Conference Paper

Reduction of Parallel Computation in the Parallel Model for Ada

S. Tucker Taft

Brad Moore

Luis Miguel Pinho*

Stephen Michell

*CISTER Research Center

CISTER-TR-160403

2016/04/11

Reduction of Parallel Computation in the Parallel Model for Ada

S. Tucker Taft, 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: +351.22.8321159

E-mail: lmp@isep.ipp.pt

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

Abstract

One of the most common program constructs to be parallelized (and where most of the times gains are obtained) isparallel loops. This paper discusses different approaches to support loop parallelization in the fine-grained parallelism model for Ada currently being proposed, focusing in particular in the issues related to reducing operations.

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

Reduction of Parallel Computation in the Parallel Model for Ada

Tucker Taft
AdaCore
USA
taft@adacore.com

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

Luis Miguel Pinho
CISTER/ISEP
Portugal
lmp@isep.ipp.pt

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

Abstract

One of the most common program constructs to be parallelized (and where most of the times gains are obtained) is parallel loops. This paper discusses different approaches to support loop parallelization in the fine-grained parallelism model for Ada currently being proposed, focusing in particular in the issues related to reducing operations.

1 Introduction

This work is part of the ongoing effort to incorporate fine-grained parallelism in Ada [1-2]. In the current proposal, the language is augmented so that Ada tasks are seen as graphs of execution of multiple control-dependent code fragments (whose execution trace is denoted as tasklet), under a fully-strict fork-join model. Although the programmer is in control of the specification of potential parallelism, the compiler and run-time co-operate to provide parallel execution, when possible. Actual parallel execution is based on the notion of abstract executors, which carry the actual execution of Ada tasks in the platform. The description of the proposal is outside of the scope of the paper, and the reader is referred to [1-2] and previous papers for the definition of the tasklet model, the syntax and semantic of parallel programs using the model, and the underlying execution model.

This paper focuses on the approach to be used to support parallelization of *for* loops and to support the (also potentially parallel) reduction of parallel computations. Loops are one of the principal constructs amenable to parallelization within a computer program, with loop iterations spread across the available processors, if there are no dependencies between iterations (or even if they exist). Parallelization, however, also introduces overhead (queuing and de-queuing of work items, synchronizing results, …), therefore parallelization frameworks allow the division of the data into several blocks (usually called *chunks* or *slices* or *tiles*), being each one processed sequentially, and multiple chunks executed in parallel.

Fig. 1 (from [2]) shows an example where for a loop where the programmer only annotates as parallel using the "**parallel**" keyword to denote that iterations are re-orderable thus thy can be executed in parallel. The implementation chooses to split the iterations into chunks[1] to allow up to four parallel executors to process the loop.

[1] As noted in [2] chunks do not need to be of equal size.

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

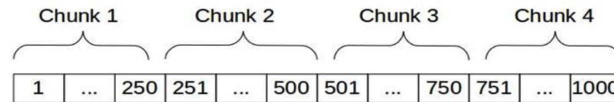

Figure 1. Example of chunking a loop (for 4 parallel workers).

Note that “up to four parallel workers” does not mean that the four chunks will all be executed in parallel to each other, even in a platform with 4 cores. It might happen that due to other tasks, only a subset of the cores are available, thus some of the executors are scheduled sequentially in the same core. Or, the runtime may not have 4 executors available and may decide to execute more than one chunk in the same executor. Splitting the chunks may even be a dynamic operation, guided by the actual execution profile in the runtime.

Consider the case of the following sequential loop [2]:

```
Sum := 0.0;

for I in Arr'Range loop
  Sum := Sum + Arr(I);
end loop;
```

Parallelizing the loop with each iteration updating the shared `Sum` variable would only work if accesses to the variable are protected. But this in fact would serialize execution, and introduce more overhead than gain. This could be addressed by re-writing the loop to allow for different variables to be used in each parallel execution, and summing them in the end.

The work in [2] allows the programmer to use parallel loops with per-chunk or per-executor/thread copies of the data, without needing to explicitly handle the code restructuring. 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 [2]:

```
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);
```

[2] In the examples in the paper, the simple “+” operation is used as an example. Nevertheless, this is an operation which a compiler can implicitly parallelize without needing to explicitly be told of reduction operators or associative and commutative properties. However, it is used as an example of more complex, and user-defined functions, where the compiler will not know of these operators and properties.

```
    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;
```

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 (instead of <>). All parallel arrays used within a given loop must necessarily have the same bounds.

As illustrated above, it is 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, [2] proposes a language-defined attribute which will do this reduction, called "`Reduced`." Thus, we 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)));
```

This makes the full example become (the `Sum` variable is no longer required):

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

```
end loop;

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, or the operation is not easy for the compiler to infer. 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.

2 Ordering of parallel computations

Although many loops can be reworked in parallel, there are restrictions in the ordering of the reduction operations which are performed. In the example above, the addition of the `Partial`_Sum values in the shared Sum variable can be done in any order since the reduction operation is associative and commutative.

Generally, a parallel computation with reduction can be seen as a 3-phase computation.

- In the first phase, the work is split into multiple chunks/workers, and a local accumulator variable is created to be processed independently in each parallel worker.
- Second, for each data in the chunk, an operation is applied that updates the local accumulator (in the example above it is the code inside the loop).
- Third, local accumulators are reduced/combined/aggregated to the "global" accumulator variable (in the example above it is the operation in the Reduce attribute).

The first step can be done all sequentially, before starting the parallel computation, or incrementally (by splitting the data first in a few chunks, starting to process in parallel only a part of the chunks, and incrementally splitting and starting the rest) or using divide-and-conquer (recursively split the chunks in parallel).

The second step is done fully in parallel. If data dependencies exist between the chunks, then some order may need to be imposed on the parallel computations. In this case, this can be done by the programmer explicitly making the synchronization, or using some sort of annotations which guide the compiler (such as the *depend* clause in OpenMP [3]).

The final step, which is one of the main subjects of this paper, can be performed sequentially, when all parallel computations are complete, or, if deemed more efficient, in parallel as soon as each chunk completes. In the case it is done in parallel, two main approaches exist: either each local accumulator is reduced to the global accumulator in parallel

(the global accumulator needs to be protected thus serializing the updates), or sets of non-overlapping local accumulators can be combined among themselves, incrementally building the final value. Noteworthy, the usual approach used for this parallel reduction is by combining pairs of local accumulators.

The order of this parallel reduction is very important, and depends on the properties of the operation being performed.

First, to allow for parallel computation, the operation should be associative (this is not strictly true, as it is possible to re-write the computation to parallelize a non-associative operation), since there is no pre-defined ordering where operations are executed.

For example, for the simple case of a sum:

```
  Sum : Integer := 0;
  Arr: array (1..4) of Integer := (1,2,3,4);
begin
  for I in Arr'Range loop
    Sum := Sum + Arr(I);
  end loop;
```

The sequential order can be defined as ((((0 + 1) + 2) + 3) + 4). If parallelized in two chunks, where each chunk is processed in parallel, and summed in the end, the order is (((0 + 1) + 2) + ((0 + 3) + 4)), which gives the same value, since '+' is associative. Note the value '0' used in both chunks. Each local accumulator needs to be pre-assigned the idempotent value of the operation, usually denoted Identity (e.g., 0 for addition, 1 for multiplication).

Contrarily, in the case of '-', which is non-associative, ((((0 - 1) - 2) - 3) - 4) ≠ (((0 - 1) - 2) - ((0 - 3) - 4)). [3]

The second important property is commutativity, particularly for parallel reductions. An associative, non-commutative operation can be parallelized, but its reduction needs to be performed guaranteeing that the order of operations mimics the sequential order (sometimes defined as the encounter order when considering containers).

For instance, if we are reducing the result of the local sum accumulators, for the example of integer addition, it does not matter the relative order of computation, as (((0 + 1) + 2) + ((0 + 3) + 4)) equals (((0 + 3) + 4) + ((0 + 1) + 2)).

But now consider the case of string concatenation (for simplicity will also be represented with operator '&', identity is the empty string ""). Since it is an associative operation, then (((("" & "a") & "b") & "c") & "d") equals ((("" & "a") & "b") + (("" & "c") & "d")). But it is not commutative, so it is not equal to ((("" & "c") & "d") & (("" & "a") & "b")).

Therefore, the parallelism runtime needs to guarantee that, although there is no defined time order relation between the execution of reductions, the operations need to provide the results in the same order as if had been done sequentially. If not mandatory, the programmer needs to be aware if the runtime does not support non-commutative reduction, in which case the programmer can take responsibility for ensuring the result is deterministic and correct, for example by using ordered containers to contain the parallel result.

[3] ((((0 - 1) - 2) - 3) - 4) is equal to (((0 - 1) - 2) + ((0 - 3) - 4)), so this is parallelizable. But it forces to re-write the operations.

For instance, in the string concatenation case, if "" & "c" is done first, and "" & "a" next (in execution order), reduction cannot be performed to guarantee the same order as if done sequentially. The implementation needs to wait that "b" or "d" is available to reduce into "a" or "c", respectively.

3 Existent approaches

Due to the ever increasing importance of parallelism, support to fine-grained parallelism is provided by many different languages and frameworks. In these, multiple different ways exist on how to support loop parallelization and reduction of parallel computation.

Probably one of the most widely used parallel programming model, OpenMP [3] allows the annotation of `for` loops with a parallel loop construct (using a `#pragma omp for`), which specifies that different iterations of the associated loop can be executed in parallel by the threads which are currently associated to the parallel region where the loop is executed. OpenMP allows the association of the reduction clause with the parallel loop [4], denoting that local copies of each variable should be created, and their values combined at the end of the iteration with a combining operator. This operator can be as specified in the specification (e.g. '+', '*', 'max', ...) or can be user defined combining operators. As noted in the specification, the number of times the combiner is executed, and the order of these executions is unspecified, so operations should be associative and commutative.

Another existent approach is the Cilk Plus language [4], which is an extension to C/C++. In this approach, the C/C++ `for` can be replaced with a specific keyword, `_Cilk_for`, which splits the iterations of the loop into chunks, allowing the runtime to execute each chunk in parallel, by threads of the runtime thread pool. Reduction in Cilk is supported by a special type of objects (Hyperobjects), which are guaranteed to have different views in each parallel computation. The reduce type of hyperobjects allow to specify a callback function which will be used by the runtime to merge pairs of views. The callback function is required to be associative, but not commutative, as the runtime combines in such a way that the same order as serial is maintained.

The current proposal to add parallelism extensions to the C language [5], supports creation of a Reduction type definition. Depending on the context where a variable of the type is used, it may refer to the partial value (if accessed inside a parallel task) or the reduced value (when in the scope it was declared). Reduction is performed by combining pairs of partial results, using the operation of the reduction type. The proposal also provides for an order property of the type, which defines either unordered, uncommuted or serial [5] order requirements in the implementation handling of the reducing.

Within the C++ extensions for parallelism technical specification [6], an equivalent effect to loop parallelization (but for containers) is obtained by algorithms provided in the standard template library. These are similar to algorithms already provided in the C++ standard for accumulating the values of a container with a specific operation. C++ Parallelism TS std::experimental::reduce provides the same functional behavior as C++ std:accumulate, but can be called in any way so operation needs to be associative and commutative (as specified in the technical specification, reduce may be non-deterministic for non-associative or non-commutative operations). A similar library based approach in C++ can be found in the Intel Threading Building Blocks [7], but that provides both a reduce which aggregates pairs

[4] Not only with loops, but in other parallel constructs, but the behavior is the same as in loops.
[5] As pointed out in that proposal, this may imply serialization.

of local results, which allows non-commutative operations, and a simple combine operation which does not required creation of objects, thus more efficient to use when this creation is expensive.

Other examples can be found in C# and Java. Fine-grained parallelism is provided by C# since .Net 4, either using the libraries provided in the TPL (Task Parallel Library) [8], or in alternative by the PLINQ framework [9]. The former provides both explicit parallelism with the ability to explicitly create Tasks (the parallel entity), or implicit by using library methods such as Paralllel.For. With this approach the programmer needs to deal with all aspects of reduction operations and synchronization in the access to the shared reduced variable. Even in the case the programmer uses implicit parallelism with aggregation (e.g. Parallel.ForEach with a partitioner), the programmer needs to provide both the delegate/lambda explicitly coding the synchronization and update of the partial variable, and the delegate/lambda explicitly coding the update of the global variable.

Since this is explicitly handled by the programmer, she can guarantee that the final reduction operation is always folding the partial parallel value into the global accumulator, so reduction operation needs not to be commutative. It needs to be associative, since the partial ordering of the updates is not defined.

PLINQ, the parallel version of LINQ, the declarative data handling syntax extension to C#, is different in that it completely hides from the programmer the parallel execution, and all synchronization in what concerns the reduction operations. PLINQ already provides basic reduction operations (such as sum, average, max, min) as well as more complex operations (union or intersection of sets, filtering, etc.). PLINQ is executed in the TPL, but the programmer loses some flexibility, as she cannot configure explicitly how parallel computations are scheduled in the underlying thread pool, something which is possible in the TPL.

In PLINQ it is possible for the programmer to define custom reduction operations, by providing both the reduction function (delegate or lambda) which updates the local accumulator, as well as the final combine function which is to be applied to combine the local accumulators. This combine operation is done in pairs (two partial accumulators are reduced to a single partial accumulator), which can be done in parallel. The programmer does not need to use locks, since there is no update to a shared global value (synchronization is done internally by PLINQ). This reduction operation needs to be associative, as there is no defined order of execution of operations. By default, for performance, PLINQ does not guarantee order during the parallel processing (although this is configurable) but in the combining phase, the underlying implementation guarantees that the logical (encounter) order of operations is maintained in the end. This implies some extra machinery in the implementation and overhead, but allows to use non-commutative operations.

In Java, previous to Java 8, parallelism in Java was provided by the use of executors and the fork/join framework (FJF) in Java 7 [10]. Both these approaches are based on the programmer explicitly structuring how code can be parallelized and how data is to be partitioned (similarly to the C# TPL basic explicit Task handling). There was no support to parallel patterns (such as Parallel Loops, or Aggregation/Reduction) although these could be built by programmers on top of the available mechanisms.

Using executors or the FJF, reduction is done explicitly by the programmer. In both approaches, the programmer can directly update global shared variables (using locks). Moreover, as the FJF is based in divide-and-conquer approaches (building a tree of parallelism), where the programmer forks the parallel computation, reduction can be done incrementally when joining in any of the tree sublevels, returning the reduced value to its immediately enclosing parallel call (therefore no locking is required). Since in both approaches the full reduction code is done explicitly by the programmer, then the operation can be non-commutative. However, as always order of reductions is not defined, therefore the operations need to be associative.

In the recent Java 8, a new form of parallelism was introduced, using the Java 8 added streams [11], the concept for processing elements in containers in a pipeline way, and using a "declarative"-based style. The parallel version of streams adds syntactic sugar on top of the FJF, although possibly with less flexibility. Using streams, operations in collections can be done in parallel (using a special iterator interface which is able to "split" containers), and reduction and collection operations specified by the user using the reduce and collect methods (the reduction operation always reduces the partial value and a previous result in a new result variable, while the collect updates an existent shared container). Similar to PLINQ, these operations need to be associative but need not to be commutative, as the underlying implementation also guarantee encounter order of the final reduce/collect operation.

4 Exploring different possibilities for loop parallelism

4.1 Type-based proposal for reduction

The proposal which was published in [2] and reviewed in section 1, allows reduction operations to be performed in parallel, but not during the parallel computation itself, as the reduce operation is only specified when the aggregated value is used. If to be done in parallel, it would require a new parallel computation each time (or the first time) the reduced value is used.

Therefore, a different idea is to "attach" the reduce operation to the declaration of the reduced variable:

```
  Sum : Integer := 0
        with Reducer => "+", Identity => 0;

begin
  for I in parallel Arr'Range loop
    Sum := Sum + Arr(I);
  end loop;

  Put_Line("Sum over Arr = " & Integer'Image(Sum)); -- Sum is already reduced
```

Since Sum is annotated with the Reducer aspect, the compiler knows that it will be used in a parallel setting, and that it should create chunk/worker local copies of the variable to be processed in parallel, and later reduced. Note that if not, parallel updates to Sum would be Erroneous Execution, as per the Ada standard [12, section 9.10].

More generically this could eventually be extended to the type declaration:

```
  type Parallel_Integer is new Integer
      with Reducer => "+", Identity => 0;

  Sum : Parallel_Integer := 0;

begin
  for I in parallel Arr'Range loop
    Sum := Sum + Arr(I);
```

```
  end loop;

  Put_Line("Sum over Arr = " & Integer'Image(Sum)); -- Sum is already reduced
```

This allows the implementation to do either parallel reduction during the computation of the loop, or sequentially after the end of the loop.

The reduction could eventually be attached to the array of partial values, instead to the final result:

```
declare
   type Partial_Array_Type is new array (parallel <>) of Float
                with Reducer => "+", Identity => 0;

   Partial_Sum : Partial_Array_Type := (others => 0.0);

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

   Sum := Partial_Sum(<>)'Reduced; -- value is reduced either here or
                                   -- already during the parallel loop
   Put_Line("Sum over Arr = " &  Float'Image (Sum));
```

Note in this last version that there is an explicit notion of the existence of partial values being used (with the box <> notation), being more clear what is happening, while the previous version it is all done implicitly by the implementation. It is important to understand if this explicit notion is required or desirable.

Nevertheless, in these approaches, and similar to what happens in [2] there is an implicit code transformation, where the data is split into chunks, and for each chunk a sequential loop is being used to iterate over the data.

4.2 Partial values handled with a library approach instead of solely syntax based

Another option is to explicitly use library calls (e.g. the hyper-objects of Cilk [4]), to handle the chunk computations, and reduction:

```
  package Parallel_Integer is new Hyperobjects (Result_Type => Integer,
                                                 Reduce => "+", Identity => 0);
  Partial_Sum: Parallel_Integer.Object;

  Sum: Integer;

begin
  for I in parallel Arr'Range loop
```

```
    Partial_Sum.Process(Arr(I));
  end loop;

  Sum := Partial_Sum.Final_Result; -- Either reduced here or already reduced
  Put_Line("Sum over Arr = " & Integer'Image(Sum));
```

An important change with this approach is the handling of situations where the accumulate operation is different of the reduction operation or works with different types from the final reducing. The former is represented in the code below where to calculate `Negative_Accumulated_Value`, it is necessary to process each element of the sequential iteration with one operation (`Accumulate => "-"`), and the final reduction with a different one (`Reduce => "+"`). An example for different types could be the concatenation of characters: accumulate would append each char to an unbounded string and reduce would add two unbounded strings. Interestingly they could even use the same overloaded operator.

```
  package Parallel_Integer is new Hyperobjects (Result_Type => Integer,
                                                Accumulate => "-",
                                                Reduce => "+", Identity => 0);

  Partial_Negative_Accumulated_Value: Parallel_Integer.Object;

  Negative_Accumulated_Value: Integer;

begin
  for I in parallel Arr'Range loop
    Partial_Negative_Accumulated_Value.Process(Arr(I));
  end loop;

  Sum := Partial_Negative_Accumulated_Value.Final_Result;
  -- Either reduced here or already reduced

  Put_Line("Sum over Arr = " & Integer'Image(Negative_Accumulated_Value));
```

In the case of the approach of section 4.1, since the code inside the loop which accumulates the partial value is written by the programmer, there is no need to indicate to the compiler what the accumulator operation is, only the final reducing function. Furthermore, this approach of calling a procedure in each iteration will be more expensive (adding the cost of the call per iteration) compared to the compiler transforming into a number of sequential loops (or this done explicitly as in the last two approaches).

In any case, there would need to be a separate interface to be used only by the implementation to “configure” the parallel computation, such as the number of chunks (either static or dynamic), the creation of one local accumulator object per worker, and to use the appropriate chunk index in the use of the hyper object within the loop. This means that there is still some implicit compiler transformation. To clarify what is happening, the special (<>) notation could be used to denote that there might be several partial sums being updated:

```
  for I in parallel Arr'Range loop
    Partial(<>).Process(Arr(I));
  end loop;
```

For the case of hyperobjects, a potential interface for the library could be [6]:

```
generic
     type Result_Type is private;
     Identity : Result_Type;
     with function Accumulate (Left, Right : Result_Type) return Result_Type;
     with function Reduce (Left, Right : Result_Type) return Result_Type;
package Hyperobjects is

     type Hyperobject is limited private;

     procedure Allocate_Local_Object(H : in out Hyperobject;
                                     Chunk : Chunk_Id);
     -- called to initialize a new chunk object

     procedure Process(Chunk: Chunk_Id, Value: Result_Type);
     -- this is the local iteration which calls Accumulate

     procedure Chunk_Completed(Chunk: Chunk_Id);
     -- compiler notifies chunk is completed.
     -- Hyperobject may reduce the value of this chunk using Reduce


     function Final_Result (Obj : Hyperobject) return Result_Type;
     --  Reduction over remaining (initialized) elements of hyperobject
     --  and final result is returned.

end Hyperobjects;
```

4.3 Potentially supporting non-commutative operations

When reduction is done implicitly by the compiler and runtime, and not under the control of the programmer, it is important to understand if the implementation guarantees encounter order, which allows for non-commutative operations, or if commutativity of reducing operation is required for efficiency reasons (one possibility would be to make this implementation-defined).

[6] A more complete interface could also exist where different types were used for the result and for the value being accumulated. For instance, concatenation of Characters into a String.

It could eventually be possible to add an annotation to the reduction operation if it is commutative, allowing the implementation to use a more efficient approach to perform the reductions, or the compiler rejecting if the implementation did not provide the functionality:

```
  type Parallel_Integer is new Integer
      with Commutative_Reducer => "+", Identity => 0;

  type Parallel_String is new String
      with Non_Commutative_Reducer => "&", Identity => "";
```

4.4 Parallel iterators

In parallel with the approach above, iterator interfaces can be provided that allow a container to be processed in parallel. Different iterator types can be provided, depending if the container size is known, and if known, if it allows access to intermediate elements.

If size is not known, only `First` and `Next` can be provided, although it would be allowed that Next is executed in a different worker, if one available.

```
type Unbounded_Parallel_Iterator is
      limited interface and Forward_Iterator;
```

Other iterator types can be made available when the size of the container is known. In this case, a generic package can be used:

```
generic
    type Iteration_Count_Type is mod <>;
package Ada.Iterator_Interfaces.Bounded_Parallel_Interfaces is

    type Bounded_Parallel_Iterator is limited interface and Forward_Iterator;

    function Iteration_Count
      (Object : Bounded_Parallel_Iterator)
       return Iteration_Count_Type is abstract;

    type Indexed_Parallel_Iterator is limited interface
      and Bounded_Parallel_Iterator;

    function Nth
      (Object : Indexed_Parallel_Iterator;
       Index  : Iteration_Count_Type) return Cursor is abstract;

end Ada.Iterator_Interfaces.Bounded_Parallel_Interfaces;
```

Both iterator types have a `Iteration_Count` function which allows the compiler to query and decide on the number of chunks. Obviously, access to later elements needs to be done using `Next`, so the container would need to be iterated to actually split parallel execution.

The `Indexed_Parallel_Iterator` would also have a function `Nth`, which would allow to immediately split the container processing.

The use of these iterators would permit the use of loops with "of" on containers (compiler can check that the specific Container implements at least one of the parallel iterator types):

```
for Obj of parallel Container loop
  Process(Obj);
end loop;
```

Other types of iterators could be provided, e.g. a divide-and-conquer iterator, which could be used to recursively split the container (similar to Java's `Spliterator` in Java 8 [13]).

Note that this is orthogonal to the use of syntax versus library calls, and both can be combined.

4.5 Explicit library calls

All these approaches rely on the implementation to perform implicit transformations (transforming the parallel loop into a number of non-overlapping sequential loops) . A different library approach could be provided where equivalent library calls were performed to mimic behaviour of parallel loops (e.g. similar to the approach in C# [8]).

```
Sum : Integer;
Parallel_Loop (From       => Arr'First,
               To         => Arr'Last,
               Initialize => 0,
               Body       => Sequential_Chunk_Code'Access,
               Reduce     => Final_Reduction'Access);
```

`Sequential_Chunk_Code` and `Final_Reduction` could be procedures specified by the programmer, or, if a "lambda" equivalent is also provided in the language, this could become:

```
Partial_Sum : Integer with Executor_Local_Storage;
Parallel_Loop (From => Arr'First,
               To   => Arr'Last,
               Initialize => is begin Partial_Sum := 0; end,
               Body => (Start, Finish : Iterator_Type) is
                  begin
                     for I in Start .. Finish loop
                        Partial_Sum := Partial_Sum + Arr(I)
                  end loop;
```

```
end,
Reduce => is begin Sum := Sum + Partial_Sum; end);
```

4.6 Programmer coded explicit chunking and reduction

Another option is to go further and not allow the compiler and runtime to handle the split and reduce, and put it all in the hands of the programmer, which would allow the source code to actually denote the intended chunking and reducing. This could be important in more critical scenarios.

In this case, the sum would be:

```
  Sum : Integer := 0;

  Number_Chunks : Integer := …; -- e.g. number of cores

  Partial_Sum: array (1 .. Number_Chunks) of Integer;

begin
  for I in parallel 1 .. Number_Chunks loop
      declare
            -- this can be thread local storage
            Chunk_First: Integer := …;
            Chunk_Last : Integer := …;
            Local_Sum  : Integer := 0;
      begin
            for Local_I in forward First .. Next loop -- sequential loop
                  Local_Sum := Local_Sum + Arr(Local_I)
            end loop;

            Partial_Sum(I) := Local_Sum;
      end; -- end of declare block

  end loop; -- end of parallel computation

  for I in forward 1 .. Number_Chunks loop -- sequential
      Sum := Sum + Partial_Sum(I);
  end loop;
```

Note that this approach is always available, whatever the more elegant syntactic sugar approaches which the language also would support.

5 Summary

This paper discussed different possibilities to support parallel loop iterations in the current proposal for fine-grained parallelism in Ada, as well as how to define reduction of the parallel values. The paper presented both syntax and library based approaches, to reduce the complexity of the program code, as well as programmer explicit handling of parallelism and reduction.

One open issue is also if reduction should only be applicable to for loops, or more widely to any parallel computation (e.g. parallel blocks). This exists in other approaches and allows to programmer to build more irregular parallel algorithms and still be able to perform reduction.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments. This work was partially supported by Portuguese Funds through FCT/MEC (Portuguese Foundation for Science and Technology) and co-financed by ERDF (European Regional Development Fund) under the PT2020 Partnership, within the CISTER Research Unit (CEC/04234); also by the European Union under the Seventh Framework Programme (FP7/2007-2013), grant agreement n° 611016 (P-SOCRATES).

References

[1] L. M. Pinho, B. Moore, S. Michell, S. T. Taft, “An Execution Model for Fine-Grained Parallelism in Ada”, Proceedings of the 20th Ada-Europe International Conference on Reliable Software Technologies, Madrid Spain, June 22-26, 2015, http://dx.doi.org/10.1007/978-3-319-19584-1_13.

[2] S. T. Taft, B. Moore, L. M. Pinho, S. Michell, “Safe Parallel Programming in Ada with Language Extensions”, Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology (HILT '14). ACM, New York, NY, USA, http://dx.doi.org/10.1145/2663171.2663181.

[3] OpenMP Architecture Review Board, “OpenMP Application Program Interface”, Version 4.5, November 2015, available at http://www.openmp.org/mp-documents/openmp-4.5.pdf, last accessed January 2016.

[4] Intel® Cilk™ Plus Language Extension Specification, Version 1.2, available at https://www.cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_1.2.htm, last accessed January 2016.

[5] Programming languages — C — Extensions for parallel programming, N1966 (2015-09-14), available at http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1966.pdf, last accessed January 2016.

[6] Technical Specification for C++ Extensions for Parallelism, N4507 (2015-05-05), available at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4507.pdf, last accessed January 2016.

[7] Intel® Threading Building Blocks (Intel®TBB) Developer Guide, Intel® TBB 4.4, available at https://software.intel.com/en-us/tbb-user-guide, last accessed January 2016.

[8] Microsoft .NET Framework Development Guide, Task Parallel Library (TPL), available at https://msdn.microsoft.com/en-us/library/dd460717.aspx, last accessed January 2016.

[9] Microsoft .NET Framework Development Guide, Parallel LINQ (PLINQ), available at https://msdn.microsoft.com/en-us/library/dd460688.aspx, last accessed January 2016.

[10] Java Fork/Join, Java Tutorials, available at https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html, last accessed January 2016.

[11] java.util.stream, Java Platform, Standard Edition 8 API Specification, available at https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html, last accessed January 2016.
[12] ISO IEC 8652:2012. Programming Languages and their Environments – Programming Language Ada. International Standards Organization, Geneva, Switzerland, 2012.
[13] java.util.Spliterators, Java Platform, Standard Edition 8 API Specification, available at https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html, last accessed January 2016.