OpenMP tasking model for Ada: safety and correctness

Sara Royuela[1], Xavier Martorell[1], Eduardo Quinones[1], and Luis Miguel Pinho[2]

[1] Barcelona Supercomputing Center
sara.royuela,xavier.martorell,eduardo.quinones@bsc.es
[2] CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto lmp@isep.ipp.pt

Abstract. The safety-critical real-time embedded domain increasingly demands the use of parallel architectures to fulfill performance requirements. Such architectures require the use of parallel programming models to exploit the underlying parallelism. This paper evaluates the applicability of using OpenMP, a widespread parallel programming model, with Ada, a language widely used in the safety-critical domain.
Concretely, this paper shows that applying the OpenMP tasking model to exploit fine-grained parallelism within Ada tasks does not impact on programs safeness and correctness, which is vital in the environments where Ada is mostly used. Moreover, we compare the OpenMP tasking model with the proposal of Ada extensions to define parallel blocks, parallel loops and reductions. Overall, we conclude that the OpenMP tasking model can be safely used in such environments, being a promising approach to exploit fine-grain parallelism in Ada tasks, and we identify the issues which still need to be further researched.

1 Introduction

There is a clear trend towards the use of parallel computation to fulfill the performance requirements of real-time embedded systems in general, and safety-critical embedded systems in particular (e.g. autonomous driving). In that regard, the use of advanced parallel architectures, like many-core heterogeneous processors, is increasingly seen as the solution. These architectures rely on parallel programming models to exploit their massively parallel capabilities. Thus, there is a need to integrate these models in the development of safety-critical systems [21].

Safety-critical systems are commonly developed with programming languages where concepts as safety and reliability are crucial. In that respect, Ada is widely used in safety-critical and high-security domains such as avionics and railroad systems. The whole language is designed to keep safeness: it enforces strong typing, checks ranges in loops and so eliminating buffer overflows, provides actual contracts in the form of pre- and post-conditions, prevents access to deallocated memory, etc. A long list of language decisions allows compilers to implement correctness techniques to certify algorithms regarding their specification.

Ada supports a concurrency model (by means of Ada tasks and protected objects) that is mainly suitable for coarse-grained parallelism. Hence, there has

been a significant effort to add support for fine-grained parallelism to Ada, to benefit from parallel architectures. The existent proposal [26] enriches the Ada core with extensions that support parallel blocks and parallel loops (including reductions). This technique is based on the notion of *tasklets* [23]: concurrent logical units within an Ada task. Since adding parallelism also means adding a source of errors (due to concurrent accesses to global data and synchronizations) the proposal addresses safety using new annotations. With that, the compiler is able to detect data race conditions[3] and blocking operations[4].

This paper evaluates the use of the OpenMP [1] tasking model to express fine-grained parallelism in Ada. OpenMP was born in the 90's out of the need for standardizing the different vendor specific directives related to parallelism. The language has successfully emerged as the de facto standard for shared-memory systems. This is the result of being successfully used for decades in the high-performance computing (HPC) domain. Furthermore, OpenMP has recently gained much attention in the embedded field owing to the augmentations of the latest specifications, which address the key issues in heterogeneous embedded systems: a) the coupling of a main host processor to one or more many-core accelerators, where highly-parallel code kernels can be offloaded for improved performance/watt; and b) the capability of expressing fine-grained, both structured and unstructured, and highly-dynamic task parallelism.

This paper shows how OpenMP can be integrated with Ada, and how correctness and thus safety are preserved when using the OpenMP tasking model [9] by virtue of compiler analyses. These analyses allow both compile-time detection of errors that may cause runtimes to break or hang, and automatic amendment of errors introduced due to a wrong usage of the user-driven parallelism. The OpenMP tasking model implements an execution pattern similar to the *tasklet* model, for an OpenMP task[5] resembles a *tasklet*. Interestingly, both models map to state-of-the-art scheduling methods, enabling to provide timing guarantees to OpenMP applications [24]. Such points make OpenMP particularly relevant for embedded heterogeneous systems, which typically run applications that can be very well modeled as periodic task graphs.

There are however a few caveats. First, the interplay between OpenMP and Ada runtimes, each with its own model. Second, although the tasking model of OpenMP has been demonstrated to be analyzable for real-time systems using the limited preemptive scheduling model [30], it is still ongoing effort to make it a standard offering. Finally, it remains as a future work to evaluate the complete OpenMP language, including its thread-based model (see Section 4.1).

---

[3] A *race condition* occurs when two or more accesses to the same variable are concurrent and at least one is a write.

[4] *Blocking operations* are defined in Ada to be one of the following: entry calls; select, accept, delay and abort statements; task creation or activation; external calls on a protected subprogram with the same target object as that of the protected action; and calls to a subprogram containing blocking operations.

[5] An *OpenMP task* is a specific instance of executable code and its data environment, generated when a thread encounters a given construct (i.e. `task`, `taskloop`, `parallel`, `target`, or `teams`)

2 Motivation: why OpenMP?

Programming multi-cores is difficult due to the multiple constraints it involves. Hence, the success of a multi-core platform relies on its productivity, which combines performance, programmability and portability. With such a goal, multitude of programming models coexist. The different approaches are grouped as follows:

Hardware-centric models aim to replace the native platform programming with higher-level, user-friendly solutions, e.g. Intel® TBB [27] and NVIDIA® CUDA [5]. These models focus on tuning an application to match a chosen platform, which makes their use a neither scalable nor portable solution.

Application-centric models deal with the application parallelization from design to implementation, e.g. OpenCL [34] and OmpSs [13]. Although portable, these models may require a full rewriting process to accomplish productivity.

Parallelism-centric models allow users to express typical parallelism constructs in a simple and effective way, and at various levels of abstraction, e.g. POSIX threads (Pthreads) [11], MPI [33] and OpenMP [12]. This approach allows flexibility and expressiveness, while decoupling design from implementation.

Given the vast amount of options available, there is a noticeable need to unify programming models for many-cores [36]. In that sense, OpenMP has proved many advantages over its competitors. On the one hand, different evaluations demonstrate that OpenMP delivers tantamount performance and efficiency compared to highly tunable models such as TBB [16], CUDA [19], OpenCL [31], and MPI [17]. On the other hand, OpenMP has different advantages over low level libraries such as Pthreads: a) it offers robustness without sacrificing performance [18], and b) OpenMP does not lock the software to a specific number of threads. Another advantage is that the code can be compiled as a single-threaded application just disabling support for OpenMP, thus easing debugging.

The use of OpenMP presents three main advantages. First, an expert community has constantly reviewed and augmented the language for the past 20 years. Thus, less effort is needed to introduce fine-grained parallelism in Ada. Second, OpenMP is widely implemented by several chip and compiler vendors (e.g. GNU [2], Intel® [4], and IBM [3]), meaning that less effort is needed to manage parallelism as the OpenMP runtime will manage it. Third, OpenMP provides greater expressiveness due to years of experience in its development. The language offers several directives for parallelization and synchronization, along with a large number of clauses that allow to contextualize concurrency, providing a finer control of the parallelism. Overall, OpenMP is a good candidate to introduce fine-grained parallelism to Ada by virtue of its benefits.

Despite its benefits, there is still work to do to fulfill the safety-critical domain requirements. Firstly, OpenMP is not reliable because it does not define any recovery mechanism. In that regard, different approaches have been proposed and some of them have been already adopted, which we discuss in Section 5.3. Secondly, both programmers and compilers must satisfy some requirements to make possible whole program analysis (such as programmers adding information in headers libraries, and compilers implementing techniques like IPO [7]).

3 Ada language extensions for fine-grain parallelism

Ada includes tasking features as part of the standard by means of tasks, which are entities that denote concurrent actions, and inter-task communication mechanisms such as protected objects or *rendezvous*. However, this model is mainly suitable for coarse-grained parallelism due to its higher overhead [32].

Efforts exist to extend Ada with a fine-grained parallel model based on the notion of *tasklets* [23], where parallelism is not fully controlled by the programmer: the programmer specifies the parallel nature of the algorithm, and the compiler and the runtime have the freedom to organize parallel computations.

Based on this model, specific language extensions have been proposed [35] to cover two cases where parallelization is suitable: parallel blocks and parallel loops, including reductions. The following subsections present the syntax and semantics proposed (which are being considered for future versions of the Ada language [8]), as well as how safety is kept in this model.

3.1 Parallel blocks

A parallel block (Listing 1.1) denotes two or more parts of an algorithm that can be executed in parallel. A transfer of control[6] or exception[7] within one parallel sequence aborts the execution of parallel sequences that have not started, and potentially initiates the abortion of those sequences not yet completed[8]. Once all parallel sequences complete, then the transfer of control or exception occurs.

Listing 1.1. Parallel block syntax with proposed Ada extensions

```
1 parallel
2     sequence_of_statements
3 and
4     sequence_of_statements
5 {and
6     sequence_of_statements}
7 end parallel;
```

Listing 1.2. Parallel loop syntax with proposed Ada extensions

```
1 for i in parallel lb..ub loop
2    sequence_of_statements
3 end loop;
```

3.2 Parallel loop

In a parallel loop (Listing 1.2), iterations may execute in parallel. Each iteration can be treated as a separate unit of work. However, this may introduce too much overhead from: a) the creation of the work item, b) the communication of results, and c) the synchronization of shared data (protected objects). To palliate this, both the compiler and the runtime are given the freedom to chunk iterations. Although not mandatory, programmers may gain control by defining

[6] A *transfer of control* causes the execution of a program to continue from a different address instead of the next instruction (e.g. a return instruction).

[7] *Exceptions* are anomalous conditions requiring special processing. Ada has predefined exceptions (language-defined run-time errors) and user-defined exceptions.

[8] The rules for abortion of parallel computations are still under discussion [25].

sized chunks. The proposal reveals the necessity of providing support for per-thread copies of relevant data to deal with data dependencies and shared data.

The authors also introduce the concept of a parallel array; that is data being updated within a parallel loop. The syntax is shown in Listing 1.3, where the use of <> indicates an array of unspecified bounds. In that case, the compiler may choose the size based on the number of chunks chosen for the parallelized loops where the array is used. Alternatively, the programmer may provide a bound, thus forcing a specific partitioning. The rule regarding transfer of control and exceptions presented for parallel blocks also applies here. For this purpose, each chunk is treated as equivalent to separate sequences of a parallel block.

Listing 1.3. Not chuncked parallel array with proposed Ada extensions

```
1  Arr : array (parallel <>) of a_type
2      := (others => initial_value);
```

3.3 Parallel reduction

The authors of the proposed Ada extensions define a reduction as a common operation for values in a parallel array that consists in combining the different values of the array at the end of the processing with the appropriate reduction operation. Syntax for parallel reductions is still under discussion [25] and the current proposal is to define this reduction in the type as in Listing 1.4.

Listing 1.4. Parallel reduction with proposed Ada extensions

```
1  ...
2     type Partial_Array_Type is new array (parallel <>) of Float;
3     with Reducer => "+", Identity => 0.0;
4     Partial_Sum : Partial_Array_Type := (others => 0.0);
5     Sum : Float := 0.0;
6  begin
7     for I in parallel Arr'Range loop
8         Partial_Sum(<>) := Partial_Sum(<>) + Arr(I);
9     end loop;
10    Sum := Partial_Sum(<>)'Reduced; -- reduce value either here or
11                                    -- during the parallel loop
12 ...
```

3.4 Safety

Despite the clear benefits of parallel computation in terms of performance, parallel programming is complex and error prone, and that may compromise correctness and so safety. Hence, it is paramount to incorporate compiler and run-time techniques that detect errors in parallel programming.

There are two main sources of errors when dealing with parallel code: a) the concurrent access to shared resources in a situation of *race condition*, and b) an error in the synchronization between parallel operations leading to a *deadlock*. To guarantee safety, Ada parallel code must use atomic variables and protected

objects to access shared data. Moreover, the compiler shall be able to complain if different parallel regions might have conflicting side-effects.

In that respect, due to the hardship of accessing the complete source code to perform a full analysis, the proposed Ada extensions suggests a two-fold solution [35]: a) eliminate race conditions by adding an extended version of the SPARK `Global` aspect to the language (this will help the compiler to identify those memory locations that are read and written without requiring access to the complete code); and b) address deadlocks by the defined execution model, together with a new aspect called `Potentially_Blocking` that indicates whether a subprogram contains statements that are potentially blocking.

4 OpenMP for fine-grained parallelism in Ada

In this paper, we propose a complementary approach for exploiting fine-grain parallelism in Ada: OpenMP. Our approach is motivated by the threefold advantage of a) being a well-known parallel programming model supported by many chip and compiler vendors, b) offering a simple yet exhaustive interface, and c) providing greater expressiveness as a result of years of experienced development.

4.1 OpenMP execution model

OpenMP provides two different models of parallelism:

Thread-parallelism, which defines a conceptual abstraction of user-level threads that work as proxies for physical processors. This model enforces a rather structured parallelism. Representative constructs are `for` and `sections`.

Task-parallelism, tasking model hereafter, which is oblivious of the physical layout. Programmers focus on exposing parallelism rather than mapping parallelism onto threads. Representative constructs are `task` and `taskloop`.

An OpenMP program begins as a single thread of execution, called the *initial thread*. Parallelism is achieved through the `parallel` construct. When such a construct is found, a *team* of threads is spawned. These are joined at the *implicit barrier* encountered at the end of the parallel region. Within that region, the threads of the team execute work. This is the so-called *fork-join model*. Then, within the parallel region, parallelism is achieved by means of different constructs: `for`, `sections` and `task`, among others.

Mutual exclusion is accomplished via the `critical` and `atomic` constructs, and synchronization by means of the `barrier` construct. Additionally, the tasking model offers the `taskwait` construct to impose a less restrictive synchronization (while a `barrier` synchronizes all threads in the current team, a `taskwait` only synchronizes descendant tasks of the binding task).

Fig. 2 shows the execution model of a parallel block with a loop implemented using the `parallel for` directive, where all spawned threads work in parallel from the beginning of the parallel region as long as there is work to do. Fig. 1 shows the model of a parallel block with tasks. In this case, the `single`

construct restricts the execution of the parallel region to only one thread until a `task` construct is found. Then, another thread (or the same, depending on the scheduling policy), concurrently executes the code of the task.

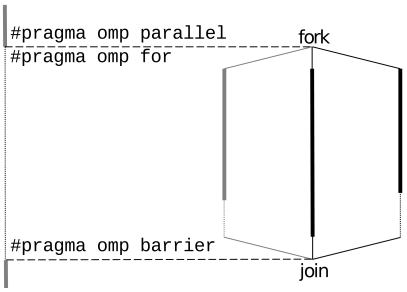

Fig. 1. Fork-join model with unstructured parallelism

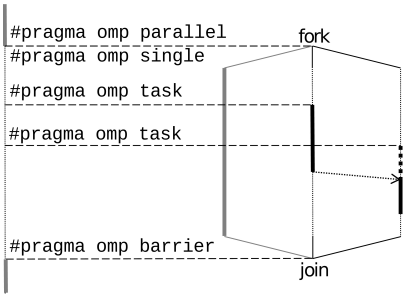

Fig. 2. Fork-join model with structured parallelism

The tasking model adapts better to the parallelism model proposed for Ada, which is oblivious of the threads as well. Thus, even if a thread-parallel version is possible, we focus on the tasking model, remaining the other as future work.

4.2 Data scoping

One of the most interesting characteristics of OpenMP is that it allows a rich definition of the scoping of variables involved in the parallel computation by means of data-sharing clauses. This scoping can be one of the following:

- *private*: a new fresh variable is created within the scope.
- *firstprivate*: a new variable is created in the scope and initialized with the value of the original variable.
- *lastprivate*: a new variable is created within the scope and the original variable is updated at the end of the execution of the region (only for tasks).
- *shared*: the original variable is used in the scope, thus opening the possibility of data race conditions.

The use of data-sharing clauses is particularly powerful to avoid unnecessary synchronizations as well as race conditions. All variables appearing within a construct have a default data-sharing defined by the OpenMP specifications (Section 2.15.1 [1]). Data-scoping rules are not based on the use of the variables, but on their storage. Thus, users are required to explicitly scope many variables, changing the default data-sharing values, in order to fulfill correctness (i.e., avoiding data races) and enhance performance (i.e., privatizing variables).

Listing 1.5 shows a simple C code with two tasks concurrently performing two multiplications. These tasks are synchronized in the `taskwait` directive previous to adding the two computed values. The code shows the default data-sharing of the variables derived following the data-scoping rules: `a`, `b` and `res`

are defined as shared because they have dynamic storage duration, whereas x and y are defined as firstprivate. This code however is not correct because the updated values of x and y are not visible outside the tasks. Hence, programmers must manually introduce the data-sharing clauses as shown in Listing 1.6.

Listing 1.5. OpenMP specification defined data-sharing clauses

```
1 int a, b, res;
2 int foo() {
3   #pragma omp parallel
4     // shared(a, b, res)
5   #pragma omp single nowait
6   {
7     int x, y;
8     #pragma omp task
9       // firstprivate(x) shared(a)
10      x = a*a;
11    #pragma omp task
12      // firstprivate(y) shared(b)
13      y = b*b;
14    #pragma omp taskwait
15    res = x + y;
16  }
17 }
```

Listing 1.6. OpenMP manually defined data-sharing clauses

```
1 int a, b, res;
2 int foo() {
3   #pragma omp parallel shared(res) \
4                        firstprivate(a, b)
5   #pragma omp single nowait
6   {
7     int x, y;
8     #pragma omp task shared(x) \
9                      firstprivate(a)
10      x = a*a;
11    #pragma omp task shared(y) \
12                     firstprivate(b)
13      y = b*b;
14    #pragma omp taskwait
15    res = x + y;
16  }
17 }
```

Manually defining data-sharing clauses is a cumbersome and error-prone process because programmers have to be aware of the memory model and analyze the usage of the variables. Fortunately, compiler analysis techniques have already proved that it is possible to automatically define data-sharing clauses [28] and statically catch incoherences in the user-defined attributes that may lead to non-deterministic results, runtime failures and loss of performance [29]. We further explain these in Section 5.

The possibility of defining data-sharing attributes makes an important difference with the proposed Ada extensions, where this task is allotted to the compiler. In that regard, OpenMP adds flexibility to the model without losing simplicity, as the attributes can still be discovered at compile time.

4.3 Supporting OpenMP in Ada

The current OpenMP specification is defined for C, C++ and Fortran. In the examples showed in Section 4.2 we use the syntax defined for C/C++. However, Ada does not group a sequence of statements by bracketing the group (as in C), but uses a more structured approach with a closing statement to match the beginning of the group. Since Ada already defines pragmas of the form `pragma Name (Parameter_List);`, we propose introducing a new kind of pragma, the `pragma OMP`, together with the directive name (e.g. `task`, `barrier`, etc.).

Listing 1.7 shows an example of the proposed syntax when the OpenMP construct applies to one statement, and Listing 1.8 shows an example where the construct applies to more than one statement.

OpenMP defines the argument of a data-sharing clause as a list of items. This does not match directly with the syntax allowed in Ada for pragmas, which is

Listing 1.7. OpenMP proposed syntax pragmas applying to one statement

```
pragma OMP (taskloop, num_tasks=>N);
for i in range 0..I loop
   ...    -- statements here
end loop;
```

Listing 1.8. OpenMP proposed syntax for pragmas applying to several statements

```
pragma OMP (task, shared=>var);
begin
   ...    -- statements here
end;
```

shown in Listing 1.9. In order to simplify the syntax needed to define data-sharing clauses, we propose to extend the definition of `pragma_argument_identifier` with a list of expressions. We will use this proposed syntax for the rest of the document.

Listing 1.9. Ada syntax for pragmas

```
pragma ::=
   pragma identifier [(pragma_argument_association {, pragma_argument_association})];
pragma_argument_association ::=
     [pragma_argument_identifier =>] name
   | [pragma_argument_identifier =>] expression
```

4.4 Parallel blocks

As previously introduced, a parallel block denotes two or more concurrent sections. In OpenMP a parallel block can be written so that each parallel region is wrapped in a task, and all tasks are wrapped in a parallel region. We use the parallel computation of the Fibonacci sequence to illustrate this scenario. Listing 1.10 shows the implementation using Ada extensions, and Listing 1.11 shows the OpenMP implementation.

Listing 1.10. Parallel Fibonacci sequence with Ada extensions

```
1 if N < 2 then
2    return N;
3 parallel
4    X:= Fibonacci(N - 2);
5 and
6    Y:= Fibonacci(N - 2);
7 end parallel;
8 return X + Y;
```

Listing 1.11. Parallel Fibonacci sequence with OpenMP tasks

```
1  if N < 2 then
2     return N;
3  pragma OMP (parallel, shared=>X,Y,
4                         firstprivate=>N);
5  pragma OMP (single, nowait);
6  begin
7     pragma OMP (task, shared=>X,
8                       firstprivate=>N);
9     X:= Fibonacci(N - 2);
10    pragma OMP (task, shared=>Y,
11                      firstprivate=>N)
12    Y:= Fibonacci(N - 2);
13 end
14 return X + Y;
```

In the Ada version, the compiler can detect that no unsafe access is made to `N`, `X` or `Y` in the parallel block, thus concluding no synchronization is required (except the one at the end of the parallel block). Furthermore, it can privatize `X` and `Y`, copying out their value after the parallel computation completes. This however, may harm performance due to the extra copies (it remains as a compiler decision). The logic behind the choice to make data-sharing transparent to the user is based on simplicity and readability, whilst safe.

In the OpenMP version, although programmers are not forced to define the data-scoping manually (since the compiler can detect the proper data-sharing attributes as it does in the Ada version), they can specify the intended model for data access. Hence, accesses to `X` and `Y` are marked as shared because there is no concurrency in the usage of these variables and they are both updated within the corresponding tasks and visible after the tasks. Additionally, the access to `N` is marked as firstprivate because the value is just read within the task. Since there is an implicit barrier at the end of the parallel construct, the return statement will always access the correct values of `X` and `Y`. This model is not as naive as the proposed Ada extensions, being a trade-off between simplicity and flexibility.

4.5 Parallel loop

As previously explained, a parallel loop defines a loop where iterations may be executed in parallel. The OpenMP tasking model offers the `taskloop` construct, which specifies that the iterations of the associated loops will be distributed across the tasks created by the construct, and executed concurrently. Users can control the number of tasks and their size with the following clauses:

- `num_tasks` defines the number of tasks created.
- `grain_size` defines the number of loop iterations assigned to each task.

We illustrate this scenario with the well-known matrix multiplication benchmark. Consider two matrices `M1` and `M2`, and the matrix `RES`, where their multiplication is stored. Listing 1.12 shows the code implemented with the syntax proposed for the Ada extensions, and Listing 1.13 shows the implementation using the OpenMP `taskloop` construct.

Listing 1.12. Parallel matrix multiplication with Ada extensions

```
1 for i in parallel 0..MAX_I loop
2   for j in range 0..MAX_J loop
3     for k in range 0..MAX_K loop
4       RES(i,j):= RES(i,j)
5                + M1(i,k) * M2(k,j);
6     end loop;
7   end loop;
8 end loop;
```

Listing 1.13. Parallel matrix multiplication with OpenMP taskloop

```
1 pragma OMP (parallel);
2 pragma OMP (taskloop,
3    private=>i, j, k,
4    firstprivate=>MAX_I, MAX_J, MAX_K,
5    shared=>RES, M1, M2,
6    grainsize=>size);
7 begin
8   for i in range 0..MAX_I loop
9     for j in range 0..MAX_J loop
10      for k in range 0..MAX_K loop
11        RES(i,j):= RES(i,j)
12                 + M1(i,k) * M2(k,j);
13      end loop;
14    end loop;
15  end loop;
16 end
```

Again, OpenMP allows more expressiveness by virtue of the data-sharing clauses. In the Ada version the compiler may not be able to determine that parallel access to RES are not data races. Moreover, the OpenMP version also allows controlling the granularity of the parallelization whereas the Ada extensions are limited to defining the number of elements of a parallel array.

4.6 Parallel reduction

For the Ada extensions, a reduction is an operation defined over the elements of a parallel array. OpenMP relaxes this constraint and defines a reduction as a parallel operation which result is stored in a variable. Different implicitly declared reduction identifiers are defined in OpenMP (e.g. +, -, *, etc.). Additionally, the specification allows user defined reductions with the syntax specified in Listing 1.14. There, `reduction_identifier` is either a base language identifier or an implicitly declared identifier, `typename_list` is a list of type names, `combiner` is the reduction expression, and `initializer_clause` indicates the value to be used to initialize the private copies of the reduction.

Listing 1.14. OpenMP syntax for user-declared reductions

```
#pragma omp declare reduction \
   (reduction_identifier : typename_list : combiner) \
   [initializer_clause]
```

The reduction itself is implemented in OpenMP by means of a clause that can be added to multiple constructs like `parallel` and `for` among others. The possibilities with OpenMP reductions underscore their versatility in the face of the proposed Ada extensions. Until OpenMP 4.5, the reduction is limited to the thread-parallelism model. Nonetheless, the planned OpenMP 5.0 [6] defines reductions for taskloops as well. Listing 1.15 shows the syntax adapted to our proposal for Ada. Clauses `num_tasks` and `grain_size` can still be used.

Listing 1.15. OpenMP parallel taskloops reduction example

```
pragma OMP parallel (taskloop reduction=>+,TOTAL);
begin
   for i in range 0..MAX_I loop
   TOTAL := Arr(i);
   end loop;
end
```

OpenMP specifies that the number of times the combiner is executed, and the order of these executions is unspecified. This means that different executions may deliver different results. To avoid this unspecified behavior some restrictions can be added to the use of OpenMP reductions in safety-critical embedded domains, such as: limiting the operations to those that are associative and commutative, and forbidding the use of floating point types.

4.7 Mutual exclusion

In OpenMP, mutual exclusion is achieved by means of two constructs: `critical` and `atomic`. While the `critical` construct restricts the execution of its associated structured block to a single thread at a time, the `atomic` construct ensures that a specific storage location is accessed atomically.

The `atomic` construct is very restrictive in the sense that it accepts a limited number of associated statements of the form defined in the specifications (Section

2.13.6 [1]). Consequently, atomics do not represent a threat concerning safety because no deadlock may be caused by their use. Differently, the `critical` construct accepts any kind of statement and, as a consequence, deadlocks may appear. Although OpenMP forbids nesting `critical` constructs with the same name, this is not sufficient to avoid deadlocks. A `critical` construct containing a task scheduling point[9] may cause a deadlock if the thread executing the critical region jumps to a region containing a `critical` construct with the same name. Section 5 discusses solutions and the work that needs to be done to integrate OpenMP and Ada mutual exclusion mechanisms.

5 Safety in OpenMP

Compilers are key tools to anticipate bugs that may appear at run-time, becoming fundamental when developing safety-critical systems. Although most OpenMP compilers do not diagnose common mistakes that cause execution errors, previous works are encouraging. The following subsections tackle the situations that jeopardize safety when using OpenMP, showing the existent solutions and also explaining additional proposals. The argumentation is orthogonal to the underlying language. The techniques used have been implemented in compilers for C/C++, hence it is possible provide them in Ada compilers as well.

5.1 Correctness: data races and synchronization

Detecting exact data races at compile time is an open challenge, and static tools still struggle to obtain no false negatives and minimal false positives. Current mechanisms have been proved to work properly on specific subsets of OpenMP such as having a fixed number of threads [22] or avoiding the use of non-affine constructs [10] [10]. A more general approach can be used to determine the regions of code that are definitely non-concurrent [20]. Although it is not an accurate solution, it will never deliver false negatives. The previously mentioned techniques can be combined to deliver conservative and fairly accurate results.

It is unattainable that compilers are able to interpret the semantics of an algorithm, thus correctness techniques are limited. However, it is feasible for compilers to observe situations that are incoherent or may lead to runtime errors. In that regard, static analysis techniques have proved to be able to catch tasks and variables that are not properly synchronized, causing both non-deterministic results (due to data races) and runtime failures (due to wrong synchronization -e.g. a task using as shared an automatic storage variable after its lifetime has

[9] A *task scheduling point (TSP)* is a point during the execution of the a task region at which it can be suspended to be resumed later, and where the executing thread may switch to a different task region. OpenMP defines the list of TSP to be: the point immediately following the generation of an explicit task, after the point of completion of a task region, and in a taskwait region among others.

[10] *Non-affine* constructs are non-affine subscript expressions, indirect array subscripts, use of structs, non-affine loop bounds, and non-affine `if` conditions, among others.

ended-) [29]. Such techniques adopt a conservative approach, in the sense that performance is secondary when correctness is on the line (e.g. privatize a variable in order to avoid a race-condition). The compiler can provide a report, so users may act in accordance with the decisions taken.

Additionally, it has been demonstrated that the compiler can determine the data-sharing attributes of a task provided that all code concurrent with the task is accessible at compile time [28]. When a variable cannot be automatically determined, the user is warned to manually scope it. Since limitations concern full access to the code, whole program analysis techniques can resolve the problem. Furthermore, the `Potentially_Blocking` aspect proposed by the authors of the Ada extensions could be used to enable the detection of such problems at compile time, avoiding the necessity of program analysis in some cases.

Listing 1.16. Example of data sharing

```
1  function Pi (n_steps: in Integer) return Float is
2     x : Float;
3     sum, res: Float := 0.0;
4     step : Float := 1.0/Float(n_steps);
5  begin
6     pragma OMP(task); -- private=>x, firstprivate=>step, shared=>sum
7     begin  -- OpenMP Task 1
8          x := 0;
9          for I in 1 .. n_steps/2-1 loop
10            x := (Float(I)+0.5)*step;
11            pragma OMP(atomic);
12            sum := sum + 4.0/(1.0+x*x);
13         end loop;
14    end;
15    pragma OMP(task); -- private=>x, firstprivate=>step, shared=>sum
16    begin -- OpenMP Task 2
17         x := 0;
18         for I in n_steps/2 .. n_steps  loop
19            x := (Float(I)+0.5)*step;
20            pragma OMP(atomic);
21            sum := sum + 4.0/(1.0+x*x);
22         end loop;
23    end;
24    pragma OMP(taskwait);
25    pragma OMP(task); -- firstprivate=>step,sum, shared=>res
26    begin -- OpenMP Task 3
27         res := step * sum;
28    end;
29    pragma OMP(taskwait);
30    return res;
31 end Pi;
```

Listing 1.16 shows a potential OpenMP Ada code computing the number pi as an example of the application of the correctness techniques implemented in an OpenMP-compliant compiler (Mercurium [15]). The constructs added by the user to express parallelism are emphasized, while the parts discovered by the compiler in order for the code to be correct are underlined. Both synchronization (with the `taskwait` construct) and mutual exclusion (with the `atomic` construct) can be decided by the compiler. Also the data-sharing clauses needed to avoid race-condition (on variable `x`) and for the code to make sense (variable

sum) are automatically determined. Given the code with none of the underlined clauses and constructs, the compiler detects:

- For tasks 1 and 2: variable x is in a race condition due to concurrency between tasks 1 and 2. Additionally, the value of this variable is defined and used (in that order) within the tasks only, thus the variable can be privatized.
- In all tasks, variable `step` is a read-only scalar. Thus, it can be firstprivate.
- Variable `sum` is updated and read among the tasks, so it has to be shared. However, it is in a race condition, so accesses must be synchronized. On the one hand, tasks 1 and 2 read-write the variable, thus the compiler adds an `atomic` construct. On the other hand, task 3 only reads the variable, so the compiler adds a `taskwait` before the task.
- For task 3, variable `res` must be used before exiting the function. Otherwise, if the task is deferred until the function returns, the variable will no longer exist. Thus, a `taskwait` must be inserted after the task.

5.2 Deadlocks

OpenMP offers two ways to synchronize threads: via directives, such as `critical` and `barrier`, and via runtime routines, such as `omp_set_lock`. In both cases, a deadlock may occur only if a thread that holds a lock tries to obtain the same lock. This is a consequence of being a model focused in languages which do not provide higher-level concurrency mechanisms. Ada code will use protected objects, so work still needs to be performed to integrate both Ada and OpenMP runtime systems.

5.3 Error handling

In the critical domain it is important to understand and specify behaviour upon failures. The technique to enable such property is error handling. There are three main mechanisms for handling errors: exceptions, error codes and call-backs. Each method has advantages and disadvantages. The first fits perfectly in the structure of exception-aware languages such as Ada. The second is suitable for exception-unaware languages such as C. Finally, the third has the advantage of isolating the code that is to be executed when an exception occurs. Although only some minor mechanisms have been included in the specifications (i.e. cancellation constructs), there are different proposals to improve OpenMP reliability by adopting error handling mechanisms in OpenMP [14] [37]. The integration of these with Ada exceptions is also in need for future work.

6 Conclusions and future work

There is an oportunity for extending Ada with fine-grained parallelism. Extensions to the language with such purpose have already been presented and are still under discussion. Nevertheless, the increasing variety of platforms and their

specific programming models force programmers to master multiple complex languages. Comparisons among multiple parallel programming models show the need to provide a common programming model for many-cores. In that regard, OpenMP is the perfect candidate, for it has succesfully emerged as the de facto standard for shared-memory parallel programming, and starts to be used for distributed memory systems. In this paper, we show how the OpenMP tasking model can successfully be applied to Ada to define fine-grained parallelism in the form of parallel blocks, parallel loops and parallel reductions. The use of OpenMP is not a threat regarding safeness, for we have shown that both compilers and runtimes can be used to check correctness and recover from failures.

There is nevertheless work to be done to understand the actual impact of mixing OpenMP and Ada tasks, because both will be mapped to the underlying threads of the operating system. Another area for future work is a potential combination of the OpenMP tasking model and the proposed syntax for Ada parallel extensions. The underlying parallel models are sufficiently close to enable this to be considered further.

References

1. OpenMP 4.5. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf (2015)
2. GOMP. https://gcc.gnu.org/projects/gomp/ (2016)
3. IBM Parallel Environment. http://www-03.ibm.com/systems/power/software/parallel/ (2016)
4. Intel® OpenMP* Runtime Library. https://www.openmprtl.org (2016)
5. NVIDIA® CUDA C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (2016)
6. OpenMP Technical Report 4: version 5.0 Preview 1. http://www.openmp.org/wp-content/uploads/openmp-tr4.pdf (2016)
7. Intel Interprocedural Optimization. https://software.intel.com/en-us/node/522666 (2017)
8. Ada Rapporteur Group: AI12-0119-1. http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai12s/ai12-0119-1.txt (2016)
9. Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., Zhang, G.: The design of OpenMP tasks. TPDS 20(3) (2009)
10. Basupalli, V., Yuki, T., Rajopadhye, S., Morvan, A., Derrien, S., Quinton, P., Wonnacott, D.: ompVerify: polyhedral analysis for the OpenMP programmer. In: IWOMP. pp. 37–53. Springer (2011)
11. Butenhof, D.R.: Programming with POSIX threads. Addison-Wesley (1997)
12. Chapman, B., Jost, G., Van Der Pas, R.: Using OpenMP: portable shared memory parallel programming, vol. 10. MIT press (2008)
13. Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: Ompss: a proposal for programming heterogeneous multi-core architectures. Parallel Processing Letters 21(02), 173–193 (2011)
14. Duran, A., Ferrer, R., Costa, J.J., Gonzàlez, M., Martorell, X., Ayguadé, E., Labarta, J.: A proposal for error handling in OpenMP. IJPP 35(4), 393–416 (2007)
15. Ferrer, R., Royuela, S., Caballero, D., Duran, A., Martorell, X., Ayguadé, E.: Mercurium: Design decisions for a s2s compiler. In: Cetus Users and Compiler Infastructure Workshop in conjunction with PACT (2011)

16. Kegel, P., Schellmann, M., Gorlatch, S.: Using OpenMP vs. Threading Building Blocks for Medical Imaging on Multi-Cores. In: Europar. Springer (2009)
17. Krawezik, G., Cappello, F.: Performance comparison of MPI and three OpenMP programming styles on shared memory multiprocessors. In: SPAA. ACM (2003)
18. Kuhn, B., Petersen, P., O'Toole, E.: OpenMP versus threading in C/C++. Concurrency - Practice and Experience 12(12), 1165–1176 (2000)
19. Lee, S., Min, S.J., Eigenmann, R.: OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. SIGPLAN Not. 44(4), 101–110 (2009)
20. Lin, Y.: Static nonconcurrency analysis of openmp programs. In: IWOMP, pp. 36–50. Springer (2008)
21. Lisper, B.: Towards parallel programming models for predictability. In: OASIcs. vol. 23. Schloss Dagstuhl LZI (2012)
22. Ma, H., Diersen, S.R., Wang, L., Liao, C., Quinlan, D., Yang, Z.: Symbolic analysis of concurrency errors in openmp programs. In: ICPP. pp. 510–516. IEEE (2013)
23. Michell, S., Moore, B., Pinho, L.M.: Tasklettes–a fine grained parallelism for Ada on multicores. In: Ada-Europe. pp. 17–34. Springer (2013)
24. Pinho, L., Nelis, V., Yomsi, P., Quinones, E., Bertogna, M., Burgio, P., Marongiu, A., Scordino, C., Gai, P., Ramponi, M., Mardiak, M.: P-SOCRATES: A parallel software framework for time-critical many-core systems. MICPRO 39(8) (2015)
25. Pinho, L.M., Michell, S.: Session Summary: Parallel and Multicore Systems. Ada Lett. 36(1), 83–90 (2016)
26. Pinho, L.M., Moore, B., Michell, S., Taft, S.T.: Real-Time Fine-Grained Parallelism in Ada. ACM SIGAda Ada Letters 35(1), 46–58 (2015)
27. Reinders, J.: Intel Threading Building Blocks. O'Reilly & Associates, Inc. (2007)
28. Royuela, S., Duran, A., Liao, C., Quinlan, D.J.: Auto-scoping for OpenMP tasks. In: IWOMP. pp. 29–43. Springer (2012)
29. Royuela, S., Ferrer, R., Caballero, D., Martorell, X.: Compiler analysis for OpenMP tasks correctness. In: Computing Frontiers. p. 7. ACM (2015)
30. Serrano, M.A., Melani, A., Bertogna, M., Quinones, E.: Response-time analysis of DAG tasks under fixed priority scheduling with limited preemptions. In: DATE. pp. 1066–1071. IEEE (2016)
31. Shen, J., Fang, J., Sips, H., Varbanescu, A.L.: Performance gaps between OpenMP and OpenCL for multi-core CPUs. In: ICPPW. pp. 116–125. IEEE (2012)
32. Sielski, K.L.: Implementing Ada 83 and Ada 9X Using Solaris Threads. Ada: Towards Maturity 6, 5 (1993)
33. Snir, M.: MPI–the Complete Reference: The MPI core, vol. 1. MIT press (1998)
34. Stone, J.E., Gohara, D., Shi, G.: OpenCL: A parallel programming standard for heterogeneous computing systems. CS&E 12(3), 66–73 (2010)
35. Taft, S.T., Moore, B., Pinho, L.M., Michell, S.: Safe parallel programming in Ada with language extensions. ACM SIGAda Ada Letters 34(3), 87–96 (2014)
36. Varbanescu, A.L., Hijma, P., Van Nieuwpoort, R., Bal, H.: Towards an effective unified programming model for many-cores. In: IPDPS. pp. 681–692. IEEE (2011)
37. Wong, M., Klemm, M., Duran, A., Mattson, T., Haab, G., de Supinski, B.R., Churbanov, A.: Towards an error model for OpenMP. In: IWOMP. pp. 70–82. Springer (2010)