PDMC 2004 Preliminary Version

Parallel and Symbolic Model Checking for Fixpoint Logic with Chop [1]

Martin Lange and Hans Wolfgang Loidl

Institut für Informatik, University of Munich, Germany
Email: {mlange,hwloidl}@tcs.ifi.lmu.de

Abstract

We consider the model checking problem for FLC, a modal fixpoint logic capable of defining non-regular properties. This paper presents a refinement of a symbolic model checker and discusses how to parallelise this algorithm. It reports on a prototype implementation of the algorithm in Glasgow Parallel Haskell (GpH) and its performance on a cluster of workstations.

Key words: Symbolic Model Checking, Functional Programming.

1 Introduction

Nowadays, model checking for (temporal) logics is commonly accepted as one of the key methods in verification. A major limitation to the usefulness of model checking for verification purposes is the state space explosion problem. To tackle this problem one employs symbolic methods [3] that work on process descriptions directly and usually achieves better performance there.

The need for efficiency together with facing huge state spaces in relevant applications often leads to the use of logics with little expressive power for model checking tasks. This bears an obvious disadvantage: what if a desired correctness property is not expressible in this logic? This justifies the search for (as efficient as possible) decision procedures for logics with higher expressive power.

A great step on the expressivity ladder regarding temporal logics was made with the introduction of FLC [9], a fixpoint logic that extends the modal μ-calculus with an operator for sequential composition. This gives FLC the power to express non-regular properties like "on every path the number of a's

[1] This work has been partially supported by the EC under the FET proactive initiative on Global Computing as project IST-2001-33149 (MRG).

so far never exceeds the number of b's" or "something holds on all paths at the same time".

In this paper we present a symbolic model checking algorithm for FLC, using BDDs, based on the sequential algorithm presented in [5] and discuss how to improve its efficiency through parallelism.

Given that — as opposed to the modal μ-calculus — the semantics of a formula is a function, this opens up new opportunities for parallel evaluations besides the obvious ones of evaluating boolean connectives simultaneously. For instance, fixpoint iterations require to test two functions for equality on a finite set of arguments which can be done in parallel.

The paper reports first runtime results, comparing a sequential and a parallel version of the symbolic model checker. We use Glasgow Parallel Haskell (GpH) as programming language for three reasons. Firstly, GpH provides an easy means of adding parallelism to an existing sequential algorithm, delegating decisions about task distribution etc to the runtime system. Secondly, using a language with functions as first-class citizens makes it easy to express the semantics of FLC formulas, which simply are functions (from sets of states to sets of states). And thirdly, Haskell's laziness provides locality for the model checking algorithm. Note that — with the semantics living in a function space — a local algorithm is now one that computes only that part of a function that is really needed. Globality in the original sense — computing all the states that have a certain property — is given by the use of BDDs.

For the basic BDD operations we use a tuned C library. We exploit the following language features of GpH: semi-explicit parallelism, for ease of expressing parallel execution; higher-order functions, for naturally expressing the model checking algorithm as a function; the foreign-function-interface (FFI) of Haskell, to incorporate Long's BDD library [8] that manages its own garbage collected heap.

1.1 Related Work

[4] presented a distributed version of a symbolic model checker for the modal μ-calculus. Parallelism is obtained through independent evaluations of sub-formulas — as we do, too — and these tasks are explicitly distributed using state space slicing. Parallelism on the evaluation of sub-formulas seems to be the only possibility to distribute the model checking task for the full modal μ-calculus, a logic involving complex fixpoint formulas.

In case of restricted alternation between fixpoints explicit graph-colouring methods have been employed that benefit from the fact that the model checking graph can be partitioned into components that can be dealt with separately. This method is used by [1] for the alternation-free fragment of the μ-calculus and by [6] for alternation depth one only. [1] also reports on a parallel version implemented in Haskell with explicit message passing. However, their explicit message passing approach to parallelisation is more verbose than our implementation in GpH. In contrast to their implementation we use a tuned C-based BDD library to gain high sequential performance, a problem reported in the pure Haskell implementation in [1].

Hence, our work differs from comparable algorithms and implementations not only by handling a much richer logic but also by using the built-in compiler support of a parallel language. Thus we do not have to administer the parallelisation task ourselves.

1.2 Organisation

The paper is structured as follows. Section 2 introduces Fixpoint Logic with Chop (FLC), focusing on properties relevant for the parallelisation of the model checking algorithm. Section 3 then describes the sequential symbolic model checking algorithm for FLC. Section 4 introduces the main concepts of Glasgow Parallel Haskell (GpH) and then discusses several parallel versions of the algorithm. Section 5 gives empirical data on running the parallel algorithm on a cluster of workstations, and summarises the results.

2 Preliminaries

Let $\mathcal{P} = \{\mathtt{tt}, \mathtt{ff}, q, \overline{q}, \ldots\}$ be a set of propositional constants that is closed under complementation, $\mathcal{V} = \{Z, Y, \ldots\}$ a set of propositional variables, and $\mathcal{A} = \{a, b, \ldots\}$ a set of action names. A *labelled transition system* is a graph $\mathcal{T} = (\mathcal{S}, \{\xrightarrow{a} \mid a \in \mathcal{A}\}, L)$ where $\mathcal{S}$ is a set of states, $\xrightarrow{a}$ for each $a \in \mathcal{A}$ is a binary relation on states and $L : \mathcal{S} \to 2^{\mathcal{P}}$ labels the states such that, for all $s \in \mathcal{S}$: $q \in L(s)$ iff $\overline{q} \notin L(s)$, $\mathtt{tt} \in L(s)$, and $\mathtt{ff} \notin L(s)$. We will use infix notation $s \xrightarrow{a} t$ for transition relations. In the context of this paper we will always assume transition systems to be finite. Thus, they can be represented using BDDs.

For a proper introduction to BDDs see for example [2]. In [3] it is shown how to use BDDs for verification purposes in general and for a variant of the modal μ-calculus due to Park and the temporal logic CTL in particular.

Given such a $\mathcal{T}$, w.l.o.g. assume that $|\mathcal{S}| \leq 2^n$ for some $n \in \mathbb{N}$. Then every state has a unique number between 0 and $2^n - 1$ and one can identify a state s with its binary representation $s_1 \ldots s_n$, $s_i \in \mathbb{B}$.

$\mathcal{T}$ can be represented by $|\mathcal{A}| + |\mathcal{P}|$ many BDDs over $2 \cdot n$ variables. We fix their names and a total order as $x_1 < y_1 < \ldots < x_n < y_n$.

- For every proposition $q \in \mathcal{P}$ we have a BDD t_q over the variables $x_1, \ldots, x_n$ s.t. $t_q(x_1, \ldots, x_n) = \mathtt{tt}$ iff $q \in L(x_1 \ldots x_n)$.
- For every action $a \in \mathcal{A}$ we have a BDD t_a over the variables $x_1, \ldots, x_n$, $y_1, \ldots, y_n$ s.t. $t_a(x_1, y_1, \ldots, x_n, y_n) = \mathtt{tt}$ iff $x_1 \ldots x_n \xrightarrow{a} y_1 \ldots y_n$.

Formulas of FLC are given by the following grammar.

$$\varphi \quad ::= \quad q \mid Z \mid \tau \mid \langle a \rangle \mid [a] \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mu Z.\varphi \mid \nu Z.\varphi \mid \varphi ; \varphi$$

where $q \in \mathcal{P}$, $Z \in \mathcal{V}$, and $a \in \mathcal{A}$. We will write σ for μ or ν. To save brackets we introduce the convention that ; binds stronger than $\wedge$ which binds stronger than $\vee$. Formulas are assumed to be well-named in the sense that each binder variable is distinct. Our main interest is with formulas that do not have free variables, in which case there is a function $\mathit{fp}() : \mathcal{V} \to \mathrm{FLC}$ that maps each variable to its defining fixpoint formula.

The set $\mathit{Sub}(\varphi)$ of sub-formulas of φ is defined as usual, with $\mathit{Sub}(\sigma Z.\psi) = \{\sigma Z.\psi\} \cup \mathit{Sub}(\psi)$.

The *tail* of the rightmost free occurrence of a variable X in ψ, $\mathit{tl}_X(\psi)$ is – intuitively – everything that occurs behind this X in ψ. Technically, it is defined inductively with

$$\mathit{tl}_X(\varphi;\psi) \;:=\; \begin{cases} \mathit{tl}_X(\psi) & \text{if } X \text{ occurs in } \psi \\ \{\varphi';\psi \mid \varphi' \in \mathit{tl}_X(\varphi)\} & \text{o.w.} \end{cases}$$

and the other cases straight-forward: $\mathit{tl}_X(\varphi \wedge \psi) := \mathit{tl}_X(\varphi) \cup \mathit{tl}_X(\psi)$, etc. The set of *tails* of a variable X, $\mathit{tl}(X)$ is the union over all $\mathit{tl}_X(X;\varphi)$ where $\mathit{fp}(X) = \sigma X.\varphi$ and all rightmost occurrences are successively replaced by $\mathtt{tt}$ if $\sigma = \nu$ and $\mathtt{ff}$ otherwise.

An *environment* $\rho : \mathcal{V} \to (2^{\mathcal{S}} \to 2^{\mathcal{S}})$ maps variables to monotone functions of sets to sets. $\rho[Z \mapsto f]$ is the function that maps Z to f and agrees with ρ on all other arguments. The semantics $[\![\cdot]\!]^{\mathcal{T}} : 2^{\mathcal{S}} \to 2^{\mathcal{S}}$ of an FLC formula, relative to $\mathcal{T}$ and ρ, is a monotone function on subsets of states with respect to the inclusion ordering on $2^{\mathcal{S}}$. These functions together with the partial order given by

$$f \sqsubseteq g \text{ iff } \forall X \subseteq \mathcal{S} : f(X) \subseteq g(X)$$

form a complete lattice with joins $\sqcup$ and meets $\sqcap$. By the Tarski-Knaster Theorem the least and greatest fixpoints of functionals $F : (2^{\mathcal{S}} \to 2^{\mathcal{S}}) \to (2^{\mathcal{S}} \to 2^{\mathcal{S}})$ exist. They are used to interpret fixpoint formulas of FLC.

To simplify the notation we assume a transition system $\mathcal{T}$ to be fixed, and write $[\![\cdot]\!]_\rho$ instead of $[\![\cdot]\!]^{\mathcal{T}}_\rho$.

$$\begin{array}{rcl@{\qquad}rcl}
[\![q]\!]_\rho &:=& \lambda X.\{s \in \mathcal{S} \mid q \in L(s)\} & [\![Z]\!]_\rho &:=& \rho(Z) \\
[\![\tau]\!]_\rho &:=& \lambda X.X & [\![\varphi;\psi]\!]_\rho &:=& [\![\varphi]\!]_\rho \circ [\![\psi]\!]_\rho \\
[\![\varphi \vee \psi]\!]_\rho &:=& [\![\varphi]\!]_\rho \sqcup [\![\psi]\!]_\rho \\
[\![\langle a\rangle]\!]_\rho &:=& \lambda X.\{s \in \mathcal{S} \mid \exists t \in X, \text{ s.t. } s \xrightarrow{a} t\} \\
[\![[a]]\!]_\rho &:=& \lambda X.\{s \in \mathcal{S} \mid \forall t \in \mathcal{S}, s \xrightarrow{a} t \Rightarrow t \in X\} \\
[\![\mu Z.\varphi]\!]_\rho &:=& \bigsqcap \{f : 2^{\mathcal{S}} \to 2^{\mathcal{S}} \mid f \text{ monotone, } [\![\varphi]\!]_{\rho[Z \mapsto f]} \sqsubseteq f\}
\end{array}$$

with the cases of $\varphi \wedge \psi$ and $\nu X.\varphi$ being similar. A state s satisfies a formula φ under ρ, written $s \models_\rho \varphi$, iff $s \in [\![\varphi]\!](\mathcal{S})$. If φ is a closed formula then ρ can be omitted and we write $[\![\varphi]\!](\mathcal{S})$ as well as $s \models \varphi$. We will give a few examples of FLC formulas.

Example 2.1 a) Let $\mathcal{A} = \{a, b, c, d\}$ and

$$\begin{aligned}\varphi_1 \;:=\; & \nu Y.[b];\mathtt{ff} \wedge ([c] \wedge [d]);Y \wedge \\ & [a];(\nu Z.[b] \wedge ([c] \wedge [d]);Z \wedge [a];Z;Z);(([a];\mathtt{ff} \wedge [b];\mathtt{ff}) \vee Y)\end{aligned}$$

Formula φ_1 expresses "the number of b's never exceeds the number of a's" which is non-regular and, therefore, is not expressible in the modal μ-calculus.

The sub-formula $\psi = \nu Z.[b] \wedge ([c] \wedge [d]);Z \wedge [a];Z;Z$ expresses "there can be at most one b more than there are a's". This can be understood best by unfolding the fixpoint formula and thus obtaining sequences of modalities and variables. It is easy to see that replacing a Z with a $[b]$ reduces the number of Z's whereas replacing it with the other conjunct adds a new Z to the sequence.

Then, $[b];\mathtt{ff} \wedge [a];\psi$ postulates that at the beginning no b is possible and for every sequence of n a-actions there can be at most n b-actions. Finally, the Y in φ_1 allows such sequences to be composed or finished in a deadlock state.

b) Let $\varphi_3 := \nu X.\tau \wedge [a];X;\langle b\rangle \wedge (\mu Y.[a];Y;[b] \vee [b];Y;[a] \vee \tau);X$. It is best understood by thinking of it as a context-free grammar with intersections. Then the inner least fixpoint formula generates sequences of an even number of $[\cdot]$ formulas before an X in which there are as many a's as b's. Together with the other conjuncts, unfolding X creates arbitrary formulas of the form $[c_1];\ldots;[c_n];\langle b\rangle;\ldots;\langle b\rangle$ where $\langle b\rangle$ occurs m times with $n \geq m$, $c_i \in \{a, b\}$ for all i and $m = |\{i \mid c_i = a\}| - |\{i \mid c_i = b\}|$. Thus, φ_3 says that after k a-actions and l b-actions it is always possible to do another $k - l$ b-actions.

3 The Sequential Algorithm

A BDD-based and sequential model checking algorithm for FLC was presented in [5]. However, the version given there is not correct. The semantics of an FLC formula is an element of a lattice of monotone functions. Fixpoints in this lattice are computed using the usual fixpoint iteration method. In order to decide whether or not a fixpoint is found, one has to compare functions on several (and possibly exponentially many) arguments rather than one only. Consequently, the symbolic model checking algorithm is more expensive than previously expected, and improving its performance becomes an even more important issue.

The algorithm is local in the following sense. However, in order to determine whether or not $s \models \varphi$ holds for given s and φ, it is not necessary to compute the entire semantics $[\![\varphi]\!]$. Instead it suffices to calculate it pointwise starting with $[\![\varphi]\!](\mathcal{S})$ for the underlying set of states $\mathcal{S}$, etc. Compare this to the usual idea of a local model checking algorithm which does not calculate the set of all states satisfying a given formula.

The algorithm MC shown in Figure 1 takes as arguments an environment ρ and an FLC formula φ and returns $[\![\varphi]\!]_\rho$. Thus, it is global in the usual

```
MC($\rho$, $\varphi$) = case $\varphi$ of
    $q$               $\rightarrow$  $\lambda t.t_q$
    $\tau$            $\rightarrow$  $\lambda t.t$
    $X$               $\rightarrow$  $\rho(X)$
    $\langle a\rangle$  $\rightarrow$  $\lambda t.\exists y_1.\dots.\exists y_n.t_a \wedge t[y_1/x_1,\dots,y_n/x_n]$
    $[a]$             $\rightarrow$  $\lambda t.\forall y_1.\dots.\forall y_n.t_a \rightarrow t[y_1/x_1,\dots,y_n/x_n]$
    $\psi_0 \vee \psi_1$  $\rightarrow$  $\lambda t.MC(\rho,\psi_0)(t) \vee MC(\rho,\psi_1)(t)$
    $\psi_0 \wedge \psi_1$  $\rightarrow$  $\lambda t.MC(\rho,\psi_0)(t) \wedge MC(\rho,\psi_1)(t)$
    $\psi_0;\psi_1$   $\rightarrow$  $\lambda t.MC(\rho,\psi_0)(MC(\rho,\psi_1)(t))$
    $\sigma X.\psi$   $\rightarrow$  let $X^0 = \lambda t.$ if $\sigma = \mu$ then ff else tt in
                      let $\mathcal{F} = \lambda f.MC(\rho[X \mapsto f],\psi)$ in
                      let $T = tc_{\psi,X,\rho}(t)$ in
                      fixPoint($T$, $\mathcal{F}$, $X^0$)
fixPoint($T$, $\mathcal{F}$, $g$) =
  $\lambda t.$ let $f = \mathcal{F}(g)$ in
      if $\forall t' \in T : f(t') = g(t')$ then $f(t)$ else fixPoint($T$, $\mathcal{F}$, $f$)($t$)
```

Fig. 1. The sequential and symbolic model checking algorithm for FLC.

sense. An evaluation strategy in a functional language guarantees that only those parts of $[\![\varphi]\!]_\rho$ are computed that are needed in order to establish or refute $s \models \varphi$.

Algorithm MC calls the function fixPoint which computes the fixpoint iterations until stability is reached. Due to our type of locality, functions are not evaluated on every argument. This means that the fixpoint iteration can become stable on a given argument although the fixpoint in the function space is not reached yet. Thus, the function fixPoint has to evaluate the fixpoint iteration on several arguments. It approximates the necessary ones (i.e. BDDs representing sets of states) using the function $tc_{\psi,X}(t)$ which takes an FLC formula ψ, a free variable X, an environment ρ and a BDD t, and first computes the reflexive and transitive closure of $\{t\}$ under the functions $\{[\![\varphi]\!]_\rho \mid \varphi \in tl(X)\}$. This set includes the set of all BDDs that are needed as arguments in order to compute the fixpoint in the function space.

4 Parallelising Algorithm MC

*4.1 Glasgow Parallel Haskell (*GpH*)*

GpH [11] is a modest conservative extension of the non-strict, purely functional programming language Haskell98 [10], adding a constructor `par` for parallel composition (the sequential `seq` combinator is already part of Haskell98): ``x `par` e`` expresses potential parallel execution of `x` and `e`. Typically `x` is a variable occurring in `e`, but it can be a general Haskell expression. The result of the expression is that of `e`, thus semantically `par` is a projection on its second argument. Operationally, this expression adds `x` to a pool of potentially parallel executions. The decision whether to indeed execute `x` in parallel, and if so, where to execute it, is made automatically by the runtime-system. In GpH the programmer uses the `par` combinator to annotate expressions as being potentially parallel and leaves the management of the parallelism to the runtime system. We call this a model of semi-explicit parallelism.

The implementation of GpH models a virtual shared heap, i.e. all program variables are accessible as if residing on the same processor. The runtime system arranges for automatic data transfer between processors if this is not the case. A virtual shared heap facilitates the development of architecture independent programs that are able to exploit large numbers of processors. In contrast, the number of processors in physical shared memory machines is bounded by hardware constraints, usually to a few dozens. For applications wanting to exploit only small amounts of parallelism such a simpler shared memory could be used. An experimental version of the runtime system for physical shared memory machines exists, but currently suffers from high overhead during normal execution, because of frequent locking that becomes necessary when allocating new data structures. Therefore, we ultimately want to integrate shared memory machines into our model by optimising the communication routines for the physical shared memory case where possible, but retain an overall virtual shared memory model even for these machines.

One important language feature for this application is the FFI interface provided in Haskell. This enables us to use existing, tuned C-code for basic BDD operations. However, since the Long BDD library [8] that we use, manages its own heap, interaction between the Haskell heap and the C heap is necessary. On Haskell side this means that a BDD is represented as a Foreign Object, i.e. a data-structure that is constructed outside the Haskell heap. This data structure is represented by a pointer into the C heap and a finaliser routine that is executed once it is not used from the Haskell heap anymore.

In order to provide a higher level of abstraction we usually use evaluation strategies [11] to express parallelism in more complex Haskell programs. These strategies are polymorphic, higher-order functions that express parallelism, evaluation order and evaluation degree. Strategies are formulated in terms of `par` and `seq` and are attached to program expressions by the `using` combinator. For example, parallel evaluation of all elements in a list `xs`, evaluating each list element to normal form, can be written as ``xs `using` parList rnf``. The predefined strategy `parList` can be written as `foldr par ()`. In this paper we will only use the overloaded strategy `rnf`, which evaluates a data structure to normal form.

4.2 The Parallel Algorithm

Our first parallel version of algorithm MC, $\text{PMC}_1(n)$ where n denotes the number of processors used in its execution, exploits parallelism over the constructs $\varphi \wedge \psi$ or $\varphi \vee \psi$. No dependencies exist between the executions of both branches, yielding a rich source of massive parallelism. The structure of this parallel algorithm is divide-and-conquer. But sequential profiling in Section 5.1 shows that unrestricted usage of parallelism over these constructs will generate far too much parallelism to be efficient. Therefore, we combine this version with a thresholding mechanism often used for divide-and-conquer parallelism: in the tree of formulas that are generated by the model checking algorithm, parallelism is only generated up to a certain depth, or threshold. We provide the threshold as an additional parameter to the algorithm.

While this parallelism can be described with only one `par` combinator, we also need to specify the evaluation order, using a `seq` combinator, and evaluation degree, using the reduce to normal form evaluation strategy `rnf`. Additionally, the system requires all data, that is input to a parallel thread, to exist in the Haskell heap. Otherwise only a pointer into the C heap would be transmitted to another processor, where this pointer would, of course, be invalid. Therefore, we have to introduce functions `freezeBDD` and `thawBDD`, which copy a BDD from the C to the Haskell heap and vice versa. The function `fMC` is a variant of MC that works over such "frozen" BDDs.

The parallel code for the $\wedge$-branch in the algorithm looks like this, describing evaluation of ψ in parallel to the main thread that evaluates $\varphi \wedge \psi$:

```
And f g -> \ t ->
  let t'       = freezeBDD t
      (x', y') = (fMC f e t', fMC g e t')
      (x, y)   = (thawBDD x', thawBDD y')
  in (rnf t') `seq` ((rnf y') `par` ((rnf x') `seq` (x `bAND` y)))
```

where `bAND` combines two BDDs with the boolean $\wedge$.

The computation of the test set T with the help of $tc_{\psi,X,\rho}$ requires calls to the model checker MC (on a smaller formula) again. This is because the tails of X in ψ need to be transformed into functions from BDDs to BDDs in order to compute the transitive closure of t under these functions. This gives rise to further evaluations of boolean connectives. Furthermore, the computation of the transitive closure over a set of functions could be parallelised.

Algorithm PMC_2 results from MC by computing the test set in parallel to the fixpoint calculation itself. This requires a change in the innermost `let` of the σ case in Figure 1:

```
let t'      = freezeBDD t
    xs'     = ((map freezeBDD) . setToList . tests . thawBDD) t'
    testset = (mkSet . map thawBDD) xs'
in
(rnf t') `seq` (rnf xs') `par` fixpoint n f a testset t
```

Again, additional code for freezing and thawing the input and output of potentially parallel computations is needed. Parallelism is generated by the `rnf xs'` clause, which performs a computation of the test set by calling `tests` (corresponding to *tc* in Figure 1). Note, that the thread computing `xs'` will run in parallel with the main thread computing `fixpoint n f a testset t`. Synchronisation will be needed when the latter requires data structures produced by the former.

The main advantage of this source of parallelism is its coarse granularity, turning the entire test set generation into a parallel thread. The form of parallelism exploited here is producer-consumer (or pipeline) parallelism, where producer and consumer work in parallel on the same data structure. This parallelism could be combined with the massive amount of divide-and-conquer parallelism in version PMC_1. Additionally, we could also compute the test set itself in parallel. This is possible since there are finitely many functions under which the closure of a set of BDDs needs to be computed. Not only can the single functions be applied in parallel, it is also possible to parallelise a single iteration with a single function since it needs to be applied to several BDDs.

5 Empirical Test Data

In this section we describe a family of transition systems that model bounded stacks or counters. Let $\mathcal{P} = \{\mathtt{tt}, \mathtt{ff}\}$ and $\mathcal{A} = \{push, pop, reset, idle\}$. An N-bit-counter is a $\mathcal{T} = (\mathcal{S}, \{\xrightarrow{a} \mid a \in \mathcal{A}\}, L)$ with $\mathcal{S} = \{0, \dots, 2^N - 1\}$ and for all $s, t \in \mathcal{S}$: $s \xrightarrow{push} t$ iff $t = s + 1$, $s \xrightarrow{pop} t$ iff $t = s - 1$, $s \xrightarrow{reset} t$ iff $t = 0$, $s \xrightarrow{idle} t$ iff $s = t$. Note that $\mathcal{T}$ does not count modulo 2^N. Instead, value 0 cannot be decreased and $2^N - 1$ cannot be increased. In other words, an N-bit-counter can be increased and decreased between the values 0 and $2^N - 1$, reset to 0 and do nothing. According to Section 2, an N-bit-counter can be represented by 4 BDDs over $2 \cdot N$ variables.

Algorithms MC, $\mathrm{PMC}_1(1)$ and $\mathrm{PMC}_1(4)$ have been run on N-bit-counters for some values of N in its sequential and parallel versions. The following formulas have been used as inputs: φ_1 and φ_3 from Example 2.1 with $a := push$, $b := pop$, $c := reset$, $d := idle$, as well as $\varphi_2 := \nu X.\tau \wedge X; \langle push \rangle$ demanding *push*-paths of unbounded length and the (μ-calculus definable) formula $\varphi_4 := \mu X.[push]; X$. The running times (in seconds) in relation to the size 2^N of an N-bit-counter are given in Figure 2.

All measurements have been run on a cluster of SuSE 9.0 PCs with 1.6GHz AMD Athlon processors, 256kB cache, 512MB of DDR SDRAM memory and a 9.5GB local harddisk. The network is a 100MB/s Fast Ethernet with latency of 120μs measured under PVM 3.4.3. We use GHC 5.02.3 for compilation, and our parallel extension of its runtime system, GUM.

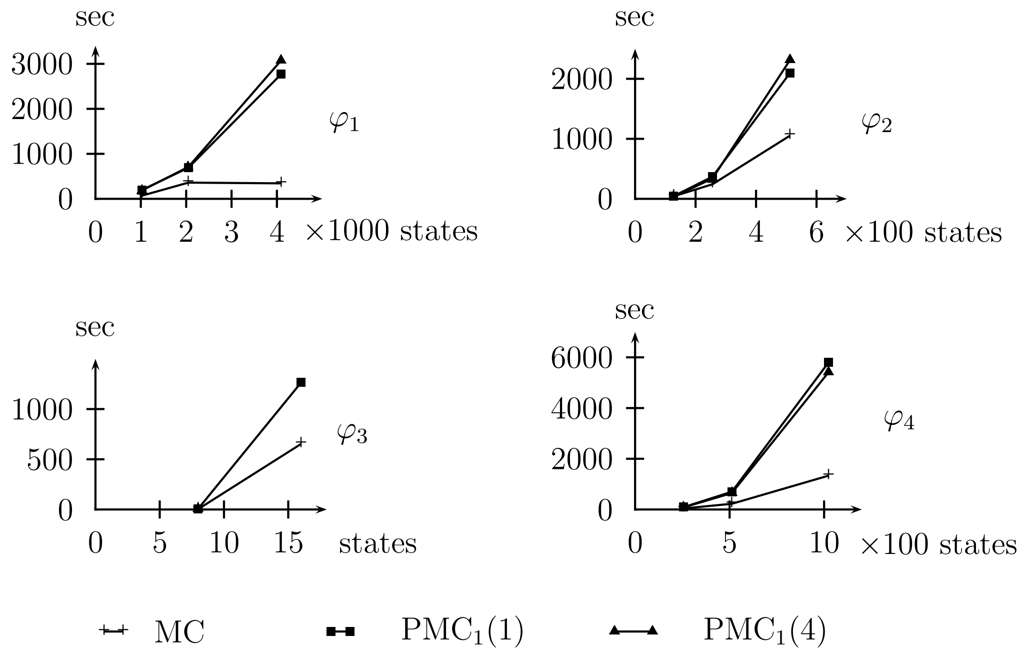

Fig. 2. The sequential and parallel running times.

5.1 Sequential Profiling

Following our methodology for developing parallel programs [7], we start with profiling the sequential code to identify the most time consuming components. We use cost centre profiling to classify time and heap consumption by program points responsible for the computation.

In analysing the profiling results for φ_3 with $N = 4$, the most striking result is the high overhead for the finaliser routines of the Foreign Objects: 47% of heap and 23% of time. The model checking algorithm generates a large number of Foreign Objects: circa 18 million through calls to $\exists$ and $\forall$, 17 million through $\vee$ operations, and 116 million through BDD $\wedge$ operations. Sharing the finaliser routines for a class of Foreign Objects, those representing BDDs, would improve sequential performance in this case. While the direct time consumption of the $\vee$ and $\wedge$ branches is fairly small, 1.3% and 8.3%, those cases initiate much larger computations in the $[\_]$ and $\langle\_\rangle$ branches, in total 25.9%, and indirectly through the fixpoint computation. The overall heap residency remains low throughout the computation, never exceeding 400kB. In summary, while the BDD operations are rather cheap, they are called very often, thus contributing to the overall costs. This is a typical characteristic for symbolic applications [7] with irregular parallelism and represents a good match with the dynamic management of parallelism performed in GpH.

5.2 Parallel Runtimes

Our performance results show that version PMC_1 produces extremely fine grained parallelism, i.e. the generated threads have only a small amount of work to do. The parallel version contains overhead for freezing and thawing BDDs and for generating and managing a large number of threads. Therefore, the naive and/or parallel version does not achieve any speedup. When combining the and/or parallelism with thresholding the total number of threads that are generated drops significantly: for φ_2 and $N = 7$ from 468 down to 408. Running this version with input φ_2 on 4 processors we obtain relative speedups of 1.28 for $N = 7$ and 1.09 for $N = 8$. For this formula the overhead of the 1 processor version, compared to a sequential version, stays within a factor of 2, but for other formulas it increases considerably. In particular, versions producing more potential parallelism perform much more data transfer between heaps. For the other formulas we achieve relative speedups of up to 1.07 for φ_1, 1.13 for φ_3 and 1.12 for φ_4. When keeping the threshold fixed, the speedups drop for larger inputs.

For an only mildly tuned parallel algorithm with highly irregular parallelism such low speedups are unsurprising. Clearly the algorithm does not suffer from a shortage of parallelism. Our current work focuses on granularity control in the algorithm, and using thresholding is one step in that direction. However, the choice of the best threshold will in general depend on the input formula, and we are currently exploring alternatives in automatically choosing this threshold. The strong dependence of performance on thread granularity is underlined by the poorer speedup for formula φ_1 and $N = 11$: 0.98 (a slow-down). For this input 2062 threads are generated in total, far more than for φ_2. Performance improves again when reducing the threshold and thus limiting the amount of parallelism as well as the overhead attached to it.

One important goal in our design of the parallel algorithm is to achieve scalability, i.e. to ensure that a larger number of processors than those currently used can be exploited without changes to the code. Unfortunately, this property also triggers more data transfer between heaps than necessary. An implementation that uses laziness to avoid these operations unless parallelism is definitely exploited should improve the one processor performance, but is tricky to implement. Hardwiring such explicit order of evaluation into the code is the main reason for its complexity in an otherwise simple model of parallelism. We therefore now work on an approach where the structure of a Foreign Object is extended with a marshalling function, that will be automatically started as part of the graph packing algorithm used by the runtime-system to transfer computations between processors. This ensures that data transfer between heaps is only done when needed for parallel execution, and eliminates this complexity from the parallel GpH code.

Whereas version PMC_2 avoids the problem of too fine-grained parallelism, it only creates one parallel thread on top of the main thread of execution.

The code discussed in Section 4 generates producer-consumer parallelism, exploiting the non-strict semantics of GpH: the consumer can start working on the intermediate data structure (the test set) even before it has been fully generated. Our measurements of this version, however, show that the data dependency between test set generation and fixpoint iteration are too tight to achieve a useful amount of parallelism: most of the time one of the two threads is blocked, waiting for remote data to arrive. For example with φ_2 and $N = 7$, during 96% of the time only one thread is running, with the other thread blocked on data that has not been produced, yet, leaving virtually no parallelism in overlapping these two computations. As a result PMC_2 does not achieve any speed-up at all: for φ_1 and $N = 10$ the speedup is 0.98, for φ_2 and $N = 7$ the speedup is 0.99. For more complex input formulas it would be possible to compute the test set for a sub-formula of the form $\sigma X.\psi$ at the very beginning, before reaching the quantifier in the formula. In this case, the test set can be computed in parallel with model checking the other parts of the formula, and all test sets can be computed in parallel, too. This, however, may create work that turns out to be unnecessary, and the effectiveness of this version very much depends on the structure of the input formula.

5.3 Summary

The presented symbolic model checker generates highly irregular parallelism in the form of deep, unbalanced divide-and-conquer trees, that need thresholding to limit the total amount of parallelism generated by and/or parallelism over the input formulas. The achieved speedups are small: in the best case, φ_2 and $N = 7$, we can report an absolute speedup of 1.2 on 4 processors, in the worst case we get considerable slow-down. We identified overheads of converting C into Haskell data structures and too fine thread granularity as the main limitations. To improve performance, we need a method for better controlling thread granularity. Using the size of the test set to guide the decision whether or not to generate parallelism in one subtree of the computation hierarchy would be promising. The parallel computation of the test set itself did not prove worthwhile, since the data-dependencies between the producer of the test set, and the consumer are too tight to allow for a significant amount of parallel execution between these two threads.

In more general terms our experience of using Haskell as the top-level language for the sequential and the parallel algorithms has been a positive one. The algorithm itself is much closer to the high-level presentation given here, compared to an earlier Java-based implementation [5]. Using a tuned BDD library gives us high performance in the sequential case already, exploiting Haskell's foreign function interface (FFI). GpH as parallel language allows rapid prototyping of different parallel versions, to a degree not possible in conventional parallel languages. However, using an external BDD library, we had to write explicit marshalling code to convert BDDs in the C heap into Haskell data structures. At the moment data marshalling has to be explicitly controlled in the GpH code adding to the complexity of the parallel algorithm. In the future we plan to integrate data marshalling into the automatic graph packing algorithm that is used to transfer computations. For writing the marshalling code, an additional layer on top of the FFI, similar to the GreenCard tool, would be very useful.

References

[1] B. Bollig, M. Leucker, and M. Weber. Local parallel model checking for the alternation-free μ-calculus. In S. Leue D. Bonaki, editor, *Proc. 9th Int. SPIN Workshop on Model checking of Software, SPIN'02*, volume 2318 of *LNCS*, pages 128–147. Springer, 2002.

[2] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[3] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[4] O. Grumberg, T. Heyman, and A. Schuster. Distributed symbolic model checking for μ-calculus. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. 13th Conf. on Computer-Aided Verification, CAV'01*, volume 2102 of *LNCS*, pages 350–362. Springer, July 2001.

[5] M. Lange. Symbolic model checking of non-regular properties. In R. Alur and D. Peled, editors, *Proc. 16th Conf. on Computer Aided Verification, CAV'04*, volume 3114 of *LNCS*, pages 83–95, Boston, MA, USA, August 2004. Springer.

[6] M. Leucker, R. Somla, and M. Weber. Parallel model checking for LTL, CTL* and L^2_μ. In L. Brim and O. Grumberg, editors, *2nd Int. Workshop on Parallel and Distributed Model Checking, PDMC'03*, volume 89 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.

[7] H-W. Loidl, P.W. Trinder, K. Hammond, S.B. Junaidu, R.G. Morgan, and S.L. Peyton Jones. Engineering Parallel Symbolic Programs in GPH. *Concurrency — Practice and Experience*, 11:701–752, 1999.

[8] D. Long. The Long BDD Library. Web page, May 2004. <`http://www-2.cs.cmu.edu/~modelcheck/bdd.html`>.

[9] M. Müller-Olm. A modal fixpoint logic with chop. In C. Meinel and S. Tison, editors, *Proc. 16th Symp. on Theoretical Aspects of Computer Science, STACS'99*, volume 1563 of *LNCS*, pages 510–520, Trier, Germany, 1999. Springer.

[10] S.L. Peyton Jones, J. Hughes *et al.* Haskell 98: A Non-strict, Purely Functional Language, 1999. Available at `http://www.haskell.org/`.

[11] P.W. Trinder, K. Hammond, H-W. Loidl, and S.L. Peyton Jones. Algorithm + Strategy = Parallelism. *J. of Functional Programming*, 8(1):23–60, Jan. 1998.