of execution are *tasks*. In this proposal, tasklets are another form of execution threads.

Eliminating concurrency and parallelism problems associated with non-protected global variables is an important step towards improving the safety of the language. To that end, we propose the addition of a `Global` aspect to the language. The main goal in the design of this aspect is to identify which global variables and access-value dereferences a subprogram might read or update.

The inspiration for this aspect comes from the SPARK language [24], which has always had global annotations. Earlier versions of SPARK augmented a subset of Ada with annotations added as specially formatted comments, which were used for static analysis by the proof system. With the addition of aspects to Ada in Ada 2012, SPARK 2014 has changed its annotations to use aspects, including the "`Global`" annotation.

To encourage convergence with SPARK we are starting from the SPARK `Global` aspect. However, for Ada, it is necessary to extend this idea to cover a broader spectrum of usage, since Ada is a more expressive programming environment than SPARK.

The `Global` aspect in SPARK 2014 is applied to subprogram specifications, and is of the following form;

```
with Global =>(Input => ...,
               In_Out => ..., Output => ...)
```

where "…" is either a single name, or a parenthesized list of names, and `Input`, `In_Out`, and `Output` identify the global variables of the program that are accessed by this subprogram, in read-only, read-write, or write-only mode, respectively. If there are no global variables with a particular parameter mode, then that mode is omitted from the specification. If there are only global inputs, and no outputs or in-outs, then this syntax can be further simplified to:

```
with Global => …
```

where again "..." is a single name, or a parenthesized list of names.

Finally, if there are no global inputs, in-outs, nor outputs, then:

```
with Global => null
```

is used.

We needed to refine the notion of SPARK's `Global` aspect, because SPARK does not support access types, and because SPARK relies on an elaborate mechanism for handling the abstract "state" of packages. The refinements we are proposing are the following:

1. Allow the name of an access type A (including "**access** *T*") to stand-in for the set of objects described by: (`for all X convertible to A => X.all`)
2. Allow the name of a package P to stand-in for the set of objects described by: (`for all variables X declared in P => X`)
3. Allow the word **synchronized** to be used to represent the set of global variables that are tasks, protected objects, or atomic objects.

Note that references to global constants do not appear in `Global` annotations.

In the absence of a global aspect, the subprogram is presumed to read and write an unspecified set of global variables, including non-synchronized ones.

Another issue for parallel safety is the aliasing of parameters with other parameters and with globals. Ada 2012 has some rules relating to aliasing that apply to the use of functions with out and in-out parameters, which reduce the problem [23, section 6.4.1]. There are also the new attributes `Has_Same_Storage` and `Overlaps_Storage` [23, section 13.3(73.1/3-73.10/3)]. In the absence of preconditions such as:

```
with Pre => not X'Overlaps_Storage(Y)
```

the compiler must presume that two parameters that are passed by reference, or a by-reference parameter and a global, might overlap if their types imply that is possible.

Given a `Global` aspect, and presuming appropriate use of `Overlaps_Storage`, the compiler is able to check for potential data races at compile-time. Our proposal does not specify whether such checks are required in all cases, or only in the presence of some sort of named "restriction."

If one wants to know whether a subprogram has side-effects, it is important to know about *all* data that might be read or written. Access types introduce difficulties in determining such side-effects, since the side-effects might result after a dereference of a series of pointers to reach an object to be updated. Our proposal addresses this by allowing the programmer to specify the name of an access type in a `Global` aspect. This would be essentially equivalent to writing something like;

```
Global => (In_Out => *.all)
```

except we can be more specific about the type of the access values being dereferenced.

For example, consider a visible access type declared as;

```
type Acc is access T;
```

and a subprogram that has a value of type `Acc` in local variable `Local`, which it then uses to read and update an object via `Local.all`. It would not be very useful to write:

```
Global => (In_Out => Local.all)
```

since "`Local`" means nothing to the caller. But it could write:

```
Global => (In_Out => Acc)
```

to indicate that the caller should be aware that a call on this subprogram is updating some object by dereferencing an access value of type `Acc`. Another problematic case involves specifying in a `Global` aspect a variable that is declared inside a package body. Directly naming such a variable would not have meaning to the caller of the subprogram, and would violate encapsulation. Similarly, suppose an access type is declared inside the body or private part of package P. In both these cases, we treat the private updatable objects as a part of the overall state of package P. We then simply indicate that the subprogram is updating some or all of the state of package P:

```
Global => (In_Out => P)
```

Now suppose that the objects being updated are all protected or atomic objects. Then the caller doesn't really need to worry about which objects are being read or updated. It is always safe to call the subprogram concurrently. It has some side effects, so you cannot assume it is a "pure" subprogram. In this case, we could describe the effects as:

```
Global => synchronized
```

if it only reads synchronized objects, or:

```
Global => (In_Out => synchronized)
```

if it might update synchronized objects as well.
One might be concerned that the number of globals in a subprogram higher in the call structure of a larger program might be unmanageable to specify in a `Global` aspect. To address this concern we propose a shorthand for the `Global` aspect:

```
Global => (In_Out => all)
```

where "**`all`**" represents all global variables. If the number of non-synchronized globals does get large, then it is likely that the subprogram cannot be used in a parallel context anyway, hence using **`all`** is generally adequate. By default, the global aspect is `(In_Out => all)` for normal subprograms, and **`null`** for subprograms in a declared-pure package.

Another important piece of knowledge the caller of a subprogram might need to know is whether or not the call is *potentially blocking*. The Ada language defines *potentially blocking* operations to include **select** statements, **accept** statements, **delay** statements, **abort** statements, and task creation or activation, among others. When executing parallel code, potentially blocking operations can cause problems such as deadlocks. Currently there is no standard way in Ada to specify that a subprogram is potentially blocking. If the compiler cannot statically determine that a subprogram call is potentially blocking, the programmer has to rely on run-time checking to detect these sorts of problems. We propose the addition of a boolean `Potentially_Blocking` aspect that can be applied to subprogram specifications to indicate whether they use constructs that are potentially blocking or call other subprograms that have the `Potentially_Blocking` aspect with a value of `True`. Such an aspect enhances the safety of parallel calls, and also generally improves the safety of Ada, since it allows the compiler to statically detect more problems involving calls on potentially blocking subprograms. The default value for the `Potentially_Blocking` aspect is `True`.

We also propose that these defaults can be overridden for a package by allowing these aspects to be specified at package level, with the meaning that they establish a default for all subprograms in the package. For example,

```
package My_Stuff
   with Global => (In_Out => Synchronized),
        Potentially_Blocking => False
is
   procedure Do_Something (X : in out T;
                           Y : in U);

   function Query_Something (A : T)
         return Z;
      ...
end My_Stuff;
```

Indicates that all subprograms in package `My_Stuff` involve access to synchronized globals, and all of these calls are not potentially blocking calls (in particular these cannot include entry calls, delays, select statements, etc. [23, section 9.5.1]). Such an annotation would alleviate the need to repeat the `Global` or `Potentially_Blocking` aspect on each subprogram, as long as the package-level default is appropriate for that subprogram.

In the absence of such an explicit package-wide default, the default for `Potentially_Blocking` would be `True`, and the default for `Global` would be `(In_Out => all)` in a normal package, and **`null`** in a declared-pure package.

6.1 Safe Implicit Parallelization

Given the information in the `Global` and `Potentially_Blocking` aspects, the compiler now has enough information to determine whether two constructs can be safely executed in parallel. When the programmer explicitly specifies that two constructs should be executed in parallel, the

compiler can use this knowledge to give appropriate warnings wherever data races are possible. However, it can be a burden on the programmer to add explicitly parallel constructs everywhere in a large program where parallel execution is safe. Therefore, this proposal is designed to enable safe *implicit* parallelization of suitably annotated Ada programs.

In general, implicit parallelization can be modeled as the compiler implicitly transforming the algorithm to use explicit parallel constructs. To determine whether data races are possible, the compiler will make conservative assumptions about each subprogram call. It will assume that each (non-synchronized) variable, package, or access collection identified in the subprogram's `Global` aspect, and each by-reference actual parameter in the call, is accessed in its entirety without any synchronization. If there is any overlap between the objects potentially accessed in two constructs, including any nested calls, the constructs will not be candidates for a transformation that would have them potentially running in parallel.

In addition to rules to prevent the introduction of data races, we also currently disallow the implicit introduction of tasklets that invoke potentially blocking operations, because we presume that blocking a tasklet might block the entire task. Therefore the compiler is not permitted to parallelize two constructs where either involves calls on potentially blocking operations.

Note that the compiler could introduce temporary variables to hold the result of parallel evaluations of subexpressions of a single larger expression, to enable a further transformation. For example, given … `F(X) + G(Y)`… the compiler could transform this to:

```
declare
   T1, T2 : Float;
begin
   parallel
      T1 := F(X);
   and
      T2 := G(Y);
   end parallel;
   … T1 + T2 …
end;
```

where `T1 + T2` is being substituted for what was originally `F(X) + G(Y)`. Other possible transformations would be to change a sequential loop into a parallel loop. In each case, these transformations would only be performed when the compiler can ensure it is not introducing potential data races as a result.

7. CONCLUSIONS AND OPEN ISSUES

This proposal provides an integrated model for safe and natural parallel computation in Ada, adding specific new parallel syntax, that is integrated with the existing syntax of Ada 2012. It provides mechanisms to parallelize blocks and "**for**" loops, as well as syntax to identify potentially shared state.

The following open topics are identified for future work:

- Containers that are to have cursors updated by some tasklet(s) in a parallel computation must be implemented in ways that support such parallel update, with mechanisms to guarantee safe access and update of the cursors by multiple tasklets.
- Ada provides a formal notion of independently addressable components for composite objects, including arrays that satisfy concurrent access requirements (ARM [23] 9.10 and C.6). It is likely that this is sufficient for safe access by tasklets of neighboring components, but more work is required for confirmation. We do not address the requirements to allocate memory for arrays, records, or containers so that access by tasklets on separate cores is optimized to avoid cache contention or similar overheads.
- Whether to support potentially blocking operations within tasklets is yet to be determined (for now we limit tasklets to invoking subprograms where `Potentially_Blocking` is False). Some algorithms might be written using explicit synchronization of tasklets between phases, but explicit blocking synchronization between tasklets puts the algorithm at risk of deadlock with certain mappings of tasklets to underlying computational elements (for example execution of the "parallel" code by a strictly sequential execution may block the task with no way to release it).
- The mapping of tasklets to heterogeneous computational elements that do not match the uniform memory access processor model. Such computation units are becoming more prevalent. Ada's distribution model with partitions and inter-partition communication subsystems may be able to be mapped into a support environment that allows the execution of tasklets across such a system.
- How to use tasklets in a real time domain. Obviously, precise control of the mapping of tasklets to underlying tasks and/or processors is a likely requirement in such a system. Additional syntax and restrictions may be required if parallel computation is to be useable in this environment.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments. This work was partially supported by General Dynamics, Canada; the Portuguese National Funds through FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within projects FCOMP-01-0124-FEDER-037281 (CISTER) and AVIACC (ref. FCOMP-01-0124-FEDER-020486); the European Union (EU) FP7 program under grant agreement n° 611016 (P-SOCRATES); and by FCT and EU ARTEMIS JU, within project ARTEMIS/0001/2013, JU grant nr. 621429 (EMC2).

BIBLIOGRAPHY

[1] G. M. Amdahl. Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities. In AFIPS Conference Proceedings, pages 483–485, 1967.

[2] E. W. Dijkstra. 1965. Cooperating Sequential Processes, Technical Report Ewd-123. Technical Report.

[3] P. B. Hansen. 1973. Concurrent Programming Concepts. ACM Comput. Surv. 5, 4 (December 1973), 223-245.

[4] C. A. R. Hoare (1978). "Communicating sequential processes". Communications of the ACM 21 (8): 666–677.

[5] H. G. Mayer, S. Jahnichen, "The data-parallel Ada run-time system, simulation and empirical results", Proceedings of Seventh International Parallel Processing Symposium, April 1993, Newport, CA, USA, pp. 621 - 627.

[6] M. Hind , E. Schonberg, "Efficient Loop-Level Parallelism in Ada", Proceedings of TriAda 91, October 1991.

[7] J. Thornley, "Integrating parallel dataflow programming with the Ada tasking model". Proceedings of TRI-Ada '94, Charles B. Engle, Jr. (Ed.). ACM, New York, NY, USA.

[8] S. Michell, B. Moore, L. M. Pinho, “Tasklettes – a Fine Grained Parallelism for Ada on Multicores”, International Conference on Reliable Software Technologies - Ada-Europe 2013, LNCS 7896, Springer, 2013.

[9] S. Michell, B. Moore, L. M. Pinho, “Real-Time Programming on Accelerator Many-Core Processors”, Proceedings of the High-Integrity Language Technologies conference (HILT 2013), November 2013.

[10] L. M. Pinho, B. Moore, S. Michell, “Parallelism in Ada: status and prospects”, International Conference on Reliable Software Technologies - Ada-Europe 2014, LNCS 8454, Springer, 2014.

[11] Intel Corporation, Cilk Plus, https://software.intel.com/en-us/intel-cilk-plus

[12] OpenMP Architecture Review Board, “OpenMP Application Program Interface”, Ver-sion 4.0, July 2013

[13] OpenCL (Open Computing Language), http://www.khronos.org/opencl

[14] NVIDIA, “NVIDIA CUDA Compute Unified Device Architecture”, Version 2.0, 2008

[15] CPLEX, C Parallel Language EXtensions study group, archives at http://www.open-std.org/mailman/listinfo/cplex

[16] Working Draft, Technical Specification for C++ Extensions for Parallelism, available at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3960.pdf

[17] D. C. Schmidt, T. H. Harrison, and N. Pryce, “Thread-specific Storage: an Object Behavioral Pattern for Efficiently Accessing per-Thread State,” *C++ Gems II*, (Robert Martin, ed.), SIGS, NY, 1999; http://www.dre.vanderbilt.edu/~schmidt/PDF/TSS-pattern.pdf, retrieved 11-Jun-2014

[18] Google Corporation, The Go Programming Language, http://golang.org/

[19] Mozilla Research, The Rust Programming Language, http://www.rust-lang.org

[20] ParaSail – Parallel Specification and Implementation Language, http://parasail-programming-language.blogspot.com

[21] B. Liskov and L. Shrira, *Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems*. Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation; Atlanta, Georgia, United States, pp. 260–267.

[22] W. Korfhage, A. P. Goldberg, "Hermes Language Experiences," Software—Practice And Experience, Vol. 25(4), 389–402 (April 1995)

[23] ISO IEC 8652:2012. Programming Languages and their Environments – Programming Language Ada. International Standards Organization, Geneva, Switzerland, 2012

[24] J. Barnes. High Integrity Software: The SPARK Approach to Safety and Security. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.