An Experience in Ada Multicore Programming: Parallelisation of a Model Checking Engine

Franco Mazzanti

Consiglio Nazionale delle Ricerche, ISTI - Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", Pisa, Italy
franco.mazzanti@isti.cnr.it

Abstract. Even if multicore architectures are nowadays extremely widespread, the exploitation of this easily available degree of parallelism is not always straightforward. In this paper we describe the experience gained in our ongoing effort to parallelise the model checking engine of a family of model checkers (KandISTI) developed at ISTI. The main focus of our experimentation is the evaluation of the minimal efforts needed to take advantage of our everyday multicore hardware for model checking purposes. Our early results relative to an initial fragment of the logic show a speedup factor of about 2.5 when 4 physical cores are available. This result, however, can only be achieved by complementing the initial high level Ada design with a second round of code fine-tuning which exploits nonstandard low level features in the implementation of the needed thread-safe data structures.

Keywords: model checking, parallel programming, multicore processor architectures, Ada programming language

1 Introduction

KandISTI [4], is a family of model checkers being developed at ISTI in the last ten years aimed to the experimentation of innovative formal verification techniques. The family is constituted by four model checkers each of which is oriented to a particular system design approach, but all of which share the same underlying abstract model and verification engine. The basic underlying idea behind KandISTI is that the evolution in time of the system behaviour can be seen as a graph where both edges and states are associated with sets of (composite) labels [9]. Labels on the states represent basic state properties, and labels on the edges represent properties of system transitions. The different flavours of the various tools have to do with the choice of one of the supported specifications languages, which range from process algebras to sets of UML-like statecharts.

The properties that can be verified on such a graph are expressed in a branching time, state and action based, temporal logic [8] which includes both the basic fix point operators and the more friendly (A)CTL-like [7] operators. The verification of a formula and the generation of the relevant part of the system evolutions

graph occur *on the fly*. Given an initial system state and a top-level formula, only the actually needed subformulas of the initial formula are analysed, and only the actually needed next states of the current state are generated. The graph generation is driven *on demand* from the ongoing formula verification. All the various tools of the KandISTI framework are programmed in Ada, and are freely usable online through a public web interface.[1] These tools are being used mainly for didactic and academic research purposes, and have not yet reached the engineering level actually needed for full-scale industrial use.

All these tools are usually executed upon consumer level hardware that normally supports at least 4 cores. The current version of KandISTI is however not able to exploit this easily available degree of parallelism. We are therefore interested to observe which degree of redesign is needed and to measure the amount of benefits that could be gained by the exploitation of the multicore structure of our systems. The parallelisation effort is still ongoing, but the early results already show a reasonable picture of the situation and of the novel design.

In Section 2 we describe the overall structure of the original sequential version of the tools, while in Section 3 we describe two possible approaches for introducing some parallelism during the verification. In Section 4 we describe a recent case study to which our KandISTI framework has been applied and we show the results of the parallelisation efforts. In Section 5 we draw some final conclusions.

2 The Basic Sequential Approach

In the following we will consider just a few operators of the logic supported by KandISTI, which are however sufficient to illustrate the overall structure of the verification mechanism. One of these is the *EX* (Exists neXt) operator.

The formula: *EX*{*action*}*subform* holds on a state s if and only if there exists an outgoing transition from s whose edge labels satisfy the transition predicate *action* and which leads to a target state s' in which the formula *subform* holds. A second operator we consider is the AG (Always Globally) operator.

The formula: AG *subform* holds on a state s if and only if for all the states reachable (in any number of steps) from s, the formula *subform* holds. We also make use of the $True$ formula which holds in any state, and the $true$ action predicate which holds for any transition.

The skeleton of the sequential verification algorithm is quite simple: we have an EvalFormula function which, depending on the kind of formula passed to it, dispatches to the appropriate specific version returning a TT or FF value according to the validity of the formula on the given state. When the formula to be evaluated is of the *EX* kind, the evaluation has the following structure:

```
1: function EvalEX(Formula, State) return Computation_Status is
2:   Result: Computation_Status;
3: begin
```

[1] http://fmt.isti.cnr.it/kandisti

```
 4:   Result = Computations_DB.CheckComputation (Formula, State)
 5:   if  Result = NOT_YET_STARTED then
 6:     Result := FF;
 7:     for E of GetEvolutions(State) loop
 8:       if E.Labels satisfy Formula.Action then
 9:         Result := EvalFormula(Formula.Subformula,E.TargetState);
10:         if Result = TT then
11:            exit;
12:          end if;
13:       end if;
14:     end loop;
15:     Computations_DB.Set_Status(Formula,State,Result);
16:  end if;
17:   return Result; -- either TT or FF
18: end EvalEX;
```

We assume the existence of a Computations_DB, i.e. a global container keeping track of all partial or completed subcomputations being performed. By calling CheckComputation (line 4), each time a new subcomputation (i.e. a pair [formula, state]) is needed we check whether that computation has already been performed. In the positive case we can return the already known result, otherwise we return a NOT_YET_STARTED value. This global container is essentially a large global hashed set. The Computations_DB is not the unique global structure: also the graph representing the possible system evolutions, which is dynamically generated as the evaluation proceeds, needs to be saved and recorded as some kind of hashed map from nodes to edges (Configurations_DB). Both these global data structures are in our case programmed using custom hash tables, which allow easier monitoring and ad hoc optimisations of the code (e.g. deletions are never needed). In the previous example of EvalEX code, the call of GetEvolutions (line 7) interacts with the underlying Configurations_DB potentially triggering the analysis of a new node and the expansion of the system evolutions graph. The case of recursive operations like the *AG* formula is a little more complex especially in the case of cyclic models. In this case the evaluation has the following structure:

```
 1: function EvalAG(Formula, State) return Computation_Status is
 2:   Result: Computation_Status;
 3: begin
 4:   Result = Computations_DB.CheckComputation (Formula, State)
 5:   if Result  = NOT_YET_STARTED  then
 6:     Result := EvalFormula(Formula.Subformula, State);
 7:     if Result = TT then
 8:       Computations_DB.Set_Status(Formula,State, TT);
 9:       for E in GetEvolutions(State)  loop
10:         Result := EvalAG(Formula, E.TargetState);
11:         if Result = FF then
12:            Computations_DB._DB.Set_Status(Formula, State,FF);
13:          exit;
```

```
14:         end if;
15:       end loop;
16:      else
17:        Computations_DB._DB.Set_Status(Formula, State,FF);
18:     end if;
19:  end if;
20:  return Result;
21: end EvalAG
```

As in the previous case we first check whether a result is already known (line 4). If the result is not known we first evaluate whether the AG subformula actually holds in the current state (line 6). If it does not, the evaluation is completed and the FF value is saved and returned (lines 17, 20). If the subformula holds in the current state we save this initial partial result (line 8) and proceed with the recursive evaluation on all the successor states. If for the some successor the AG formula does not hold we save the FF status, and exit the loop. If the AG formula holds for all successors we finally return the already saved TT result. If during the evaluation of AG we meet a nested recursive evaluation of AG (because of loops in the graph) we take its already saved TT value and continue the analysis, according to the maximum fix point semantics of AG.

The above verification structure is actually a simplification of the real sequential algorithm used in the KandISTI framework. In particular we have not shown the overhead needed to save the information required to generate a counter-example (or the proof) for the formula together with the final result. Moreover, we have not shown the possibility to truncate the recursive evaluations at increasing levels of depth (bounded model checking) in order to search for counter-examples or proofs smaller than those otherwise generated by a purely depth first evaluation approach.

3 Towards a Parallel Approach

The introduction of parallelism in the verification process has to successfully overcome three difficulties.

- **Memory model**: A sequential program may usually rely on a flat view of the available memory; it is a compiler/hardware task to optimise the performance by keeping data inside registers, local caches or physical RAM in a way is completely transparent to the programmer. In the case of parallel programs the programmer has to apply Volatile/Atomic aspects to the shared objects and types, and apply the appropriate synchronisations mechanisms to serialise the accesses to the shared objects, in order to preserve a coherent inter task view of the memory. This also implies hardware/compiler efforts that may greatly reduce the overall performance of the program.
- **Synchronisations**: The executions of operations over shared data (hash tables/ queues) need to be properly synchronised, and this requires additional overhead in the execution of otherwise simple operations.

– **Parallel design**: Finally, the parallelisation of the evaluation algorithm may introduce additional complexities and overhead with respect to the plain sequential depth-first approach.

These problems do not have a known unique solution, as the most effective choice may actually depend on the details of the hardware, on the structure of the state-space, on the structure of the formula, on the desired degree of parallelism, and on the overall structure of the parallel design. Some experimentation is therefore needed. In the following we describe two possible approaches that differ in the amount of redesign they require, and in the reasonable benefits that we can expect from them.

3.1 Parallel Graph Generation

In many cases the complexity of generating the needed fragment of state-space can be higher than the complexity of evaluating a formula over it. A first experiment that could be done is therefore letting the sequential verification program to proceed in parallel with one or more other tasks that simply speedup the evolutions graph traversal and state-space generation. Since the evaluation task and the additional model-generating tasks both work upon the shared hash tables used to elaborate the nodes and to store the representation of the graph, we still need to use a thread safe implementation for these tables. Clearly it is not possible to model the whole hash table as a single protected object, but we need to synchronise the accesses to the data in a more fine-grained way. We can do that, for example, by partitioning the table and associating locks to the various regions so that accesses to different parts need not to get synchronised. In Ada, the classical abstract, portable way to implement a semaphore requires the use of a protected object; however if we choose this implementation we see that the choice has pernicious effects on the program performance (specific data is shown in Section 4). The problem lies on the high overhead introduced by protected objects for synchronising very small operations. Indeed, these may require expensive context switching each time a task not allowed to immediately proceed with its operations. An alternative solution is that of using custom, system dependent, implementations of spinlocks, exploiting the compiler built-in lock-free primitives,[2] based on atomic compare-and-swap and memory fence processor operations. Using these lightweight non context switching (busy waiting) primitives our parallel version of the program actually shows some significant speedups, especially in the case of verification of simple properties requiring the full state space analysis. The advantage of this approach is that, since we do not touch the evaluation algorithm, only rather small modifications have to be done to the model checker code. However, since the model generation is now separated and independent from the evaluation algorithm we completely lose the advantages of the on-the-fly model generation.

[2] in our case we use the GNAT run-time component System.Atomic_Primitives

3.2 Parallel Formula Evaluation

The above considerations suggested to further investigating how to introduce the parallelism directly inside the evaluation algorithm. Our scenario is that we have only a limited number of cores on a single processor, therefore it might not be necessary to push the parallelism of the evaluation to its limits. As a first approach we plan to introduce the parallelism only into the recursive AG operators. The worst-case complexity of the verification (which for the CTL-like fragment of the logic is linear w.r.t. the number of states, number of edges and size of the subformula) is only obtained when recursive operators (like AG) need to explore the whole state-space before producing a result. Therefore our approach directly targets this worst case of evaluation.

Handling of non-recursive operators In the sequential case the evaluation proceeded by constructing (in a depth first mode) a graph of subcomputations. Our goal is now to build that same structure in a concurrent way. We start by associating to each logical subcomputation, corresponding to a pair [formula, state], an evaluation fragment. All evaluation fragments are stored inside a global shared container implemented as a thread-safe hashed set. Each evaluation fragment has references to the set of fragments on which it depends, and references to the set of fragments that depend on it. In this way we have a graph structure modelling the ongoing evaluation process.

From the point of view of this data type, the main difference with respect to the sequential case is that now evaluation fragments must synchronise the operations working upon them, and that also the shared global hashed set of all fragments must synchronise all operations of insertion and retrieval. We suppose to have a pool of worker tasks, where each task performs a cycle in which the task takes one item from a shared container of needed fragments and elaborates the fragment, until the whole verification is completed. The parallel evaluation process begins with the insertion in the shared container of the initial fragment corresponding to the top-level formula and the initial system state.

The precise effect of the elaboration of a fragment associated to a pair [formula, state] depends on the kind of the formula, on the possible evolutions of the state, and on several design choices to be performed (like the amount of parallelism we want to introduce or the order of analysis of subfragments). The overall structure of the evaluation process for a fragment comprises the following steps:

- The fragment internal structure is initialised. In particular, the current system state is analysed and its next successors states (and edge labels) are computed (if not yet known). This is the point that may trigger the evaluation-driven model generation step. The eventually needed subfragments are identified, and if not already existing they are created and added to the global shared container.
- We start the check of the status of the needed subfragments. The status of a fragment can be one of: JUST_CREATED, NEEDED, TT, FF.

- If the status of a subfragment is JUST_CREATED, then a reference to the current fragment is recorded inside the subfragment parents' list. Moreover the subfragment is introduced into the global shared container of *needed fragments* (to be eventually elaborated by some worker task). The status of the subfragment is also changed to NEEDED.
- If the status of the subfragment is already NEEDED, then just the recording of a reference to the current fragment inside the parents' list of the subfragment is sufficient.
- If the status of a subfragment is TT, or FF, we take care of this information deciding if is it sufficient or not in order to complete the evaluation of the current fragment. For example, in the case of an *or* formula, an early TT result is sufficient to establish the TT status of the current fragment without any further evaluation.

If during the elaboration of a fragment we can establish its definitive TT or FF status, this information is recorded (if not already present) inside the fragment data and the same information is notified back to all the registered parents so that they can handle it.

If a fragment is intended to be elaborated in a sequential way (e.g. as in the case of fragments associated to *EX*, operators) the evaluation of subfragments proceeds as long as the previous subfragments have an already computed TT or FF value. As soon as a subfragment is found with no definitive value (i.e. a JUST_CREATED or NEEDED value), the elaboration of the current fragment is temporarily abandoned after the recording of the current fragment in the parents' list of the subfragment. When a notification from the subfragment arrives, depending on the delivered TT or FF value, either a final result is established (and notified back to the parents), or the elaboration of the remaining list of subfragments is resumed.

The operations working over fragments must behave in an atomic way, guaranteeing that the semantics of a set of parallel invocations is the same as if they were executed in some sequential order. This can be achieved by implementing fragments as Ada protected objects, or by implementing them as plain records and manually encoding the needed synchronisations of the calls using spinlocks or semaphores. We have experimented with both approaches and the results are illustrated in the Section 4.

Handling of recursive operators The above description describes well the behaviour of the evaluation for all the fragments referring to non-recursive logical operators (e.g. *and*, *or*, *not*, *EX*, *AX*,<>, [] operators). In the case of recursive parallel operators (like AG,) the situation is more complex.

Handling cycles in the state space: Let us consider a system having just three states, as shown in Fig. 1.

The evaluation of the initial fragment F1 leads to the creation of the two other fragments F2 and F3, which are evaluated in parallel. The evaluation of F2 finds the fragment F3 in the NEEDED state, and just registers itself as a parent fragment depending on F3. The same happens during the evaluation of

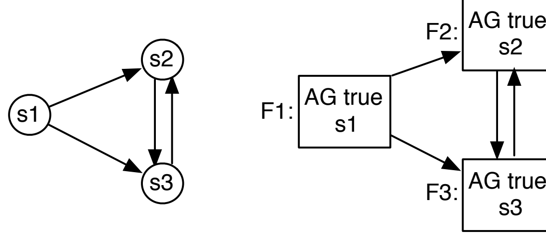

Fig. 1: Mutually dependent fragments in the evaluation of *AG true*,

F3: fragment F2 is found as already NEEDED and therefore F3 just registers itself in the F2 parents list. Now all the possible elaborations are completed (there are no more fragments to elaborate) but no result has been produced. If the evaluation of an *AG* operator completes (in the sense that there is nothing more to compute) without producing any result, then the TT result is the correct result to be returned. The situation is partly similar to the sequential case when the discovery of a still in progress subcomputation was treated as if a TT result was found. The difference is that we can discover these cases only when the full exploration of the *AG* formula is ended.

Handling early completions: The evaluation of an *AG* fragment spreads the parallel elaboration of its subfragments. If formula *subform* is eventually found to be FF in some state, then the corresponding parent *AG subform* , and all its ancestor fragments up to the root *AG* formula, must be set to the state FF. This is already done by the backward notification procedures. However we must also stop all the still ongoing parallel spreading of *AG* fragments. The specific evaluation procedure for *AG* fragments aborts its elaboration when the status of the root *AG* formula is found to be FF (another global shared variable AGSTATUS is used for this purpose), therefore stopping the further spreading of subfragments. When an *AG* fragment is aborted, its status is reset to JUST_CREATED, and the same happens recursively to all its not yet resolved ancestors. Done that, the remaining still unresolved *AG* fragments can instead be set to the TT status, as they are related to isolated loops in the graph, not affected by the nodes for which the formula is false.

Dealing with parallel root evaluations: Another limitation of the evaluation of *AG* formulas is that we cannot start two parallel evaluations from two different root states. If that happened the two evaluations could interfere. For example, it would become particularly difficult to understand which fragments to abort when an FF result if found in some state (we should not abort fragments being used also by other root *AG* evaluations). Therefore, we decide to concentrate all parallel efforts to the recursive evaluation of one *AG* formula at a time. Considering our relatively limited degree of expected hardware parallelism, this is in practice not a big limitation. While the evaluation of some *AG* formula is in progress, we enqueue all the concurrently arising evaluation requests for the same root AG formula, and when one evaluation completes we just resume another root evaluation from that queue. Notice that we do not prevent different AG formulas to be evaluated in parallel. For example, let us consider the formula:

$AG\ ((not\ EX\{b\}\ True)\ or\ AG\ EX\{a\}\ True)$ [3]

In this case we would have a parallel evaluation of the outer AG (searching for nodes which have outgoing {b} transitions), and a set of root AG evaluations looking for {a} transitions that would happen to be serialised and executed one at the time. The full implementation of the parallel version of our model checkers is still in progress. We have to support further recursive and not recursive CTL-like operators, the parameterisation of formulas, and to understand how to deal with fix points. Nevertheless we can already experiment with the current framework to evaluate the current design strategies and to optimise the implementation of the shared data types. In the next section we will show a case study that we have adopted as an initial benchmark to evaluate our progress.

4 Case Study

In order to have an early feedback about the performed design choices, we have tested our parallel approach with a case study recently analysed within our KandISTI framework. The case study is taken from the railway domain, and is related to the verification of a deadlock avoidance technique implemented inside the software which controls the movements of (driverless) trains inside a given yard [12, 13]. In that case study we had a formal model of the system based on the railway layout shown in Figure 2, including a certain number of trains moving inside the yard according with their predefined set of missions. The purpose of the verification of the formal model is to verify that the deadlock avoidance kernel of the control system is actually able to dispatch the trains on the layout without ever causing situations of deadlock, even in presence of arbitrary delays with respect to the planned timetable. Several models of different complexity can be built, depending on layout region in which the trains are moving, on the number of trains considered and on the length of their missions.

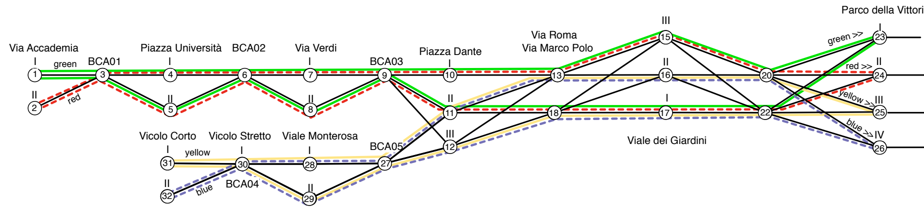

Fig. 2: The yard layout and the missions for the trains of the green, red, yellow and blue lines

The whole model has been described by a UML statechart and the verification is carried out with the UMC model checker [5]. We have considered three models of growing complexity as illustrated in Table 1.

[3] for all reachable nodes it is true that either the node has no outgoing {b} transitions, or the node and all its successors have at least one outgoing {a} transition.

Table 1: The three reference models

M1:	7 trains moving with short missions (one way traversal of the yard)	(323,195 states)
M2:	8 trains moving with short missions (one way traversal of the yard)	(1,636,538 states)
M3:	8 trains moving with long missions (two way traversal of the yard)	(8,878,643 states)

Table 2 [4] shows the evaluation time for an exhaustive deadlock analysis performed by verifying the formula *AG EX* {*true*} *True* (which simply states that all reachable states do have at least one successor), using the original purely sequential version of the tool UMC (with and without all compile time optimisations turned on).

Table 2: Sequential evaluation times for $AG\ EX\{true\}\ True$

Model	**Evaluation time**	**-O3 (optimized version)**
M1:	21 sec.	11 sec.
M2:	110 sec.	57 sec.
M3:	660 sec.	371 sec.

In Tables 3 and 4, instead, we show the evaluation times resulting from the first approach in which the sequential evaluation of the formula is helped by the parallel model generation by 1, 2 or 3 worker tasks. We show two data sets, one for the case in which data synchronisation is achieved using semaphores implemented by protected objects (Table 3), and one in which semaphores are implemented by custom (busy waiting) spinlocks (Table 4).

Table 3: Parallel graph generation (using protected objects)

Model		+1 task	+ 2 tasks	+3 tasks
M1	22 sec.	49 sec.	72 sec.	84 sec.
M1(-O3)	14 sec.	42 sec.	65 sec.	73 sec.
M2:	121 sec.	259 sec.	382 sec.	446 sec.
M2 (-O3)	75 sec.	220 sec.	349 sec.	394 sec.

The data in Table 3 shows clearly that, when protected types are used for implementing semaphores, the presence of worker tasks instead of improving the overall performance of the model checker, drastically reduces it. In particular, the introduction of a second worker has the immediate effect of more than doubling the execution time. Moreover we can see that the execution times of the parallel version with zero worker tasks (i.e. still using thread-safe global containers, but running only the sequential evaluator task), suffers only a small reduction of performance w.r.t. the original purely sequential version when no optimisation is performed on the compiled code. This small reduction is caused

[4] The data in the following tables are taken on a MacBook Pro early 2013, with a quad-core Intel Core i7-3740QM @ 2.70GHz CPU, 16GB RAM, running OS X El Capitan Version 10.11.3. The code has been compiled with GNAT GPL 2015 (20150428-49). Execution times computed by performing calls to Calendar.Time at the beginning/end of each evaluation. Sources, executables, models and test data are available from http://fmt.isti.cnr.it/WEBPAPER/AE2016-data.zip

by the presence of atomic/volatile aspects in the shared data types and by the overhead introduced by the calling of the semaphore primitives (which however never happen to block the executing task). The negative effect of the presence of atomic/volatile data is more evident when optimisations are turned on, since this aspect directly prevents many memory based optimisations.

Table 4: Parallel graph generation (using spinlocks) - times and speedups

Model		+1 task	+ 2 tasks	+3 tasks
M1	22 sec.	15 sec. *(1.46)*	13 sec. *(1.69)*	15 sec. *(1.46)*
M1 (-O3)	14 sec.	9.8 sec. *(1.42)*	9 sec. *(1.55)*	8 sec. *(1.75)*
M2:	116 sec.	79 sec. *(1.46)*	69 sec. *(1.68)*	82 sec. *(1.43)*
M2 (-O3)	72 sec.	48 sec. *(1.50)*	45 sec. *(1.60)*	50 sec. *(1.44)*
M3:	701 sec.	480 sec. *(1.46)*	408 sec. *(1.71)*	485 sec. *(1.43)*
M3 (-O3)	452 sec.	294 sec. *(1.53)*	265 sec. *(1.70)*	296 sec. *(1.52)*

With the use of spinlocks instead of protected objects we see some gains in the exploitation of the multicore architecture. As shown in Table 4, the evaluation time in this case decreases of a percentage varying from 35% to 42% when +2 worker tasks are activated. However the big gain is obtained by the activation of the first worker, the second and other workers have a much smaller effect. This effect can be explained by the fact that the more worker tasks we create, the sooner the generation of the full model completes (while the inter-tasks interferences grow): once the model is fully generated no more gains are given by the parallelism. We can see that these gain are mostly preserved also in the case of compilations with full optimisations turned on.

Finally, let us see what happens with our parallel evaluation approach, when both the model generation and the fragment evaluation are being carried out in parallel by several evaluator tasks. Also in this case we take into consideration the possibility of achieving mutual exclusion in the fragment data manipulation either using a protected type or by using custom spinlocks (Table 5). In both cases we use custom spinlocks for the synchronisation of the accesses to the shared global containers.

Table 5: Parallel formula evaluation

	using protected objects				using spinlocks			
Model	1 task	2 tasks	3 tasks	4 tasks	1 task	2 tasks	3 tasks	4 tasks
M1 (-O3)	12.7 sec.	7.4 sec.	5.9 sec.	5.4 sec.	12.5 sec.	7.2 sec.	5.4 sec.	4.5 sec.
speedup		*1.71*	*2.13*	*2.32*		*1.73*	*2.31*	*2.77*
M2 (-O3)	66 sec.	37 sec.	29 sec.	28 sec.	65 sec.	36 sec.	27 sec.	24 sec.
speedup		*1.78*	*2.27*	*2.35*		*1.8*	*2.4*	*2.7*
M3 (-O3)	437 sec.	265sec.	207 sec.	189 sec.	414 sec.	251 sec.	192 sec.	164 sec.
speedup		*1.64*	*2.15*	*2.31*		*1.64*	*2.15*	*2.5*

In this case the differences between the two implementations of thread-safe fragments are smaller, but there is still an advantage is using a handmade locking mechanism rather than a protected type implementation of these objects. This is especially true if we consider that the current locking mechanism is very

trivial (we just model full mutual exclusion between state-changing operations) and could be much improved by a more careful design. On the other side, the protected type based implementation is more abstract and less error prone, and we are happy with the choice of having selected it as our first implementation, leaving to a possible second round of optimisations the task of replacing it with something more fine-tuned with the actual optimisation needs.

The version that makes use of spinlocks for synchronising the accesses to fragment data and shared containers has been tested also on an 8-core Linux workstation. We want to observe the program behaviour when the number of available physical cores is increased. The results are shown in Table 6,[5] and a graphical comparison with the speedups reported in the right side of Table 5 is shown in Figure 3.

Table 6: Parallel formula evaluation, using spinlocks, on eight-core workstation

Model	1 task	2 tasks	3 tasks	4 tasks	5 task	6 tasks	7 tasks	8 tasks
M1 (-O3)	10.9 sec.	6.61 sec.	5.10 sec.	4.42 sec.	3.94 sec.	3.67 sec.	3.51 sec.	3.45 sec.
speedup		*1.64*	*2.13*	*2.46*	*2.76*	*2.97*	*3.10*	*3.15*
M2 (-O3)	55.1 sec.	34.2 sec.	25.9 sec.	21.9 sec.	19.7 sec.	19.1 sec.	18.4 sec.	17.9 sec.
speedup		*1.61*	*2.12*	*2.51*	*2.79*	*2.88*	*3.01*	*3.07*
M3 (-O3)	346 sec.	207 sec.	153sec.	146 sec.	131 sec.	124 sec.	120 sec.	124 sec.
speedup		*1.67*	*2.26*	*2.36*	*2.64*	*2.79*	*2.88*	*2.79*

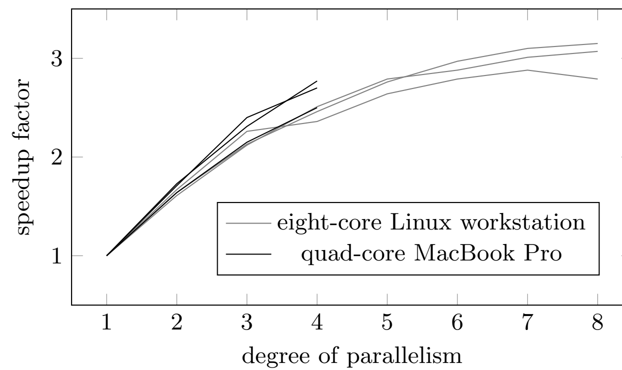

Fig. 3: Best speedups on a 4-core and 8-core processors

Overall, when the number of parallel tasks passes from 1 to 4, but even more when it passes from 4 to 8, the speedup factor grows much less than in the

[5] Test executed on a eight-core workstation with a Intel(R) Xeon(R) E5-2630 v3 @ 2.40GHz CPU, 64G RAM, running Ubuntu 14.04.4 LTS (GNULinux 3.13.0-83-generic x86_64), GNAT GPL 2015 Compiler

optimal linear scale. Surely much work can, and has to be done, in reducing the conflicts on the shared data, e.g. with more careful designs of the shared containers and of the overall parallelisation strategy.

5 Related Work

The definition of highly efficient thread-safe containers is surely one of the key factors for an efficient multicore programming. Many suggestions in this sense are available in the literature. Indeed the use of distributed / lock-free hash tables, both in the case of single processors multicore and in the case of multiprocessors with local and shared memory, has been widely studied (e.g. for parallel state space generation) and several solutions have been illustrated which allow a linear speedup w.r.t the degree of available parallelism [2, 14, 10, 3].

Also the parallelisation of model checking algorithms is a widely investigated field. Most of the studies, however, are related to the parallelisation of linear time logics (LTL), or to global model checking approaches, where the whole state space is generated and available before the beginning of the evaluation process, or to symbolic (e.g. BDD based) approaches to model checking were the state space is finite and all states have the same size. A survey of the currently adopted LTL approaches can be found in [11, 1].

In our case, instead, we want to deal with potentially infinite state spaces, where the states may have an unbounded size (e.g. because of the presence of unbounded data structures like queues). Moreover we want to deal with a CTL-like branching time logic (as initial fragment of the logic supported by our KandISTI framework). Finally, we want to preserve our explicit, local (i.e. on-the-fly), approach on which the sequential versions of our model checkers are based.

From this point of view we have several points in common with the approaches in [15, 6]. The main difference w.r.t them is that we are more focused on multicore single processor architectures while these other works are more oriented towards highly parallel/distributed architectures. It is interesting to observe that the speedup factor of about 2.5 that we obtain for our worst evaluation case when all our 4 cores are used, is obtained in [6] only on parallel architectures with at least 10 processors. Cleary the difference is that, in their case, they can reach speedup factors of 14 with 50 processors, while we are constrained to our small number of cores. However, since single processors with 16 cores are already on the market it would be interesting to see how our approach behaves on such more powerful systems. Another major difference with respect to the approach in [15] is that the logic taken into consideration by those authors is constrained to a fragment of the CTL logic where state subformulas are restricted to atomic propositions. That simplification rules out all the complexities related to existence of multiple concurrent root evaluations of recursive operators.

6 Conclusions

We have presented some early experiences gained from an ongoing process of code rewriting relative to the multicore parallelisation of a sequential model checking Ada program. In our case we have initially constrained the parallelism inside the *AG* operator (to be followed by all the other recursive operators), as this directly allows to target our worst-case execution times for finite systems. It is however reasonable (and will be the target of future studies) to introduce the parallelism in a much more widespread way. In that case the target would not much be the reduction of the worst case execution time, rather than to open the path towards a fully parallel breadth-first evaluation strategy which would allow to capture the shortest counter-examples and would allow to better deal also with potentially infinite-state systems (in presence of a finite proof or counter-example).

Computationally intensive programs making heavy use of shared data must employ highly optimised thread-safe data structures. Unfortunately this kind of abstractions are not currently available in the standard Ada container libraries, therefore we must resort to some manual implementation of them. In doing that we cannot make use of the Ada standard synchronisation mechanisms like protected types, but we need to make use of compiler-dependent, system dependent libraries providing access to low-level lock free atomic primitives. It is indeed a pity that these basic building blocks for multicore programming are not made immediately available by the language definition.

It is well known that debugging a parallel program is much harder than debugging a sequential program. However, when dealing with a shared memory architecture, just missing the volatile/atomic aspect over an object would give raise to spurious errors extremely difficult to find. The situation is worsened when the programmer is forced to use handcrafted versions of thread-safe data types and containers, and handcrafted synchronisation primitives. The result is that debugging a parallel program might easily become a nightmare. Definitely, the aid of a tool from this point of view would be extremely desirable (especially for the analysis of the volatile/atomic variables).

One of the confirmed strong points of Ada, however, is that it allows a nice, abstract design of the parallel algorithms using the language-defined, concurrency-oriented constructs. The optimisations needed for the best exploitation of the underlying hardware can be added at second round of code fine-tuning, with a relatively little effort.

In the design of a parallel system it is also quite easy to introduce logical design errors caused by unexpected thread interactions. From this point of view the construction of a formal model of the system and the exhaustive verification of the robustness the design might become an almost necessary help. Indeed this is another direction in which our project intends to move, with the goal of having a fully validated parallel evaluation engine. In spite of all difficulties and of the incompleteness of the experiment, the introduction of just a limited form of parallelism (e.g., 4 cores) allows us to obtain a performance speedup factor around 2.5, and this is surely a satisfactory initial result.

References

1. Barnat, J., Brim, L., Leucker, M.: Parallel Model Checking and the FMICS-jETI Platform. In: 12th International Conference on Engineering of Complex Computer Systems. ICECCS 2007, pp. 330-338. IEEE Computer Society (2007)
2. Barnat, J., Rockai, P.: Shared Hash Tables in Parallel Model Checking. ENTCS, Volume198, Issue 1, pp. 79-91. Elsevier (2008)
3. Barnat, J. , Rockai, P. , Still, V. , Weiser, J. : Fast, Dynamically-Sized Concurrent Hash Table. In: Model Checking of Software - 22nd International SPIN Symposium, SPIN 2015. LNCS, vol. 9232, pp. 49–65 Springer, Heidelberg (2015);
4. ter Beek, M., Gnesi., Mazzanti, F.: From EU projects to a family of model checkers - From Kandinsky to KandISTI. In: Software, Services, and Systems. LNCS, vol. 8950, pp. 312–328, Springer, Heidelberg (2015)
5. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: A state/event-based model-checking approach for the analysis of abstract system properties. Science of Computer Programming, Volume 76, Issue 2, pp. 119–135, Elsevier (2011)
6. Bollig, B., Leucker, M., Weber, M.: Local Parallel Model Checking for the Alternation-Free μ-Calculus. In: Model Checking of Software, SPIN 2002. LNCS, vol. 2318, 128–147 Springer, Heidelberg (2002)
7. De Nicola, R., Vaandrager, F. : Action versus state based logics for transition systems. In: Semantics of Systems of Concurrent Processes, LNCS, vol. 469, pp. 407–419, Springer, Heidelberg (1990)
8. Fantechi, A., Gnesi, S., Lapadula, A., Mazzanti, F., Pugliese, R., Tiezzi, F.: A logical verification methodology for service-oriented computing. ACM Transactions on Software Engineering and Methodology, Volume 21, Num. 3, ACM Press, (2012)
9. Gnesi, S., Mazzanti, F.: An Abstract, on the Fly Framework for the Verification of Service-Oriented Systems. In: Rigorous Software Engineering for Service-Oriented Systems, LNCS, vol. 6582, pp. 390–407, Springer, Heidelberg (2011)
10. Laarman A. W., van de Pol J. C., M. Weber, M.: Boosting Multi-Core Reachability Performance with Shared Hash Tables. In: Formal Methods in Computer-Aided Design, Lugano, Switzerland. IEEE Computer Society, (2010).
11. Laarman, A.: Scalable Multi-Core Model Checking. CTIT Ph.D. Thesis, Formal Methods and Tools, University of Twente, (2014) http://dx.doi.org/10.3990/1.9789036536561
12. Mazzanti, F., Spagnolo, G.O., Della Longa, S., Ferrari, A.: Deadlock avoidance in train scheduling: A model checking approach. In: Lang, F:, Flammini, F. (eds) Formal Methods for Industrial Critical Systems, FMICS 2014; LNCS, vol. 8718, pp. 109–123, Springer, Heidelberg (2014)
13. Mazzanti, F., Spagnolo, G.O., Ferrari, A.: Design of a deadlock-free train scheduler: a model checking approach. In: Julia M. Badger, J. M.0, Rozier, K. Y. (eds), NASA Formal Methods, NFM 2014. LNCS, vol. 8430, pp. 264–269, Springer, Heidelberg(2014)
14. Saad, R.T., Zilio, S. D., Berthomieu, B.: A general lock-free algorithm for parallel state space construction. In: 9th Int. Workshop on Parallel and Distributed Methods in Verification (PDMC 2010), IEEE (2010)
15. Saad, R.T., Zilio, S. D., Berthomieu, B.: An experiment on parallel model checking of a CTL fragment. In: Chakraborty, S., Mukund, M. (eds) Automated Technology for Verification and Analysis ATVA 2012. LNCS, vol. 7561, pp. 284–299, Springer, Heidelberg (2012)