Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 220 (2008) 19–34
www.elsevier.com/locate/entcs

A Multi-Core Solver for Parity Games

Jaco van de Pol and Michael Weber[1]

Formal Methods and Tools, University of Twente, The Netherlands
{vdpol,michaelw}@cs.utwente.nl

Abstract

We describe a parallel algorithm for solving parity games, with applications in, e.g., modal μ-calculus model checking with arbitrary alternations, and (branching) bisimulation checking. The algorithm is based on Jurdzinski's Small Progress Measures. Actually, this is a class of algorithms, depending on a selection heuristics. Our algorithm operates lock-free, and mostly wait-free (except for infrequent termination detection), and thus allows maximum parallelism. Additionally, we conserve memory by avoiding storage of predecessor edges for the parity graph through strictly forward-looking heuristics. We evaluate our multi-core implementation's behaviour on parity games obtained from μ-calculus model checking problems for a set of communication protocols, randomly generated problem instances, and parametric problem instances from the literature.

Keywords: parity games, boolean equation systems, model checking, multi-core algorithm, μ-calculus

1 Introduction

In this paper we propose and evaluate a (multi-core) parallel algorithm to solve two-player parity games [27]. These games are played on a game graph, whose vertices are partitioned in those where player Even and those where player Odd moves, so they are turn-based. A play is an infinite sequence of moves. Every vertex is associated with a priority. Player Even wins a play if the least priority that occurs infinitely often is even. The problem of solving a game, is to determine from which vertices player Even has a (memory-less) strategy that always wins.

Solving parity games is equivalent to model checking for μ-calculus [28,9]. Also, there is a direct correspondence to (simple) Boolean Equation Systems (BES) [31]. Important applications of parity game solvers are verifying that a software design meets its requirements, equivalence checking of system descriptions, and others.

For fragments of the μ-calculus, such as LTL, CTL, the alternation-free fragment [12], and the purely disjunctive fragment [16], very efficient algorithms are

[1] This research has been partially funded by the Netherlands Organisation for Scientific Research (NWO) under FOCUS/BRICKS grant number 642.000.05N09

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2008.11.011

known. However, the μ-calculus itself is strictly more expressive than these fragments (cf. [9]). For instance, all kinds of fairness conditions, and many other specification patterns, can be directly formulated in μ-calculus. By focusing on two-player parity games, we aim at algorithms for the μ-calculus in its full generality.

It is known that solving parity games is in $\mathsf{NP} \cap \mathsf{co\text{-}NP}$ (and $\mathsf{UP} \cap \mathsf{co\text{-}UP}$) [25], basically because memory-less strategies are sufficient, and such strategy can be checked in polynomial time; by the theorem of determinacy [15] the winning set of player Odd is exactly the complement of the winning set of player Even. However, the question if there exists a polynomial algorithm is still open despite much research effort, which makes this one of the most exciting problems in traditional complexity theory. The complexity depends on the number of vertices (n) and edges (m), but also many algorithms have a crucial dependency on d, the number of priorities in the game, or the number of alternations of μ and ν-operators (least/greatest fix point operators).

For small d, the best known algorithm is based on small progress measures by Jurdzinski [25]. It has time complexity approximately $\mathcal{O}(m \times n^{\lfloor d/2 \rfloor})$ and runs in $\mathcal{O}(d \times n)$ space.[2] Similar complexities had already been derived for μ-calculus (cf. [9]) and Boolean Equation Systems [34]. When d is large, the sub-exponential algorithm of [26] is better, with worst case time complexity $n^{\mathcal{O}(\sqrt{n})}$. It is worth to mention here also the *strategy improvement* algorithm [36], whose exact worst case time complexity is not yet known, and might even be polynomial.

1.1 Contributions

Besides these theoretical observations, a practical problem is that model checking often leads to extremely large game graphs. In particular, the size of the game graph is potentially the product of the size of the system's model (itself a product of the size of its components) and the size of the formula to check. This means that n easily hits the 10^{10} bound and beyond. Several techniques to alleviate this exist, like automatic abstraction, on-the-fly generation, BDD representation and partial-order reduction. Another alternative is to use larger computer equipment, such as clusters or GRIDs of workstations, or multi-core machines with big memories and many CPUs. Of course one hopes that this solution not only alleviates the memory problem, but also yields time gains, by using parallel algorithms.

Our contribution is a multi-core version of Jurdzinski's algorithm based on small progress measures. This algorithm assigns vectors of numbers to vertices in the game graph. These vectors are lifted in a particular way, until a fix point is reached. Because many vertices can be lifted at the same time, we expect that this algorithm can be easily parallelized, and perhaps distributed. However, lifting a vertex typically enables lifting some of its predecessors, so the vertices cannot be treated completely independent. The key to implement Jurdzinski's algorithm is to develop a heuristics for choosing the next vertex to lift.

[2] We are aware of two recent improvements: Schewe reports an $\mathcal{O}(m \times n^{d/3})$ algorithm [33], and [14] reports an acceleration for the 3-priorities-case.

We aimed at an efficient algorithm that is well-suited for a multi-core machine. In this model, a number of processor communicates via shared memory. Our design goals were to maximize memory efficiency, and at the same time ensure maximal parallelism. In particular, our design avoids the storage of predecessor edges, and is lock-free, and almost wait-free. This puts quite some restrictions on the heuristics to select the next vertices to lift; in particular, the absence of direct predecessor information is tough.

Section 2 introduces parity games, Jurdzinski's sequential algorithm based on progress measures, and some examples of μ-calculus formulas. The design of our parallel algorithm and implementation, and various heuristics that we developed are described in Sections 3 and 4, respectively. Finally, Section 5 reports the experiments and measurements that we performed on games arising from model checking intricate fairness properties on communication protocols, and from randomly generated and constructed problem instances.

1.2 Related Work

The most well-known sequential model checkers for full μ-calculus are the explicit state model checker CWB-NC [5] and the symbolic model checker NuSMV [11]. From the numerous sequential model checkers for fragments of the μ-calculus, we only mention the Evaluator from the CADP toolset [32], for regular alternation free μ-calculus, which is based on Boolean Equation Systems.

For comparing parallel and distributed model checkers, we distinguish multi-core checkers from distributed checkers on clusters of workstations. Furthermore, we have to distinguish various sublogics: (1) reachability analysis/state space generation; (2) alternation free μ-calculus, including CTL; (3) one-alternation case, with sublogics (3a) LTL; (3b) CTL*; (5) full μ-calculus. Reviewing (1) is out of scope of this paper.

For the alternation-free case (including CTL), we are only aware of distributed implementations. One of the first approaches is reported in [3]. The work in [8] is based on game graphs. The Evaluator's algorithms for alternation-free have been distributed in [23,24]. An assumption-based approach to distribution is reported in [10].

A distributed algorithm for the one-alternation case was proposed in [30] and implemented in [19]. In theory, this fragment includes LTL and CTL*, but in practice an effective translation (i.e., without exponential blow-up in the size of the formula) from LTL and CTL* to μ-calculus is not known [13,6]. Several approaches for distributed LTL model checking exist; we only mention multi-core implementations. A dual-core implementation for SPIN is described in [20] and a multi-core implementation for LTL model checking is in [2].

For CTL*, a multi-core parallel implementation is described in [21]. The algorithm is based on Hesitant Alternating Games and explores the induced and/or graph with a stack to detect cycles. The approach is reported to be close to [8]. The authors also discuss pitfalls for shared-memory implementations.

For distributed model checking of the full μ-calculus, we are only aware of the

symbolic approach reported in [18]. This algorithm is based on distributing Binary Decision Diagrams over a network of workstation.

Finally, a different route for tackling μ-calculus model checking is the reduction from parity games to the satisfiability problem, as presented by Lange [29]. Like this work, it is also based on progress measures. In combination with a parallel SAT solver (e.g., [37]), this could give rise to a parallel solver for the μ-calculus. We leave this line of research as future work.

We conclude that there exists only one distributed implementation for full μ-calculus. It is based on BDDs and not available as a tool. All other distributed implementations work for fragments only. The only multi-core implementation for a branching logic that we know of is for CTL*.

2 Parity Games, Progress Measures and μ-Calculus

2.1 Parity Games

A *parity game* is of the form $(V_\Box, V_\Diamond, E, p)$, where $V := V_\Box \uplus V_\Diamond$ is a finite set of vertices, partitioned disjointly in $V_\Box$ where player Odd moves, and $V_\Diamond$ where player Even moves. The set of edges $E \subseteq V \times V$ denotes the possible moves, and is assumed to be total (i.e., every vertex has an outgoing edge). Finally, $p : V \to \{0, 1, \ldots, d-1\}$ is the *priority function*, assigning a natural number below a given d to each vertex.

A *play* is an infinite sequence $\pi = \pi_0\pi_1 \ldots$ of vertices such that $\forall i.\, \pi_i E \pi_{i+1}$. A play is *won* by player Even, if the least number that occurs infinitely often in π is even; otherwise player Odd won the game.

A *strategy* for player Even is a selection function $\sigma : V_\Diamond \to V$, such that $\forall v \in V_\Diamond.\, vE\sigma(v)$. A play π is *consistent* with σ, if $\pi_{i+1} = \sigma(\pi_i)$ for all i with $\pi_i \in V_\Diamond$. A strategy σ for Even is *winning on* $w \in V$, if every play π starting in w and consistent with σ is won by Even. The *winning set* of player Even, $W_\Diamond$, is defined as the set of all vertices $w \in V$ on which player Even has a winning strategy. Solving a parity game means computing the winning set $W_\Diamond$.

2.2 Progress Measures

We now recapitulate Jurdzinski's algorithm to solve parity games based on small progress measures [25]. The algorithm associates to each vertex an integer vector, whose length equals the number of priorities; this gives rise to $\mathcal{O}(n \times d)$ memory. For each vertex, its vector is initially $\overline{0}$, and increases in the lexicographic order until a fix point is reached. For odd i, the number in the i-th component is limited by the number of vertices with priority i. For even i, the i-th component is kept 0. This gives rise to the $n^{d/2}$ complexity.

Given a parity game $G = (V_\Box, V_\Diamond, E, p)$ with $V := V_\Box \uplus V_\Diamond$ and priorities bounded by d, let m_i (for $i < d$) be the number of elements $v \in V$ with $p(v) = i$. Define $M_i := \{0\}$, if i is even; $\{0, 1, \ldots, m_i\}$ if i is odd. Define $M_G := M_0 \times M_1 \times \cdots \times M_{d-1}$, and define $M_G^\top := M_G \cup \{\top\}$.

For vectors $\alpha, \beta \in M_G$, $\alpha < \beta$ denotes the lexicographic order, while the pre-

orders $<_i$, $=_i$, $>_i$ truncate the vectors to components $0, \ldots, i$ before comparing them. This is extended to $M_G^\top$, by considering $\top$ as the maximal element in the orderings.

For $\alpha \in M_G^\top$, with $\mathrm{succ}_i(\alpha)$ we denote the minimal $\beta \in M_G^\top$ such that either i is even and $\alpha \leq_i \beta$, or i is odd and ($\alpha <_i \beta$ or $\alpha = \beta = \top$).

Example 2.1 Let $d = 5$ and $(m_i)_{i<d} = (0, 5, 0, 3, 0)$. We then have $(0, 3, 0, 2, 0) < (0, 4, 0, 3, 0)$ (lexicographically) and $(0, 3, 0, 2, 0) =_2 (0, 3, 0, 4, 0)$, because after truncation we get $(0, 3, 0) = (0, 3, 0)$. We have the following sequence of $<_3$-bigger successors in $M_G^\top$:

$$(0, 3, 0, 2, 0) <_3 (0, 3, 0, 3, 0) <_3 (0, 4, 0, 0, 0) <_3 (0, 4, 0, 1, 0) <_3 \cdots$$

Truncating vectors at 1, we get the following sequence of $<_1$-bigger successors:

$$(0, 3, 0, 2, 0) <_1 (0, 4, 0, 0, 0) <_1 (0, 5, 0, 0, 0) <_1 \top$$

Note that the $\leq_2$-bigger successor $\mathrm{succ}_2(0, 3, 0, 2, 0) = (0, 3, 0, 0, 0)$, even though it gets smaller in $<$. □

The goal of the algorithm is to compute a so called *game parity progress measure*, which is an assignment $\varrho : V \to M_G^\top$, such that for all $v \in V$:

- If $v \in V_\Diamond$ then $\varrho(v) \geq_{p(v)} \mathrm{succ}_{p(v)}(\varrho(w))$ for some $(v, w) \in E$
- If $v \in V_\Box$ then $\varrho(v) \geq_{p(v)} \mathrm{succ}_{p(v)}(\varrho(w))$ for all $(v, w) \in E$

The least game parity progress measure is computed as the least simultaneous fix point of lifting the measure of each vertex when needed, starting with the 0-vector. Here, with an eye towards the distributed implementation, we allow that a whole set U gets lifted at once. Choosing only singletons for U yields the original algorithm of [25], computing the same fix point. In Sec. 4 we discuss heuristics to choose U in a parallel algorithm. The complete algorithm is given by:

$$\begin{array}{l} \varrho := \lambda v \in V.\, (0, \ldots, 0) \\ \textbf{while } \exists U \subseteq V.\, \varrho < \mathit{Lift}(\varrho, U) \textbf{ do } \varrho := \mathit{Lift}(\varrho, U) \end{array}$$

$$\text{where } \mathit{Lift}(\varrho, U)(v) = \begin{cases} \varrho(v) & \text{if } v \notin U \\ \mathit{min}_{(v,w)\in E} \mathrm{succ}_{p(v)}(\varrho(w)) & \text{if } v \in U \cap V_\Diamond \\ \mathit{max}_{(v,w)\in E} \mathrm{succ}_{p(v)}(\varrho(w)) & \text{if } v \in U \cap V_\Box \end{cases}$$

Theorem 2.2 (Jurdzinski [25]) *If ϱ is the least game parity progress measure for parity game G, then the winning set is $W_\Diamond = \{w \mid \varrho(w) \neq \top\}$. Moreover, the strategy $\tilde{\varrho}(v : V_\Diamond) := \mathit{min}_{(v,w)\in E}(\varrho(w))$ is a winning strategy for player Even.*

2.3 Modal μ-Calculus

We briefly recapitulate the modal μ-calculus. It has the constants true and false, Boolean connectives, modal operators ($[a]$ meaning "for all direct a-successors" and

$\langle a \rangle$ meaning "there exist a direct a-successor"), and the least (μ) and greatest (ν) fix point operators, binding propositional variables X.

$$\Phi ::= \mathsf{true} \mid \mathsf{false} \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \neg\Phi \mid [a]\Phi \mid \langle a \rangle \Phi \mid X \mid \mu X.\, \Phi \mid \nu X.\, \Phi$$

These formulas are interpreted over Labeled Transition Systems (LTS). An LTS is a tuple $L = (S, A, R)$, where S is the set of states, A the set of action labels, and $R \subseteq S \times A \times S$ is the set of transitions.

It is well known that the model checking problem $L, s \vDash \Phi$ can be transformed to a parity game of size $|L| \times |\Phi|$ [31,9]. We will not provide a translation here, but only give a small table, which is useful to relate the logical connectives to the game language.

Player	Symbol	Logical operators
Even	$\Diamond$	$\langle\cdot\rangle, \vee, \nu$
Odd	$\Box$	$[\cdot], \wedge, \mu$

In particular, the outermost (most significant) μ-quantified variable will correspond to the highest odd priority 1. The outermost ν variable corresponds to 2 or 0, depending on whether it is enclosed by a μ or not.

In the sequel we introduce a number of formulas in the μ-calculus, partly to show its potential and also because we used them in our experiments. In all formulas below, we assume that the LTS does not contain deadlocks (which could be checked separately by the formula $\nu X.\, [\mathsf{true}]X \wedge \langle \mathsf{true} \rangle \mathsf{true}$), in order to simplify the formulas to their essential structure.

(1) From any reachable state, there exist a path with infinitely many τ-steps (and possibly some other steps).

$$\varphi_{\text{inftau}} = \nu X.\, ([\mathsf{true}]X \wedge \nu Z.\, \mu Y.\, (\langle \mathsf{true} \rangle Y \vee \langle \tau \rangle Z))$$

Intuitively, the νX part makes sure that the rest holds in every reachable state. The νZ part allows infinitely many τ's, and the μY ensures that only finitely many other actions can occur in between. In CTL* (talking about actions) this could be expressed as $AG(EGF\tau)$. Note that this formula cannot be expressed in LTL, nor in CTL.

(2) For any sequence, if it contains infinitely many r steps, then it contains infinitely many w steps.

$$\varphi_{\text{infrw}} = \nu X.\, \mu Y.\, \nu Z.([w]X \wedge [r]Y \wedge [\neg r \wedge \neg w]Z)$$

This outer νX ensures that any trace with infinitely many w steps is accepted. If this doesn't apply, only a finite number of r steps is accepted by the μY. The inner νZ still allows infinitely many other steps in this case.

This formula can be expressed in LTL by $A(GFr \rightarrow GFw)$. Dams describes how this could be translated to a μ-calculus formula with one alternation [13], but

the blowup in size would be considerable. We think it is interesting that a concise μ-calculus formula with two alternations exist.

3 Parallel Algorithm

Parallelizing the small progress measures algorithm for a multi-core setting requires surprisingly little modification. Alg. 1 depicts the pseudo code.

Algorithm 1 ParallelSmallProgress$_N(\varrho, V)$

Require: $\varrho(v) = \langle 0, \ldots, 0 \rangle$ for all $v \in V$
Require: $V = \biguplus_{i=1}^{N} U_i$
1: **repeat**
2: **for all** $i \in \{1, \ldots, N\}$ **in parallel do**
3: $lifted_i \leftarrow$ LiftSome(ϱ, U_i)
4: $lifted \leftarrow \bigvee_{i=1}^{N} lifted_i$
5: **until** $\neg lifted$
6: **return** ϱ

We assume the parity graph to be a read-only data structure, shared between N workers. The set of vertices V in the graph is partitioned into disjunct sets U_i, each of which is owned by a single worker. Furthermore, all progress measures are shared between workers. A measure $\varrho(v)$ with $v \in U_i$ can only be modified by its owner, worker i. On the other hand, $\varrho(v)$ can be read at any time by any other worker. This is a classical *single writer/multiple readers* situation. We return to the resulting synchronization issues in Sec. 3.1.

The algorithm proceeds as follows: all partitions U_i are handled in parallel by their owning workers (lines 2–3). Function LiftSome(ϱ, U_i) returns whether at least one measure was lifted in partition U_i. If in at least one partition a measure has been lifted (line 4), this can potentially result in further lifts. Thus, the whole process repeats (line 5) until no measure was lifted, in which case a fix point has been reached and the algorithm terminates.

The function ParallelSmallProgress adds an implicit synchronization step before line 4 to detect termination and is thus not completely *wait-free*. This is not an issue in practice, if V is partitioned such that the amount of work for each worker is similar, and the number of iterations of the loop in lines 1–5 is small in comparison to the number of lift operations performed.

As mentioned before, the lifting for each partition is carried out by function LiftSome (Alg. 2). In lines 4–8, we iterate over the vertices in partition U in an ordering determined by a permutation H with repetition allowed. H can be chosen arbitrarily without affecting the correctness of the algorithm, however, some orderings are better than others. We will return to the topic of choosing a good H in Sec. 4.

Algorithm 2 LiftSome(ϱ, U)

```
1:  lifted ← false
2:  repeat
3:    lifted′ ← false
4:    for all v ∈ H(U) do
5:      α_v ← Lift(ϱ, v)
6:      if ϱ(v) < α_v then
7:        ϱ(v) ← α_v
8:        lifted′ ← true
9:    lifted ← lifted ∨ lifted′
10: until ¬lifted′ or some arbitrary number of iterations reached
11: return lifted
```

3.1 Updating Measures in Parallel

Continuing with LiftSome, we note that multiple instances of this function are executed in parallel, modifying the globally shared ϱ in line 7 without any synchronization between workers.

We now recall that the value of $\textsc{Lift}(\varrho, v)$ depends on the measures of v's successors. They can be updated by their respective owners executing line 7 at the same time when $\textsc{Lift}(\varrho, v)$ reads their values in line 5. For this to be correct, a sufficient condition is that the assignment in line 7 is executed atomically. We can then distinguish two cases:

- When reading $\varrho(w)$ with $(v, w) \in E$, a worker sees the new value α_w for w. $\varrho(v)$ is updated accordingly.
- A worker still sees the old value of $\varrho(w)$. This is not harmful, as it is the same situation as if $\textsc{Lift}(\varrho, v)$ happened before the $\varrho(w) \leftarrow \textsc{Lift}(\varrho, w)$ update. Eventually, this update will be carried out, and any change in value will be picked up by the next invocation of $\textsc{Lift}(\varrho, v)$. A change of $\varrho(w)$ will also force at least one more iteration over V.

3.2 Atomic Updates without Locking

In the previous section, we have seen the need for an atomic update in line 7 of LiftSome (Alg. 2). Several options exist to realize this without resorting to mutual exclusion primitives like locks.

The easiest and by far fastest option is that a progress measure fits into a machine word, as these usually allow atomic updates.[3] With a contemporary 64-bit processor, we can fit measures for games with a small maximal priority d_{small} into a 64-bit word. This can be achieved by storing in a measure only the counts for the odd priorities (even priorities have constant zero counts), and the additional constraint that $\sum_{i=1,\text{odd}(i)}^{d_{\text{small}}} \lceil \log_2 m_i \rceil < 64$, and one value reserved to denote $\top$.

[3] This is true at least for Intel EM64T and AMD x86-64 processors.

If the parity game has more priorities, we can fall back to a different scheme, still under the assumption that a machine word can be updated atomically. Instead of updating a measure vector directly, we then introduce an indirection through indexed sets (one for each worker) of measures. If a new measure α_v in line 5 (Alg. 2) is not in the indexed set already, it is inserted (being a local operation this needs no locking). In any case, the measure's index in the set uniquely identifies it, and thus can be assigned to $\varrho(v)$ (atomically again) in line 7 instead of the measure itself. Reading the measure of v then requires to look up the measure at index $\varrho(v)$.

It is sufficient to store at most $|U_i| + 1$ different measure vectors in the indexed set for each worker i at any given time. Usually, the number is lower than that, considering for example the initial situation, where all measures are initialized to the zero vector. Hence, this scheme also conserves memory.

On the other hand, the number of measures encountered during the run of the algorithm is much higher than that. This suggests that the indexed set needs to be coupled with a *reference counting* or *garbage collection* scheme to be feasible. Larger measures could even benefit from the vector folding described in a different context [7].

Both solutions to the atomicity problem presented so far rely on machine word-sized atomic updates, but the availability of such an operation might not always be guaranteed. A third solution, with only the weaker requirement of a processor not reordering memory writes[4] is to update the measure vector starting from the *least significant bit* (lsb). This way, arbitrarily large vectors can be supported without locking.

With a small change to LiftSome (replacing the assignment in line 5 with $\alpha_v \leftarrow \max\{\varrho(v), \textsc{Lift}(\varrho, v)\}$), the algorithm becomes robust against workers reading a partially updated measure α_{part}. It is important to realize that partial lsb-first updates do not exceed the actual new measure value (while msb-first updates very well can). If $\textsc{Lift}(\varrho, v)$ returns α_{part} while it should actually return α_v, erroneously α_{part} could be propagated. However, the $\max\{\dots\}$ modification ensures the monotonicity of ϱ at position v: if $\varrho(v) \geq \alpha_{\text{part}}$, the old measure is retained. If $\varrho(v) < \alpha_{\text{part}}$, α_{part} becomes the new measure for v, and possibly propagates. Eventually $\textsc{Lift}(\varrho, v)$ will return the correct α_v (lock-freeness guarantees progress). Because $\alpha_v \geq \alpha_{\text{part}}$ (due to lsb-first updates), α_v will be assigned to $\varrho(v)$, and propagate as well, thus overwriting any earlier propagation of α_{part}. The additional work is the price to pay for less synchronization.

3.3 Implementation Details

For our experiments, we implemented the algorithm for a thread-based multi-core setting (shared memory). We used the *Intel Threading Building Blocks* (TBB) [22] as concurrency abstraction. It provides high-level operations like `parallel_for` and `parallel_reduce`, shielding us from many threading details and allowing a very concise implementation, with close resemblance to the pseudo code presented here.

[4] This is true for the Intel EM64T architecture, but not for Intel Itanium, for example.

For scalable memory allocation within worker threads, the TBB's `scalable_allocator` is employed. Regarding the partitioning of V, we simply split it into evenly sized pieces, one per worker.

4 Selecting Measures for Lifting

In Alg. 2, we abstracted from the order in which measures are lifted by means of a permutation H, with allowed repetition. In this section, we will discuss a number of possible choices for H. These are heuristics, meaning that they are not always optimal, but our experiments confirm that they work well in practice.

4.1 Work List Approach

The value of a progress measure for vertex v depends only the measures of its immediate successors. In other words, changes effectively propagate backwards through the parity graph. A natural selection strategy is then to push *predecessors* of v onto a *work list*, if $\varrho(v)$ was lifted. The next vertex to be lifted is then taken from the work list. The termination criterion of the algorithm is an empty work list. Different data structures for the work "list" are conceivable: stack, (priority) queue, etc.

A severe drawback is that this scheme requires storing predecessors information for the parity graph. In combination with the forward-looking measure computation in Lift, this means we need to effectively store the graph twice, or alternatively store measures twice (with the common back-propagation).

For a parallel algorithm, another drawback of this work list approach is the need for synchronization. Several strategies exist to avoid global locks on a shared work list, for example, let each worker thread have a private work list, and queues to inject vertices into other workers' work lists. However, these solutions tend to be complex, and are usually at least not *wait-free*, or even not *lock-free*.

4.2 Swiping

A simpler solution is *swiping*: each worker continuously iterates over its partition U_i. At least in the beginning this appears to be an effective strategy, considering that usually most measures can be lifted a few times. However, in the worst case only one vertex per iteration over U_i is lifted successfully, causing $\mathcal{O}(n)$ work per single lift operation. This case can be avoided, if the iteration order matches the backward-propagation nature of measures. For example, if for most $(v, w) \in E$, v comes before w in the sequence of vertices, swiping *backwards* (i.e., starting with the last vertex) over the sequence will approximate the above work list selection scheme without its storage overhead.

Fortunately, generating the parity graph by any forward exploration procedure provides us with such an ordering of vertices for free, and as we will see in Sec. 5, it has the desired effect, when swiping the resulting vertex list in back-to-front manner. Conversely, when swiping in *forward* order (starting from the first vertex)

we can expect very little lifts per iteration.

4.3 Focus List Approach

In order to forego the use of predecessor information, but still get the benefit of finding with little work measures which are likely to change, we consider a strictly forward-looking heuristic.

The basic assumption is that a vertex whose measure changed once might have it change again. Hence, during swiping we put vertices with changing measure onto a size-bounded *focus list* with a non-zero "credit" value. After the swipe across U finished (line 9), vertices on the focus list are handled. If their measures keep changing, their credit is increased linearly, if they don't change, it is decreased. Vertices whose credit drops below zero are removed from the focus list. This continues until the focus list becomes empty, in which case we resume with line 10.

Note, that with the above scheme, a vertex will stay on the focus list even if its measure is not changing at every lift attempt, as long as it still has enough credit left.

Vertices which are part of a cycle in the parity graph can gain very high credit until their measures cease to change. However, once stable, we would like them to be removed from the focus list as fast as possible, otherwise many fruitless lift attempts would be spent on them.

A linear credit decay for failed lifting attempts can be ruled out, as it would take roughly as many failed attempts as a vertex had successful lifts, to drop it from the focus list. Fortunately, an exponential decay turns out have the qualities we are seeking: after accumulating some credit, a vertex can survive a number of unsuccessful lifting attempts (and, more importantly, attempts with alternating success), but once it stops changing, the exponential decay ensures its fast removal from the focus list.

A focus list can significantly speed up finding a solution if a small self-increasing cycle gets "trapped" on it, because many lift attempts will be successful. Unfortunately, we can also expect the absence of speed-up, considering the following two scenarios:

(i) Measures belonging to a cycle end up on different focus lists (which are allocated per-worker). Due to the lack of synchronization between workers, this setting would be unstable: if those measures are lifted in an unfortunate order, they don't gain enough credit to stay on the focus lists, and continually bounce off it.

(ii) All measures belonging to a cycle end up on the same focus list, causing a high load for the worker it belongs to, while other workers tend to become idle.

Table 1
Running ParallelSmallProgress$_N$ (with varying heuristics plugged in) on different types of parity games. Size (number of vertices in the parity graph), out degree and maximum priority are parameters of the randomly generated parity graphs. Elapsed wall clock time is represented in seconds. "t/o" means time-out (> 1200 seconds).

N	Forward	Backward	Backward/Focus	Backw./Focus-1
(1) onebit_6 $\models \varphi_{\text{infrw}}$: size 3,586,945				
1	7.71	3.19	3.38	5.42
2	3.96	1.93	2.07	3.17
4	2.27	1.27	1.39	2.15
8	1.83	1.26	1.25	2.03
(2) onebit_6 $\models \varphi_{\text{inftau}}$: size 8,930,305				
1	2.87	3.35	3.51	5.72
2	1.79	2.03	2.11	3.41
4	1.28	1.42	1.46	2.27
8	1.12	1.15	1.17	1.77
(3) swp_2_5 $\models \varphi_{\text{infrw}}$: size 17,747,461				
1	90.64	12.38	24.63	22.87
2	50.34	7.78	22.77	15.47
4	31.67	6.14	19.84	10.94
8	24.22	6.83	21.99	11.23
random-23413: size 5,000,000, out deg. 32, max. prio. 4				
1	29.85	29.82	39.51	34.36
2	16.46	17.34	21.37	21.03
4	11.45	11.54	14.53	13.12
8	8.64	8.79	10.55	10.02
random-26126: size 100,000, out deg. 4, max. prio. 4				
1	t/o	t/o	34.68	37.82
2	t/o	t/o	73.83	135.71
4	t/o	t/o	t/o	t/o
8	t/o	t/o	t/o	t/o

5 Experiments

We evaluated the algorithm presented here with several selection heuristics on different types of problem instances. These will be explained in more detail in the coming sections. For the sake of discussion, we list some representative measurements in Tab. 1. Columns labelled "Forward", "Backward", etc., refer to the heuristics introduced in Sec. 4.2 and 4.3.

All our experiments were performed on a computer with two Quad-Core Intel Xeon E5320 processors (1.86 GHz) and 8 GB RAM, running Linux 2.6.18; we ran the algorithm in turn with $N = 1$, 2, 4, and 8 worker threads enabled, each five times. Because run times of repeated experiments vary little, we use the average run time here. Times represent the wall clock time (in seconds) needed to calculate M_G, and the actual computation of progress measures. This being a global algorithm, it is assumed that the parity game graph is already present in memory. Hence, generation time is excluded. Another reason is that parity games can be obtained from a number of different sources with varying performance characteristics.

We note that the measurements presented here do not exceed a maximum priority of 4, in part, for lack of suitable test cases. Our implementation uses the efficient machine word representation mentioned in Sec. 3.2 if possible, but falls back to a slower representation for games with higher priority. Experiments have shown that this representation pessimizes results by a factor of around 30, however, we also choose not to spend effort on optimizing it.

5.1 Case Studies

We now describe the case studies based on model checking. We checked existing μCRL specifications of communication protocols against the formulas of Sec. 2.3: With the mcrl2 toolset [17], we obtain a *simple BES* from the specification and the formula. This process is akin to state space generation and well understood. In our cases, this resulted in a simple BES, which is isomorphic to a parity game graph, on which we applied our solver. The used procedure ensures that all the vertices in the game graph are reachable from some initial vertex, and that the game graph is generated in breadth-first order.

We took the following well-known communication protocols. In all cases, a sender tries to transmit data to a receiver through a lossy channel, and acknowledgements are sent back through another lossy channel.

Sliding Window Protocol In SWP [35,1] data is buffered in a window, and the whole window can be transmitted, allowing for a higher usage of bandwidth. In this case, data elements are distinguished by sequence numbers modulo twice the maximal window length. On asynchronous arrival of acknowledgements, the windows slides forward. swp_M_N sends M data elements, with window size N.

Onebit Sliding Window Protocol The Onebit SWP [4] is similar to the Sliding Window Protocol, but the window is limited to size 1. While very simple, this protocol is more interesting than, e.g., the *alternating bit protocol* (ABP), because it allows more parallel behavior than ABP. onebit_M denotes Onebit SWP with M distinct data elements.

In addition, we conducted experiments with the already mentioned ABP, and the *bounded retransmission protocol*, however, those results are elided because they show no significantly different behavior.

Evaluation

Concentrating first on the top half of Tab. 1, we can see gains by increasing the number of worker threads (N) for most of the combinations of case studies and selection heuristics, albeit quite varying. It turns out that the complete asynchronicity of workers for most of the run time has the downside of causing extra iterations until a fix point is reached, in particular, if the amount of work for each drops below a certain threshold. In addition, we cannot rule out effects of *false sharing*, as have been observed by Inggs and Barringer [21].

It is perhaps interesting to know that case (3) takes up around 1.3 GB RAM to store the complete graph with forward edges, and the progress measures. This is relatively small still, and in the range of a common desktop workstation. We were unable to fit the next bigger instance, swp_2_6 $\models \varphi_{\text{infrw}}$, into our 8 GB memory limit, as it is larger by a factor of 7.

Because of the aforementioned way of generating the game graphs in breadth-first order, the "Forward" selection heuristic performs here almost always worse than the "Backward" heuristic, sometimes significantly so, see case (3). This confirms

our argument from Sec. 4.2.

Interestingly, due the maintenance overhead, the focus list heuristic (here filled with a backward swipe) fares worse than a simple "Backward" heuristic, if the parity graph is in the right order. This is particular bad in case (3) again, where all speed-up is destroyed. Limiting the number of iterations in loop 2–10 of Alg. 2 (here to a single iteration) can alleviate the issue, and this despite of the early exit causing more iterations of the outer loop in Alg. 1, and hence causing actually more synchronization.

The advantages of the focus list becomes apparent in the following section, because it catches some corner cases.

5.2 Generated Problem Instances

In the bottom half of Tab. 1, we show two interesting randomly generated parity games. Instance random-23413 exhibits decent speed-up with all heuristics, in particular the random order of vertices makes "Forward" and "Backward" swiping behave identical. On the other hand, only the focus list heuristics could handle random-26126, even after increasing the time-out for the others to several hours. It behaves as anticipated: a small cycle gets trapped and the corresponding measures are increased very quickly to their final value.

While the trapping of small cycles appears to work fairly reliable, so far we encountered such situations only in our randomly generated instances, where they appear frequently. On the downside, and as predicted, situations like in random-26126 seem to limit the effectiveness of our parallel implementation, and in this case it is even detrimental, for the reasons discussed in Sec. 4.3: increasing the number of workers also *increases* the run-time! Fortunately, our experiences so far indicate that "real-world" case studies are much more well behaved.

Hard Series. Quite expectedly, when we tried our solver on some examples designed to be hard nuts for an algorithm based on parity games, e.g., the one from Jurdzinski's original paper [25], we failed to obtain an answer within a reasonable amount of time, already for small instances.

6 Conclusion

We presented a multi-core algorithm for solving parity games and experimented with selection heuristics which can speed up the algorithm significantly. We tested this on a number of realistic case studies, with encouraging results. However, we have not touched here a number of topics which will likely increase the basic algorithm's efficiency: as mentioned by Jurdzinski [25], preprocessing the parity game can prove beneficial (SCC decomposition, eliminating trivial cycles, compressing priority ranges, etc.). Furthermore, the frequency count of priorities gives a good indication whether switching to the co-game is more efficient. Other future work includes more research into heuristics, in combination with a load-balancing scheme.

Acknowledgement

We are indebted to Martin Leucker for pointing us to Jurdzinski's algorithm, and for discussions on the topic during his visit at CWI. We also wish to thank Jan Friso Groote for assistance with the mcrl2 toolset when generating some parity game instances used here.

References

[1] Badban, B., W. Fokkink, J. F. Groote, J. Pang and J. van de Pol, *Verification of a sliding window protocol in μCRL and PVS*, Formal Asp. Comput. **17** (2005), pp. 342–388.

[2] Barnat, J., L. Brim and P. Rockai, *Scalable multi-core LTL model-checking*, in: D. Bosnacki and S. Edelkamp, editors, *SPIN*, Lecture Notes in Computer Science **4595** (2007), pp. 187–203.

[3] Bell, A. and B. R. Haverkort, *Sequential and distributed model checking of petri nets*, STTT **7** (2005), pp. 43–60.

[4] Bezem, M. and J. F. Groote, *A correctness proof of a one-bit sliding window protocol in μCRL*, Comput. J. **37** (1994), pp. 289–307.

[5] Bhat, G. and R. Cleaveland, *Efficent local model-checking for fragments of the modal μ-calculus*, in: T. Margaria and B. Steffen, editors, *TACAS*, Lecture Notes in Computer Science **1055** (1996), pp. 107–126.

[6] Bhat, G. and R. Cleaveland, *Efficient model checking via the equational μ-calculus*, in: *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society Press, New Brunswick, New Jersey, 1996, pp. 304–312.

[7] Blom, S., B. Lisser, J. van de Pol and M. Weber, *A database approach to distributed state space generation*, in: B. Haverkort and I. Černa, editors, *Proc. of the 6th International Workshop on Parallel and Distributed Methods in verifiCation*, ENTCS, 2007, pp. 17–32.

[8] Bollig, B., M. Leucker and M. Weber, *Local parallel model checking for the alternation-free μ-calculus*, in: *Proc. 9th International SPIN Workshop on Model Checking of Software (SPIN '02)*, Lecture Notes in Computer Science **2318** (2002), pp. 128–147.

[9] Bradfield, J. and C. Stirling, *Modal logics and mu-calculi: an introduction*, in: J. Bergstra, A. Ponse and S. Smolka, editors, *Handbook of Process Algebra*, North-Holland, 2001 pp. 293–330.

[10] Brim, L., K. Yorav and J. Zidkova, *Assumption-based distribution of CTL model checking*, STTT **7** (2005), pp. 61–73.

[11] Cimatti, A., E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani and A. Tacchella, *Nusmv 2: An opensource tool for symbolic model checking*, in: E. Brinksma and K. G. Larsen, editors, *CAV*, Lecture Notes in Computer Science **2404** (2002), pp. 359–364.

[12] Cleaveland, R. and B. Steffen, *A linear–time model–checking algorithm for the alternation–free modal mu–calculus.*, in: K. G. Larsen and A. Skou, editors, *Proceedings of Computer-Aided Verification (CAV'91)*, Lecture Notes in Computer Science **575** (1992), pp. 48–58.

[13] Dam, M., *CTL* and ECTL* as fragments of the modal μ-calculus*, Theoretical Computer Science **126** (1994), pp. 77–96.

[14] de Alfaro, L. and M. Faella, *An accelerated algorithm for 3-color parity games with an application to timed games*, in: W. Damm and H. Hermanns, editors, *CAV*, Lecture Notes in Computer Science **4590** (2007), pp. 108–120.

[15] Grädel, E., *Positional determinacy of infinite games*, in: *Proceedings of STACS 2004: 21st Annual Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science **2996** (2004), pp. 2–18.

[16] Groote, J. F. and M. Keinänen, *Solving disjunctive/conjunctive boolean equation systems with alternating fixed points*, in: K. Jensen and A. Podelski, editors, *TACAS*, Lecture Notes in Computer Science **2988** (2004), pp. 436–450.

[17] Groote, J. F., A. H. J. Mathijssen, M. A. Reniers, Y. S. Usenko and M. J. van Weerdenburg, *The formal specification language mCRL2*, in: E. Brinksma, D. Harel, A. Mader, P. Stevens and R. Wieringa, editors, *Proc. of Methods for Modelling Software Systems (MMOSS)*, Dagstuhl Seminar Proceedings 06351, 2007.

[18] Grumberg, O., T. Heyman and A. Schuster, *Distributed symbolic model checking for μ-calculus*, Formal Methods in System Design **26** (2005), pp. 197–219.

[19] Holmén, F., M. Leucker and M. Lindström, *Uppdmc: A distributed model checker for fragments of the mu-calculus*, ENTCS **128** (2005), pp. 91–105.

[20] Holzmann, G. J. and D. Bosnacki, *Multi-core model checking with SPIN*, in: *IPDPS* (2007), pp. 1–8.

[21] Inggs, C. P. and H. Barringer, *CTL* model checking on a shared-memory architecture*, Formal Methods in System Design **29** (2006), pp. 135–155.

[22] Intel Corporation, "Intel Threading Building Blocks," (2006).
URL http://threadingbuildingblocks.org/

[23] Joubert, C. and R. Mateescu, *Distributed local resolution of boolean equation systems*, in: *PDP* (2005), pp. 264–271.

[24] Joubert, C. and R. Mateescu, *Distributed on-the-fly model checking and test case generation*, in: A. Valmari, editor, *SPIN*, Lecture Notes in Computer Science **3925** (2006), pp. 126–145.

[25] Jurdzinski, M., *Small progress measures for solving parity games*, in: H. Reichel and S. Tison, editors, *STACS*, Lecture Notes in Computer Science **1770** (2000), pp. 290–301.

[26] Jurdzinski, M., M. Paterson and U. Zwick, *A deterministic subexponential algorithm for solving parity games*, in: *SODA* (2006), pp. 117–123.

[27] Klauck, H., *Algorithms for parity games*, in: E. Grädel, W. Thomas and T. Wilke, editors, *Automata, Logics, and Infinite Games*, Lecture Notes in Computer Science **2500** (2002), pp. 107–129.

[28] Kozen, D., *Results on the propositional mu-calculus*, Theoretical Computer Science **27** (1983), pp. 333–354.

[29] Lange, M., *Solving parity games by a reduction to SAT*, in: *Proc. of the Workshop on Games in Design and Verification (GDV)*, 2005.

[30] Leucker, M., R. Somla and M. Weber, *Parallel model checking for LTL, CTL* and L^2_μ*, in: L. Brim and O. Grumberg, editors, *Proceedings of the 2nd International Workshop on Parallel and Distributed Methods in verifiCation*, ENTCS **89** (2003), pp. 4–16.

[31] Mader, A., "Verification of Modal Properties Using Boolean Equation Systems," Ph.D. thesis, Technische Universität München (1996).

[32] Mateescu, R., *CAESAR_SOLVE: A generic library for on-the-fly resolution of alternation-free boolean equation systems*, STTT **8** (2006), pp. 37–56.

[33] Schewe, S., *Solving parity games in big steps*, in: V. Arvind and S. Prasad, editors, *FSTTCS*, Lecture Notes in Computer Science **4855** (2007), pp. 449–460.

[34] Seidl, H., *Fast and simple nested fixpoints*, Information Processing Letters **59** (1996), pp. 303–308.

[35] Tanenbaum, A. S., "Computer Networks," Prentice-Hall, 1981.

[36] Vöge, J. and M. Jurdzinski, *A discrete strategy improvement algorithm for solving parity games*, in: E. A. Emerson and A. P. Sistla, editors, *CAV*, Lecture Notes in Computer Science **1855** (2000), pp. 202–215.

[37] Yulik Feldman, N. D. and Z. Hanna, *Parallel multithreaded satisfiability solver: Design and implementation*, Proc. of the 3rd International Workshop on Parallel and Distributed Methods in Verification **128** (2005), pp. 75–90.