CISTER
Research Center in
Real-Time & Embedded
Computing Systems

Technical Report

Tasklettes – a Fine Grained Parallelism for Ada on Multicores

Stephen Michell

Brad Moore

Luis Miguel Pinho*

*CISTER Research Center

CISTER-TR-130304

2013/06/10

Tasklettes – a Fine Grained Parallelism for Ada on Multicores

Stephen Michell, Brad Moore, Luis Miguel Pinho*

*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

The widespread use of multi-CPU computers is challenging programming languages, which need to adapt to be able to express potential parallelism at the language level. In this paper we propose a new model for fine grained parallelism in Ada, putting forward a syntax based on aspects, and the corresponding semantics to integrate this model with the existing Ada tasking capabilities. We also propose a standard interface and show how it can be extended by the user or library writers to implement their own parallelization strategies.

Tasklettes – a Fine Grained Parallelism for Ada on Multicores

Stephen Michell[1] Brad Moore[2] Luís Miguel Pinho[3]

[1] Maurya Software Inc, Canada, stephen.michell@maurya.on.ca
[2] General Dynamics, Canada, brad.moore@gdcanada.com
[3] CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto, Portugal, lmp@isep.ipp.pt

Abstract. The widespread use of multi-CPU computers is challenging programming languages, which need to adapt to be able to express potential parallelism at the language level. In this paper we propose a new model for fine grained parallelism in Ada, putting forward a syntax based on aspects, and the corresponding semantics to integrate this model with the existing Ada tasking capabilities. We also propose a standard interface and show how it can be extended by the user or library writers to implement their own parallelization strategies.

1 Introduction

The development of ubiquitous multi-CPU computers has led to a more pressing need to be able to express parallel computational algorithms effectively in general purpose programming languages.

The development of programs that capture concurrent properties of algorithms has been a focus of many research papers (since [1, 2]) and has been implemented in operating systems, such as DEC RT-11, HP, Unix, POSIX and Microsoft Windows, and programming languages concurrent Pascal [1], Ada [3] and Java [4]. Most approaches focused either on concurrency in the small (a few of threads or interrupts) or specialized processor domains, such as SIMD (Single Instruction Multiple Data) environments. Recently, other languages, such as C# and C++ have added or are investigating methods of implementing fine grained concurrency, which is the subject of this paper. We are ignoring much of the work for SIMD machines for now as they use specialized toolsets and techniques.

Unfortunately, the perception of the general threading or tasking environment is that the threads or tasks are too expensive in resource usage, cumbersome (in terms of being easy to use by average programmers), not easily mapped to the physical resources at hand at the time of program execution, and divergent from the problem space when attempting to apply concurrency to computationally intensive activities [5, 6].

Arguably, the Ada tasking model with its first-class task types and well defined syntax/semantics for inter-task interactions [7] simplifies the expression of many concurrent properties and solutions. However, other models, such as loop-level paral-

lelism or parallel subprogram execution are not easily expressed in the current Ada model. In addition, the resource consumption issues and cost of dynamic task creation and destruction of Ada tasks when used in undisciplined ways still begs for a better approach to map the concurrency power available at the hardware level to algorithms written at the below-subprogram level.

Parallel programming in Ada was considered several years ago ([8,9,10]). Mayer and Jahnichen [8] introduce a `parallel` keyword, which applies to for loops, allowing a specific compiler to optimize loop iterations, targeted to a multiprocessor platform. Hind and Schonberg [9] also targeted the optimization of parallel loops, introducing the concept of lightweight (mini) tasks, to reduce the overhead of using tasks for parallelism. Thornley [10] proposes two extension keywords to standard Ada: `parallel` and `single`, where `parallel` is used for declaring that a block or a `for loop` will be executed in parallel.

More recent proposals have been made to extend Ada's capabilities by using generics [11], pragmas [11], and language constructs [12]. This work builds upon these to present a more unified proposal.

In this paper we address these issues as they relate to the Ada programming language; propose syntax for Ada that more closely matches the need for fine grain concurrency than exists at present; and propose semantics for the syntax presented that seamlessly integrates the existing Ada tasking capabilities and the new fine grain concurrency.

2 Problem Analysis

Concurrency as a discipline has been the subject of intense research from the days of Per Brinch Hansen [13]. The most common usage was to handle external events, to manage the progression of work and to ensure that work was scheduled according to the importance (priority) of the work. For the majority of systems there was a single CPU that was the resource to be scheduled, and for the rest there were a few CPUs that were shared between many more tasks.

As long as CPU speed was increasing exponentially, the pressure to increase throughput by increasing CPU count was overwhelmed by that speed increase. When maximum CPU speed became capped in the mid-2000's, the pressure to increase performance by adding cores became overwhelming. We now stand on the threshold of “too many cores”, where chip manufacturers prepare to deliver hundreds or thousands of cores, each with tens or hundreds of “lanes” for parallel work.

With these changes, there will be many more cores than tasks ready to execute at any one time. These cores are available to subdivide heavy calculations when the algorithms can be effectively parallelized.

There is an apparent belief that we can create lightweight threads and that a program can detect how many cores are available at any one time and allocate the lightweight threads to cores to execute a parallel algorithm. This belief ignores the fact that the operating system schedules all resources, memory, threads, and cores. A program cannot schedule cores without scheduling the threads that could be using them.

An approach institutionalized by MIT [14, 15] and now commercialized by Intel [16] and being used by ParaSail [17], Intel's Threading Building Blocks [18], Java Fork/Join [19], OpenMP [20], Microsoft's Task Parallel Library [21], is to put light-weight "tasks" on top of thread pools. This is a promising approach that we investigate further to implement the model that we develop here.

Another issue that must be addressed for such distribution is the nature of the algorithm being distributed. Any algorithm that is a candidate for parallel execution must calculate a deterministic result independently of the order in which the fragments are combined. This is simple if there is only a single operation (such as "+" or "*") and the operation is commutative, but may not be trivial for non-commutative operations such as "-" or for more complex combinations of such functions or operations.

Often, the algorithm must be rewritten to add partial temporary accumulator variables and to combine these temporary variables correctly to produce the correct result. In some cases, the compiler may be able to perform such rewrites, but it is ultimately the programmers' responsibility to be aware of such issues and to ensure that when parallelism is applied to a programming construct, the algorithm as written will not be incorrect when executed in parallel.

3 Semantic Model

In order to effectively describe the new concurrent behavior, we introduce a unit of concurrency called a "tasklette". Unlike tasks, tasklettes are not nameable or directly visible in a program. A tasklette carries the execution of a subprogram or of a code fragment in parallel with other tasklettes executing the same code fragment (with different variables) and possibly in parallel with other tasklettes executing code fragments from other Ada tasks.

Tasklettes come in two types. The first type is invisible to the programmer and is created by the compiler when it can determine that an operation can be parallelized and submitted to multiple CPUs. Example of such usage could be the default initialization or assignment of values to arrays of records or the copy of a large structure using the Ada assignment operator.

The second tasklette type is the subject of this paper and requires the programmer to use explicit syntax to guide the compiler and runtime. This syntax will include the use of aspects on subprograms or on loops. This syntax will be specified in the next section, followed by examples.

A major impetus behind making tasklettes not declarable is to separate the programmer from the implementation of the parallel constructs [1]. Programmers will declare an intent that code fragments be executable in parallel, but need not concern themselves with the details of the parallelism itself. Tasklettes are meant to augment, not replace tasks as the visible unit of concurrency.

---

[1] This is the opposite of tasks, where the decision was to make the parallel computation obvious, since tasks are used to express concurrent activities while tasklettes are used to map the application to the underlying platform.

Tasklettes behave as if each one is executed by a single Ada task that is explicitly created for the execution of the tasklettes and terminated immediately after execution of the code fragment. Instead of attempting to map tasklettes to cores, we map them to tasks and use the Ada tasking model to express the concurrency since tasks in Ada already have a computationally sound model that addresses priority and scheduling on multicore platforms. To not base this concurrency on tasks could mean extreme difficulty in using tasks and tasklettes in the same partition.

Any such tasks that execute the tasklettes are usually hidden from the programmer, and the only interface that the compiler exposes (even if we create our own task pool) is a set of packages and generics to let the pool provide the service. This interface is specified in section 5.

At the present time we are working to extend the model to include real-time behaviors, task priorities, and the Ravenscar tasking model. To date we have found no fundamental limitations that would prevent this extension.

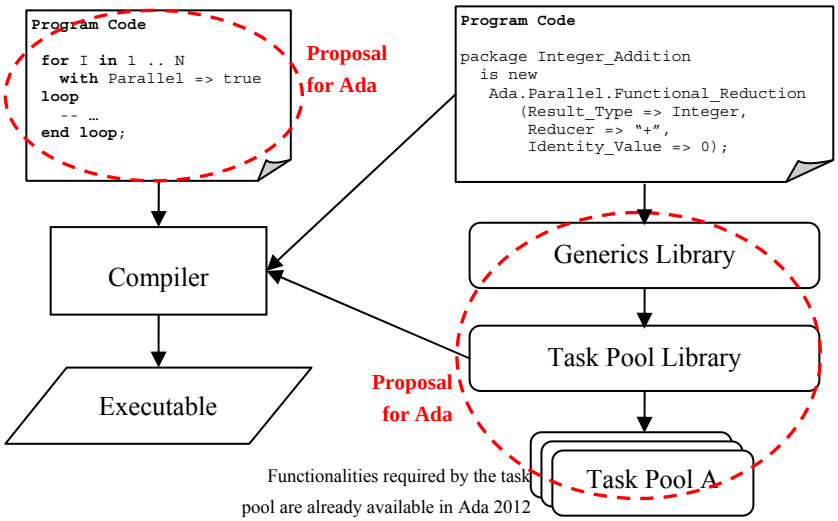

Figure 1 – the proposed model

We propose a runtime model where the execution of code fragments executed by multiple tasklettes is restricted to producing the same result as would happen if executed sequentially. This has some obvious constraints on the user code:

— Parallel code fragments must not update any non-atomic variable read by another code fragment that is executed in parallel, without making special language measures to protect the variable [2].

— Program execution must not proceed beyond the parallelized code until tasklettes executing that fragment have completed and delivered their results.

[2] If the underlying implementation does not use tasks, then protected operations and potentially blocking operations cannot be called directly from user code. As the proposed model is based on mapping tasklettes to tasks this restriction is not needed.

Our proposal includes an Ada interface to implement the semantic model provided in this paper. Compilers and runtimes would be free to provide an implementation that does not use the interface as long as the execution of the tasklettes has the same semantic effect given here. However, if a task pool is provided for the default, then it must be used as specified here. Figure 1 provides an overview of the proposed model.

4 Syntax

In this section we present the addition of parallelization abilities to subprograms and loops [3]. The most relevant addition to the language is the introduction of the `Parallel` aspect, which, when applied to a specific construct instructs the implementation that parallelization should be provided.

It is important to note that the programmer must also be able to specify the underlying behavior of the runtime, both controlling the scheduling of tasklettes and the parameters of the task pool. This is achieved through the programmer being able to interact with the underlying library as will be detailed in Section 5.

4.1 Parallel Subprograms

Subprograms are a natural candidate for parallelization, in particular for the case of pure functions, which are not intended to present side effects. Nevertheless, even subprograms that operate on shared data can be parallelized as it may be possible for the programmer to control contention, to verify that contention is controlled, or to verify that parallel access is on non-overlapping areas of data [4].

Two possibilities are provided for the placement of the aspect specification. One is to place `with Parallel` in the specification of the subprogram, and the second is for the syntax to be placed in the actual call.

The first method (on the subprogram specification) can be used to create default behavior and make the parallel nature visible to callers. The second approach supports legacy libraries, and allows the programmer to have a fine-grained control of the parallel behavior (at the time of call, e.g. due to different execution conditions).

In its simple form, if the programmer accepts the default behavior of the underlying task pool and tasklette scheduling, she just needs to include the Parallel aspect in the call to the subprogram:

```
Ret := Call (Parameters)
   with Parallel => True;
```

[3] We also considered the parallelization of blocks, but after analyzing we found that the syntax required to make them effective would be similar to declaration of "anonymous" subprograms, e.g. with in and out parameters, so we decided to propose that programmers specify parallel subprograms in these cases.

[4] Access even on non-overlapping areas of data may cause contention as a write into a variable may cause a cache line to be invalidated, thus impacting on variables in the same line.

Note that, in accordance to the rules for aspects [22], the `=> true` can be omitted. In case **`with`** `Parallel` is not included the value of the aspect is false, so the subprogram cannot be executed asynchronously.

In order for the return of the parallel call to be "safe", and so that no waiting needs to be implemented explicitly by the programmer, the asynchronous call waits either on (what comes first):

- Access to the variable holding the call result, or
- The end of the enclosing scope of the call.

This restriction implements fully-strict parallelism [23], and guarantees that the asynchronous subprogram has access to the stack frame of the enclosing scope of the call in its execution, a similar approach as determined for Cilk, and which has been proposed for C++ [24]. An example for instance is a parallel solution for the Fibonacci series, which could be written as [5]:

```
function Fibonacci(N: Natural) return Natural is
  function Sequential_Fibonacci (N: Natural)
      return Natural is
    ... -- Some implementation of iterative fibonacci
  end  Sequential_Fibonacci;
begin
  if N < Cut then -- to stop parallelism (efficiency)
    return Sequential_Fibonacci(N);
  else
    return
      Fibonacci (N-1) with Parallel => True
      + Fibonacci (N-2) with Parallel => False;
  end if;
end Fibonacci;
```

We also considered that a way to control parallel actions is necessary, including (i) "select"ing on multiple alternative parallel actions, continuing after getting the first result, and (ii) directly requesting the abort of all tasklettes still pending in the current scope (for scope exit). We have identified this as future work.

4.2 Parallel loops

Parallelizing loops is one of the best known examples of the advantages of parallel execution. A simple loop that can be parallelized is:

```
for I in 1 .. N
  with Parallel => True
loop --... end loop;
```

[5] To increase efficiency this solution parallelizes one of the branches, since the existing task can do the N-2 branch, and stops parallelization when it is more efficient to go sequentially.

In the simple case the compiler could create *N* tasklette objects, one per iteration, which would be placed in the queue of the default task pool. This approach is only appropriate if each iteration is computationally intensive (e.g. ray tracing). In most cases, the advantages of parallelization are only obtained if some partitioning is used.

This partitioning could be performed by the compiler, dividing the range in P "chunks" (even with variable or varying dimensions), with P < N, e.g. based on the number of available cores.

Nevertheless, in most cases loop iterations are not independent, and some sort of reduction may be required. As an example, if the loop was calculating a sum, then each chunk would produce a partial value which would need to be reduced to one result. Generalizing, for some code:

```
X : User_Type := Some_Default;
for I in 1 .. N loop
  X := Func(X, I);
end loop;
```

the programmer could express the loop in terms of a formal parallel model, being able to identify to the compiler the reduction operation, the variable which will be partially calculated in each chunk of iterations and then accumulated (or reduced), and the identity value, which is used to initialize each partial result:

```
 function G (Accumulator, Iteration_Result: User_Type)
   return User_Type is
    ... -- some function
begin
 X : User_Type := Some_Default;
 for I in 1 .. N
   with   Parallel => True, Reduction => G,
          Accumulator => X,
          Identity => Some_Identity_Value
 loop
    X := Func(X, I);
 end loop;
end;
```

Nevertheless, for the general case, the compiler may not be able to parallelize a sequential loop without the help of the programmer (e.g. even in simple aggregation loops, if the operation is not associative). One approach would be for the programmer to rewrite the loop in terms of the associative operation. For instance:

```
for I in 1 .. N loop
  Sub := Sub - Buffer(I);
end loop;
```

could be rewritten in order to have an associative operation being performed (`Sub := Sub + (-Buffer(I))`[6]. This would nevertheless require the programmer to change the code. Instead, our proposal allows for the programmer to directly identify reduction and identity:

```
for I in 1 .. N
  with  Parallel => True, Reduction => "+",
        Accumulator => Sub, Identity => 0
loop
  Sub := Sub - Buffer(I);
end loop;
```

4.3 Explicit control of partitioning

For the cases that the partitioning is not easy to understand or the reduction operation is not as simple to identify (or the programmer prefers to explicitly handle it in the loop), we also allow for the explicit control of partitioning.

In its simple form, the programmer may partition the loop into "chunks", using a sequential iteration in each chunk. For example (for simplicity assuming that N is divisible by `Chunk_Size`):

```
for I in 1 .. N/Chunk_Size with Parallel => True
loop
  for J in I*Chunk_Size .. (I+1)*Chunk_Size-1
                with Parallel => False loop
   --...
  end loop;
end loop;
```

The `Parallel => False` in the inner loop is not necessary, but can be given for higher clarity. `Chunk_Size` can be a constant, a user variable (e.g. the user queries the number of cores in the platform) or even a function supplied by the task pool (e.g. how many tasks in the task pool are available).

The chunk policy can also be provided as an aspect of the loop. This allows more advanced partitioning approaches, with variable chunk sizes depending on the system load, dynamically managed by the underlying runtime. In this case, the start and finish of the chunk is obtained using attributes on the appropriate loop control variable:

```
for I in 1 .. N
       with Parallel => True,
            Chunk_Size => [N_Core | auto | dynamic]
loop
  for J in I'Chunk_First .. I'Chunk_Last loop
```

[6] Compilers may eventually be able to perform many automatic parallelizations in these simple examples being shown. Nevertheless the model is for the general case.

```
    -- Other attributes could give size and range
    -- No aspect in the inner loop so it is sequential
  end loop;
end loop;
```

Dependencies and reduction can be supported by declaring local variables inside the loop [7], and aggregating the result in a global variable (would need to be protected).

```
Sub := ...;
for I in 1 .. N loop
  Sub := Sub - Buffer(I);
end loop;
```

could become:

```
Sub := ...;
for I in 1 .. N with Parallel => True,
                     Accumulator => Sub
loop
  declare
    Local_Sub : Integer := 0;
  begin
    for J in I'Chunk_First .. I'Chunk_Last loop
      Local_Sub := Local_Sub - Buffer(I);
    end loop;
    Sub := Sub + Local_Sub;
  end;
end loop;
```

The `Sub` variable can now be updated in parallel. Therefore, the `Accumulator` aspect in the loop also signals the compiler that `Sub` must be protected.

A more complex case is when the code presents a loop-carried dependency, where subsequent iteration of a loop requires the computed value of the previous iteration:

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

This code cannot be automatically converted into a parallel loop. It can nevertheless be parallelized using a prefix-sum algorithm [25], since the operation is associative.

The proposed approach also considers another level of abstraction, where the programmer is able to specify and control the underlying scheduling, manipulating more directly the operations being performed by the runtime. To support this control, an

[7] This could also allow the compiler to optimize the placement of variables in Non Uniform Memory Architectures (NUMA), as these variables will only be used in one core.

aspect `Parallel_Manager` is used, to specify the object that controls parallelism. This is presented in the next section.

5 Interface to the Runtime

The goal of the parallel runtime is to define an interface that provides flexibility to the application programmer, yet minimizes the implementation burden for the compiler writer. The desire is also to provide an interface that could be standardized so that a parallel library writer could plug in different parallelism strategies and allow the application programmer to have fine-grained control over the parallelism. The runtime consists of the task pool interface and the parallelism generics.

5.1 Task Pool Interface

We define an interface (partly shown below) to a task pool facility that provides the abstraction of managing a set of tasks as general purpose workers where a worker can be dispatched to a tasklette. The parallelism strategy implemented by a library writer interacts with the task pool. The model is that the parallelism manager offers work (procedure `Offer_Work`) to the task pool in the form of a work plan object defined by the library writer. The task pool releases a worker which calls the `Engage` method of the plan to perform the work. The `Engage` call is essentially the tasklette code that is executed, which ultimately calls out to execute the users' parallel algorithm.

```
package Ada.Parallel.Task_Pools is
   -- A Work Plan defines the work strategy
   type Work_Plan is limited interface;

   procedure Engage (Plan : Work_Plan) is abstract;
   -- When a worker starts executing, it engages the
   -- work plan. This call represents the tasklette
   -- code. Engage executes the plan. Upon returning,
   -- the Worker returns to the task pool

   type Task_Pool_Interface is limited interface;

   procedure Offer_Work
     (Pool : in out Task_Pool_Interface;
      Item : aliased in out Work_Plan'Class;
      Worker_Count : Positive_Worker_Count)
   is abstract;
   -- Allows a work plan to request workers from the
   -- task pool. The Work plan is offered to the task
   -- pool, which is then engaged by each worker
end Ada.Parallel.Task_Pools;
```

The task pool interface shown above can be implemented by extending the interface by any number of implementations that could be provided by library writers. Some possible candidates are unbounded task pools, where the number of workers can dynamically increase to accommodate the load, bounded task pools where the number of worker tasks is statically defined, and Ravenscar tasks pools that are compatible with the Ravenscar profile tasking restrictions.

5.2 Parallelism Control

Having presented the task pool interface, we now consider the parallelism generics that interact with the task pool. We have identified a need for two forms of parallelism in an application; non-recursive parallel subprograms, and divide and conquer parallelism, which covers both parallel loops and recursive subprograms.

5.2.1 Non-Recursive subprograms

The non-recursive subprogram case is perhaps the simplest, since there is only one call involved and thus there is no need for a parallelism strategy such as work-sharing, work-seeking, or work-stealing [8], nor is there a need for reduction. In addition, there are no specific restrictions on the parameter profile of the subprogram, and the compiler writer can implement the calls without the need for library support [9]. For example, the compiler can create a wrapper for each non-recursive subprogram to be called in parallel. The wrapper declares a tasklette which obtains a worker task from a task pool, and then invokes the real call from the context of the worker task. Since the tasklette is declared within the stack frame of the wrapper, it can issue the call to the real subprogram simply passing the parameters passed to the wrapper straight through to the real subprogram. The parallel non-recursive call is simple enough that it warrants no further discussion.

5.2.2 Divide and Conquer Parallelism

The other types of parallelism in our proposal are parallel loops and parallel recursion. These apply parallelism by utilizing a divide and conquer strategy. There are several possible sub-strategies. For instance, a load balancing sub-strategy might be utilized if the effort to process items in the loop varies through the iteration or if the recursion is unbalanced. Work-sharing might be chosen if the work can be divided more evenly. Regardless of the sub-strategy chosen, reduction may be needed if the parallelism produces a result. For these forms of parallelism, a library approach is proposed. Such a library implemented in Ada would involve generics, since the data types, loop iteration index types, and result types are user-defined and may range from simple elementary types such as Integer and Float, to complex user-defined record structures and tagged types.

---

[8] More details on these strategies can be found in [26].

[9] If a library for this strategy is provided, the compiler will implement the calls to this library.

The runtime library model for divide and conquer parallelism consists of a hierarchy of packages that can be selected for specific purposes. These libraries provide the parallelism, while the compiler performs a transformation from the syntax features described above.

The run time interface consists of a three stage generic instantiation (shown later):

- Stage 1 => Reduction primitives
- Stage 2 => Work type + Strategy Interface
- Stage 3 => Parallelism Strategy

The first instantiation allows the application programmer to specify the result type, the reducing function, and the identity value for the result type.

The second instantiation defines the data type describing the work to be processed in parallel, and defines the interface to be implemented by library writers for the third level instantiation. There are two possibilities here. If the parallelism is a parallel loop, then the work type is the iteration index type. This may be any user-defined scalar type. If the parallelism is for a recursive subprogram, then the work type is the data type that represents the work to be performed.

The third instantiation defines the parallelism strategy and how the work is to be performed in parallel. While the first two stages establish the parallelism framework and are proposed for standardization, the third level libraries implement the level 2 interface and may be provided by third-party library writers.

Level one Instantiation Interface

```
generic
  type Result_Type is private;
  with function Reducer (Left, Right : Result_Type)
             return Result_Type;
  Identity_Value : Result_Type;

package Ada.Parallel.Functional_Reduction is
end Ada.Parallel.Functional_Reduction;
```

As can be seen, the first instantiation is trivial, defines no operations, and requires no body [10]. This instantiation is only used if a result is to be generated, and allows a programmer to specify the reducing operation needed for the parallelism opportunity.

Level two Generic Interface for Parallel Loops

```
generic
   type Iteration_Index_Type is (<>);
package Ada.Parallel.Functional_Reduction.Loops

    type Parallelism_Manager is limited interface;
```

[10] More complex situations, such as where the reducers are of different type from the result value, can be handled by other generics, with more parameters.

```
      procedure Execute_Parallel_Loop
         (Manager : Parallelism_Manager;
          From    : Iteration_Index_Type
              := Iteration_Index_Type'First;
          To      : Iteration_Index_Type
              := Iteration_Index_Type'Last;
          Process : not null access procedure
              (Start, Finish : Iteration_Index_Type;
               Item : in out Result_Type);
          Result : in out Result_Type)
      is abstract;

   end Ada.Parallel.Functional_Reduction.Loops;
```

The level two instantiation for parallel loops is a child package of the `Functional_Reduction` package of level 1. This package allows the programmer to specify the data type associated with the loop index as the `Iteration_Index_Type`. The manager type defines the interface to be implemented by the library writer for the level 3 instantiation. The library writer must also provide a constructor function called `Create` (with defaulted parameters) that returns a `Manager` object [11].

Level two Generic Interface for Recursive Subprograms

```
   generic
      type Work_Type is private;
      --  Data type to be processed recursively

   package Parallel.Functional_Reduction.Recursion is

      type Parallelism_Manager is limited interface;

      function Execute_Parallel_Subprogram
        (Manager : in out Parallelism_Manager;
         Item : Work_Type;
         Worker_Count : Worker_Count_Type :=
                              Default_Worker_Count;
         --  Top level item to process recursively
         Process : not null access function (
             Item : Work_Type) return Result_Type)
         return Result_Type
      is abstract;
   end Parallel.Functional_Reduction.Recursion;
```

[11] This function is to be used by the compiler to create a manager object for each parallelism opportunity. The function must provide the parameters to match the aspects specified at the parallelism opportunity.

Similarly, the level 2 instantiation for recursive subprograms defines the interface that the library writer needs to implement and is proposed for standardization. The `Execute_Parallel_Subprogram` is invoked by a wrapper function generated by the compiler, which manages the parallelism opportunity. As before, a `Create` constructor function returns a manager object, which is called by compiler generated code to initialize an object declared in the declaration section of the wrapper function for the programmers' code.

5.3 Example: Parallel Loops

To demonstrate the 3 stage process, consider the earlier example to calculate the sum of integers from 1 to N.

The first stage generic instantiation sets up the reduction needed for Integer addition.

```
with Ada.Parallel.Functional_Reduction;
package Integer_Addition is new
   Ada.Parallel.Functional_Reduction
      (Result_Type => Integer,
       Reducer => "+",
       Identity_Value => 0);
```

For the second phase instantiation, we need to decide if the parallelism applies to a loop or to a recursive subprogram. In this case, we are interested in a loop. The package instantiation from the first phase is used to create the parallel loop generic.

```
with Integer_Addition;
with Ada.Parallel.Functional_Reduction.Loops;
package Integer_Addition_Loops is new
     Integer_Addition.Loops
        (Iteration_Index_Type => Integer);
```

For the third and final phase instantiation, we need to specify the parallelism strategy. For this phase, we can instantiate a generic library provided by a library writer, which may be a third party developer, a library provided by the compiler vendor, or a library written by the application programmer.

Assuming that a work sharing library is of interest for this loop, one might instantiate the third phase at library level to look something like:

```
with Integer_Addition_Loops;
with Ada.Parallel.Functional_Reduction.Loops.
                                          Work_Sharing;
package Work_Sharing_Integer_Addition_Loops is new
   Integer_Addition_Loops.Work_Sharing;
```

Now that the parallelism package has been fully instantiated, it can be used in an application program, to generate the parallelism.

```
with Work_Sharing_Integer_Addition_Loops;
with Ada.Parallel.Task_Pools.Bounded;
use Ada.Parallel;
package WSIA renames
                Work_Sharing_Integer_Addition_Loops;
-- ...
  declare
     Sum : Integer := 0;
  begin
    for I in 1 .. N
      with Parallel => True, Worker_Count => 4,
           Task_Pool => Task_Pools.Bounded.Default_Pool,
           Parallel_Manager => WSIA.Parallelism_Manager,
           Accumulator => Sum
    loop
        Sum := Sum + I;
    end loop;
    -- ...
```

Another example is provided in Appendix, illustrating how one might instantiate the parallelism generics to solve the parallel fibonacci problem recursively in parallel.

6 Conclusion and Future Work

We have shown a powerful model that permits fine grained concurrency to be added to Ada and is consistent with the Ada tasking model, which we intend to propose to the Ada standardization committee as an extension of Ada.

Our research indicates that we can not only add a fine-grained concurrency mechanism to Ada, as shown in this paper, but this fine grained concurrency can be specialized to behave correctly in situations where Ada must meet difficult constraints, such as in hard real-time systems. These additional capabilities are being refined and will be presented in other works (an initial model is provided in [27]).

The programmer should also have the ability to control execution of parallel tasklettes, aborting loop iterations that are no longer necessary (e.g. in a search operation). This will be further investigated. Other constructs that can be provided with parallelism annotations are select statements, which are identified as future work.

Acknowledgments. We would like to thank the anonymous reviewers for their valuable comments. This work was partially supported by Portuguese 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 VIPCORE (ref. FCOMP-01-0124-FEDER-015006) and AVIACC (ref. FCOMP-01-0124-FEDER-020486) projects.

References

1. P. B. Hansen, “The Programming Language Concurrent Pascal”, IEEE Transactions on Software Engineering 1, 2 (June 1975), 199-207.
2. C. A. R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.
3. Ada Programming Language, ANSI/MIL-STD-1815A-1983, 1983.
4. Java Language Specification, http://www.oracle.com/java, last accessed February 2013.
5. H. Sutter and J. Larus, “Software and the concurrency revolution,” Queue, vol. 3, pp. 54–62, September 2005.
6. K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. “The landscape of parallel computing research: A view from Berkeley”, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006.
7. Ada 83 Rationale, available at http://www.adaic.org/ada-resources/standards/ada83/, last accessed February 2013.
8. 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.
9. M. Hind , E. Schonberg, “Efficient Loop-Level Parallelism in Ada”, Proceedings of TriAda 91, October 1991.
10. 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.
11. B. Moore, “Parallelism generics for Ada 2005 and beyond”, Proceedings of the ACM SIGAda Annual Conference (SIGAda'10), October 2010.
12. H. Ali, L. M. Pinho, “A parallel programming model for Ada”, Proceedings of the ACM SIGAda Annual Conference (SIGAda'11), November 2011.
13. Hansen, P. B., “Structured Multiprogramming”, Communications of the ACM, Volume 15 Issue 7, July 1972.
14. M. Frigo, C. E. Leiserson, and K. H. Randall. “The implementation of the cilk-5 multithreaded language”. SIGPLAN Notice, 33:212-223, May 1998.
15. C. Leiserson, “The Cilk++ concurrency platform”, Proceedings of the 46th Annual Design Automation Conference , ACM New York, USA, 2009.
16. Intel, Cilk Plus, http://software.intel.com/en-us/articles/intel-cilk-plus/, last accessed February 2013.
17. Tucker Taft, “Designing ParaSail, a new programming language”, http://parasail-programming-language.blogspot.pt/, last accessed February 2013.
18. Intel. Threading Building Blocks, http://threadingbuildingblocks.org/, last accessed February 2013.
19. D. Lea, “A Java fork/join framework”, Proceedings of the ACM 2000 conference on Java Grande, JAVA '00, pages 36-43, New York, NY, USA, 2000. ACM.
20. A. Marowka, “Parallel computing on any desktop”. Communications of the ACM, 50:74-78, September 2007.
21. Microsoft. Task parallel library, http://msdn.microsoft.com/en-us/library/dd460717.aspx, last accessed February 2013.
22. J. G. P. Barnes, “Rationale for Ada 2012: 1 Contracts and aspects”, Ada User Journal, Volume 32 (4), December 2011.
23. R. D. Blumofe and C. E. Leiserson, “Scheduling multithreaded computations by work stealing”. Journal of the ACM, 46(5):720–748, 1999.
24. P. Halpern, “Strict Fork-Join Parallelism”, JTC1/SC22/WG21 N3409, September 2012.

25. R. E. Ladner, M. J. Fischer, "Parallel Prefix Computation", Journal of the ACM 27 (4): 831–838, 1980.
26. B. Moore, “A comparison of work-sharing, work-seeking, and work-stealing parallelism strategies using Paraffin with Ada 2005”, Ada User Journal, 32(1), March 2011, available online at http://www.ada-europe.org, last accessed February 2013.
27. B. Moore, S. Michell and L. M. Pinho, “Parallelism in Ada: General Model and Ravenscar”, 16th International Real-Time Ada Workshop, York, UK, April 2013.

Appendix - Parallel Recursive Subprogram Example

In the main part of the paper, an example was provided that illustrated the approach for fine grained parallelism control for parallel loops. This appendix provides a similar example that is intended to show the recursive subprogram case.

Once again a three state generic instantiation can be applied if the recursion needs to generate a result. Here we consider the recursive Fibonacci example.

As with the parallel loop example, the reducing operation is integer addition, therefore the first stage instantiation from the loop example can be reused.

For the second stage instantiation, the recursive subprogram generic needs to be instantiated. In this case, the work type, `Integer`, is the type of the top level work item to be processed, which corresponds to the `Value` parameter of the Fibonacci function. We can then provide the following instantiation for the second phase.

```
with Integer_Addition; -- from Section 5
with Ada.Parallel.Functional_Reduction.Recursion;
package Integer_Addition_Recursion is new
     Integer_Addition.Recursion (Work_Type => Integer);
```

For the third phase, we will assume that a parallel library writer has provided a work-seeking library for recursion. As with the parallel loop case, the instantiation is straightforward, since there are no formal parameters to the generic.

```
with Integer_Addition_Recursion;
with Ada.Parallel.Functional_Reduction.
                              Recursion.Work_Seeking;
package Work_Seeking_Integer_Addition_Recursion is new
   Integer_Addition_Recursion.Work_Seeking;
```

Now that the third phase instantiation exists, the application programmer can rewrite the Fibonacci example as follows to obtain a parallel result with fine-grained control of the parallelism.

```
with Work_Seeking_Integer_Addition_Recursion;
with Ada.Parallel.Task_Pools.Bounded;
package WSeIA renames
               Work_Seeking_Integer_Addition_Recursion;
```

```
function Fibonacci (Value : Natural) return Natural
   with Parallel => True, Worker_Count => 4,
        Parallel_Manager => WSeIA.Parallelism_Manager,
        Task_Pool => Parallel.Task_Pools.Default_Pool;
```

As seen, the `Parallel_Manager` aspect can be provided in the spec (or body) of the subprogram, but can be overridden by the caller code. It specifies a manager to be used when the subprogram is called with `Parallel => True`.

The body of Fibonacci can be written in very much the same style as it would have been for the sequential case. In this case, the implementer of the Work-seeking abstraction declares an atomic boolean variable [26], `Seeking_Work`, which is referenced from the users' code to see if there are idle workers looking for more work. Note that an attribute must be provided that permits access to the parallelism manager object for the local scope.

```
function Fibonacci (Value : Natural) return Natural is
  Sequential_Cutoff : constant Integer := 22;
begin
   if Value < 2 then
      return Value;
   elsif  Parallel'Manager.Seeking_Work and then
        Value > Sequential_Cutoff then
      return
        Parallel_Fibonacci (Value - 2)
           with Parallel => True
         + Parallel_Fibonacci (Value – 1);
   else
      return Fibonacci (Value - 2) +
             Fibonacci (Value - 1);
   end if;
end Fibonacci;
```