On Parallel Software Verification using Boolean Equation Systems

Alexander Ditter[1], Milan Češka[2], and Gerald Lüttgen[1]

[1] University of Bamberg, 96045 Bamberg, Germany,
{alexander.ditter,gerald.luettgen}@swt-bamberg.de,
[2] Masaryk University, 602 00 Brno, Czech Republic
xceska@fi.muni.cz

Abstract. Due to the recent technological developments multi-core and many-core hardware platforms have become widely accessible. These parallel architectures have been used to significantly accelerate many computationally demanding tasks. In this paper we describe a parallel approach to solve *Boolean Equation Systems* (BESs) in the context of *model checking*. We focus on the applicability of state-of-the-art, shared-memory parallel hardware – multi-core CPUs and many-core GPUs – to speed up the resolution procedure for BESs. For this setting, we experimentally show the scalability and competitiveness of our parallel approach, compared to an optimized sequential implementation, based on a large benchmark suite with examples from industry and academia.

Keywords: formal verification, parallel model checking, boolean equation systems

1 Introduction

In this paper we propose and evaluate a parallel approach to the resolution of *Boolean Equation Systems* (BESs) on parallel, shared memory systems, i.e., utilizing state-of-the-art multi-core and many-core processors – though not in a hybrid setting. Our goals are (i) to evaluate the scalability of our parallel approach with respect to an increasing number of parallel *processing units* (PUs), and (ii) to prove its competitiveness in comparison with an optimized sequential algorithm, which we implemented analog to the description in [1].

Motivation. Hardware manufacturers no longer increase clock rates, but the number of available PUs of modern processors. Along with the evolving trend towards massively parallel, throughput oriented hardware architectures [13], these developments have led to an ever increasing interest in the parallelization of software. This trend has already found its way into the field of software verification and model checking years ago [4, 16, 17] and must to be considered further in order to be able to push the limits of verification techniques further towards industrial strength, allowing one to deal with larger state spaces and providing rapid feedback to developers.

Today's processors can be divided into two main branches: (i) CPU-based multi-core processors with up to tens of cores and (ii) GPU-based many-core processors with up to several hundreds of cores. The key differences are, (i) the ability to efficiently deal with control flow at the expense of lower data throughput, and (ii) the ability to provide high data throughput rates at the expense of a lack of efficient, control flow guided execution.

We assume the current trend to continue – see e.g., Intel's "Terra Scale Computing"[3] project – suggesting future hardware to consist of more, yet simpler PUs. With respect to parallel algorithms this hardware development favors approaches that are geared towards the *single instruction multiple data* (SIMD) paradigm, as they can most easily take advantage of this type of parallel hardware. Therefore it is inevitable to consider the applicability of massively parallel, SIMD-based (i.e., many-core) systems in our experiments.

Background. The standard model checking problem [8], $M \models \varphi$, can be encoded by a BES [23], where the solution of the BES is equivalent to the solution of the underlying model checking problem. The BES is obtained by the combination of a *Labeled Transition System* (LTS), corresponding to M, and a property φ (e.g., deadlock freedom) that is to be checked for this LTS. Consequently, the data dependencies within the resulting BES are closely related to the structure of the LTS from which it was generated from. For our evaluation we rely on the well established VLTS benchmark,[4] which provides 40 LTSs – originating from academia and industry – that can be checked for deadlocks and livelocks, i.e., our resulting benchmark suite consists of a total of 80 BESs.

The average branching factor, i.e., the average number of outgoing edges per vertex, over all 40 LTSs in the benchmark is 5.73. With respect to parallelization, this number can be interpreted as an upper bound for the potential parallelism that is inherent to an LTS, as in our setting information needs to be propagated along edges. For work-set-based producer-consumer parallelizations this means that (i) for each work item processed only few new work items are expected to be added to the work-set, and (ii) synchronization is needed for concurrent operations on the dynamic data structure used to store the work items.

Due to this, our approach is not based on the producer-consumer paradigm, but on a fixed point iteration. This promises a much higher potential for the utilization of parallel hardware as it does not require dynamic data structures. In our particular setting, data operations can even be implemented lock-free, as the fixed point iteration is used to solve a monotonic function. Furthermore we do not have to populate a work-set as we propagate all possible changes during an iteration, at the price of computational overhead, which is negligible considering the ever growing number of parallel PUs.

Cilk Plus and CUDA. Our approach is based on data-parallelization, which is commonly referred to as "fine grained" parallelization (in contrast to task-parallelization, i.e., "coarse grained" parallelization). In order to efficiently par-

[3] http://techresearch.intel.com/ResearchAreaDetails.aspx?Id=27
[4] http://www.inrialpes.fr//vasy/cadp/resources/benchmark_bcg.html

allelize this type of problem the choice of framework is very important, as it most significantly influences the overhead connected to context switches. In case of our multi-core parallelization the overhead of manual thread maintenance is not negligible, since the amount of productive work per thread invocation is very limited. Therefore, the direct use of multi-threading environments, such as PThreads [26], is very likely to nullify the gain we expect from the parallelization itself. For this reason we chose Intel's Cilk Plus framework,[5] which offers a work stealing based thread-pool and internally employs efficient scheduling and load balancing mechanisms. The scheduling of workers is not explicit and more lightweight than the manual management of threads.

For general purpose programming on GPUs, NVIDIA's *Compute Unified Device Architecture* (CUDA)[6] is the de facto standard framework for parallel computation. It provides an *Application Programming Interface* (API), allowing the utilization of NVIDIA's GPUs for massively parallel, throughput oriented applications beyond the scope of rendering graphics. As the CUDA framework is tailored to applications with many data-parallel threads, light-weight computations per thread and frequent context switches [13], it is well suited for our application.

Contributions and Related Work. In the area of model checking the sizes of input problems become exceptionally large. For this reason, a lot of research has been put into the development of techniques that can reduce the problem sizes by, e.g., applying abstractions, using efficient data structures such as *Binary Decision Diagrams* (BDDs), or limiting the exploration of the problem domain only to relevant parts.

In contrast to this, our approach does not aim at a reduction of the problem size, but even favors large problems in order to exploit modern parallel hardware and thereby increasing the performance of model checking. In the context of this paper we want to restrict ourselves to algorithms and tools related to the solution of the (alternation-free) μ-calculus [19], along with their parallelization and/or the resolution of BESs. For completeness sake we want to mention two sequential tools, namely "evaluator", which we used to generate random BESs and the BESs from the benchmark suite, and "bes_solve", which we used to verify our results – both tools are part of the CADP tool set [24].

Furthermore, we restrict the scope of this paper to the evaluation of the solution step in the model checking process, as there exist several efficient and even parallel approaches for the construction of compact data representations in our setting [3, 4, 20], which can be used for the preprocessing of the input data.

Only two approaches based on the parallel resolution of BESs, are known to us. The first one [28] is based on a multi-core parallelization of the "Gaussian Elimination" as proposed in [23], which turns out not to be viable in practice, due to its exponential space complexity. The second one [18], is tailored to distributed systems aiming at the resolution of extremely large BES instances. There exists

[5] http://software.intel.com/en-us/articles/intel-cilk-plus/

[6] http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf

further distributed implementations in the general context [6, 14, 16, 22], but their general goal, in contrast to our approach, is to increase the total amount of memory in order to deal with larger problem instances, rather than to improve on their run-time performance, as network latency generally degrades the overall performance significantly.

The experimental evaluation of the parity game based approach presented in [27], which performs a parallel resolution of μ-formulae on shared-memory multi-core systems, provides scalability results for up to eight workers. Yet, the range of examples is restricted to three *Sliding Window Protocol* (SWP) and two randomly generated instances and their run-times are not related to existing sequential algorithms. We, however, present a parallel, shared-memory model checking approach that is based on a fixed point iteration used for the parallel resolution of BESs (cf. Sec. 3). Even though this approach is targeted at large BES instances, we are not only concerned about the capability to check large models, but also the improvement of run-time performance. The evaluation of our multi-core implementation confirms the scalability results presented in [27], extends them to a much larger set of different benchmark examples and, most importantly, puts them in relation to an optimized sequential BES solver (cf. Sec. 2). In addition, we show that our approach also scales on many-core architectures, boosting the run-time performance by one order of magnitude, outperforming the optimized sequential baseline significantly (cf. Sec. 4).

2 Fixed Points and Boolean Equation Systems

Fixed Points and the μ-calculus. Fixed points may be used to express temporal properties, such as liveness (i.e., something good will eventually happen) and safety (i.e., something bad will never happen). The following intuition describes the meaning of the least (μ) and greatest (ν) fixed point operators in the context of temporal logic based model checking: μ is used to express liveness properties with the initial assumption that every state violates this property and ν is used to express safety properties with the initial assumption that every state satisfies this property.

The μ-calculus [19] is a powerful formalism, e.g., subsuming the temporal logics LTL, CTL and CTL* [11], for expressing temporal properties. It is defined by the following grammar:

$$\varphi ::= \top | \bot | X | \neg\varphi | \varphi \wedge \varphi | \varphi \vee \varphi | [a]\varphi | \langle a \rangle \varphi | \nu X.\varphi | \mu X.\varphi$$

where Var is a set of propositional variables with $X \in Var$ and Act is a set of actions with $a \in Act$. In our setting μ-formulae are used to express properties over LTSs as exemplary depicted in Fig. 1.

Boolean Equation Systems. BESs are sets of equations, resembling monotonic functions over the Boolean lattice $\{\mathit{false} < \mathit{true}\}$, of the form: $\sigma X = \varphi$.

Here, the *left hand side* (LHS) X is a Boolean variable from the set of propositional variables χ, $\sigma \in \{\mu, \nu\}$ are the least and greatest fixed point operators, and the *right hand side* (RHS) is of the form $\varphi ::= \top | \bot | X | \varphi \wedge \varphi | \varphi \vee \varphi$.

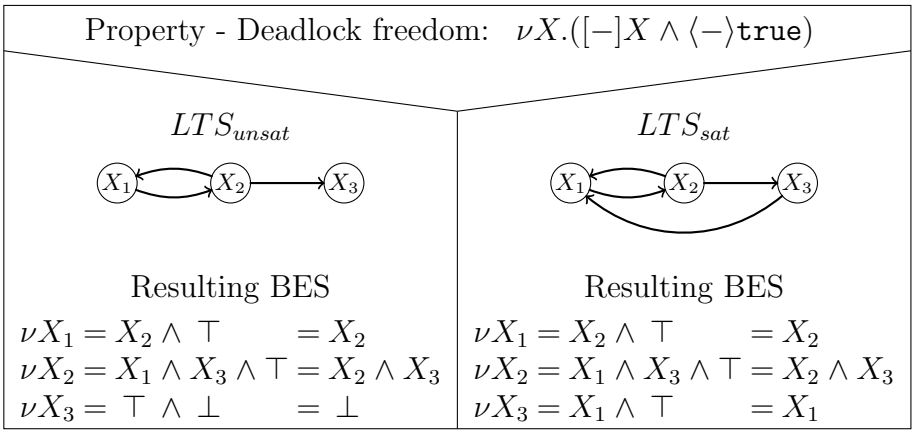


Fig. 1. Interpretation of μ-formula over LTSs.

In the context of model checking, BESs are the result of the interpretation of a μ-formula over an LTS (cf. Fig. 1). Since the formula has to be verified for every state of the LTS, the resulting BES is of size $|LTS|$ x $|\varphi|^k$, i.e., the size of the BES is proportional to the size of the LTS and exponential in the complexity of the μ-formula, where k is the number of alternations of different fixed point operator types binding the same variables. Each fixed point operator of the formula is resembled by a so called *block* in the resulting BES, containing the set of equations associated with this operator.

While equations may be reordered arbitrarily within a block this is not the case for the ordering of blocks as it may lead to the computation of a wrong fixed point. The order in which blocks have to be processed is defined by their nesting within the μ-formula, namely they have to be solved "from inside out". In practice this is not of importance, as – even though with increasing nesting depth μ-formulae become strictly more expressive – it often suffices to consider formulae up to an alternation depth of two [7], which are equally expressive as and may be translated into CTL* [10].

Optimized Sequential Resolution of BESs. To be able to conduct a fair evaluation of our parallel implementations in terms of run-time competitiveness, we have implemented an optimized, sequential CPU-based algorithm, in style of the "chasing ones" as proposed in [1]. This approach is work-set-based, using a queue to store work items, where a work item is equivalent to one equation of the BES. The computation in this algorithm starts at those equations, where the LHS is directly assigned the value *true* or *false*, and propagates this information to all equations relying on the value of these particular LHSs. For this purpose equations must be enriched with information about such backward dependencies. As space and time complexity of this approach are linear in the size of the BES, it is well suited as a baseline for comparison with our parallel implementations.

3 Basic Algorithm and Parallelization

While a lot of effort has been put into the development and optimization of sequential model checking algorithms in order to fight computational complexity and state space explosion, our aim is to investigate whether a parallel approach can be more efficient and provide scalability not only on multi-core (CPU) architectures but also on many-core (GPU) architectures. For this purpose, we chose a fixed point iteration based algorithm, which we show to be well suited for such a parallelization. In this section we first present the algorithmic background of our approach, followed by the concepts of our parallel implementations.

Basic Fixed Point Algorithm. The listing of Algorithm 1 illustrates the fundamental idea of the fixed point computation we employ for the resolution of BESs in our multi-core and many-core implementations.

Algorithm 1: FIXEDPOINT algorithm

```
   Input  : BES
   Output: Solution of BES
1  Initialization of LHSs                    // true for σ = ν; false for σ = μ
2  foreach block B do                                    // block order matters
3      do
4          variablesChanged ← false
5          foreach equation E ∈ B do          // equation order does not matter
6              LHS ← evalRHS(E)
7              if LHSChanged then
8                  variablesChanged ← true
9      while variablesChanged
```

It consists of two nested loops, the outer one over the BES blocks (line 2) and the inner one over all equations within a block (line 5). The inner loop computes the value of the LHS of an equation according to the evaluation of its respective RHS, where the RHS either consists of a terminal value (i.e., *true* or *false*) or LHS variables connected by Boolean operators. In the beginning, all LHSs are initialized depending on their associated fixed point operator σ, as *false* in case $\sigma = \mu$ and *true* in case $\sigma = \nu$ (line 1). This *initial approximation* is derived from the Knaster-Tarksi fixed point theorem [29], where $\mu f = \bigsqcup\{f^i(\mathit{false}) : i \in \mathbb{N}\}$ and $\nu f = \bigsqcap\{f^i(\mathit{true}) : i \in \mathbb{N}\}$. The termination of the fixed point computation is detected by a marker variable, indicating whether one or more LHSs have changed during an iteration (line 9).

Parallel Fixed Point Computation. The core idea for the parallelization of the basic fixed point algorithm is the parallel execution of the inner loop of Algorithm 1 (line 5), computing the LHS value of an equation. It is important to note that the order in which equations are evaluated does not matter within

the loop, as our parallel frameworks are not aimed at the explicit scheduling of threads. Considering the fact that this operation needs be executed for all equations during each iteration step, this approach exposes much potential for parallel computation, even within one iteration step, as we expect the number of equations to be very large, e.g., the largest LTS in the benchmark contains 33,949,609 states. The soundness of the approach is guaranteed by the fact that BESs resemble monotonic functions, i.e., even if the evaluation of a RHS depends on several other LHS variables – which in a parallel setting are potentially modified concurrently – the updated value of each LHS is available and thus can be propagated in the subsequent iteration.

Multi-Core Data Structure. Data structures for multi-core systems have to follow two main objectives. On one hand they have to provide good data locality, i.e., data necessary for a computation should be closely grouped so it can, ideally, be stored in the same cache line of a CPU. On the other hand, unrelated data should be separated in such a way that it does not interfere with each other in order to avoid harmful effects, such as cache thrashing, where independent data sets depend on and thus compete for the same cache lines. Due to these two factors and the structure of our input data (variable(s) $\in$ equation(s) $\in$ block(s) $\in$ BES) we have decided to use a nested data structure, where each aforementioned component is modeled by a structured type. In this layout, all data needed to evaluate one equation – the most frequent operation in our algorithm – is stored in a single structure resembling an equation, thus, accounting for good data locality. Clearly, this also provides good separation and any further improvement would require machine dependent optimization.

Multi-Core Parallelization. For the parallelization of Algorithm 1 on CPUs we employ the Cilk Plus framework provided by the Intel C/C++ compiler. We chose Cilk Plus because it is well suited for problems with fine grained data-parallelism and irregular structure, as shown in [12], which also is the case in our setting. Cilk Plus maintains a pool of workers, each of which is mapped to a thread during execution, that supports work stealing, i.e., taking over work that was initially assigns to another worker. This is in contrast to having to create, manage and delete threads manually, inducing a much higher overhead.

The key idea of our multi-core implementation is the parallelization of the inner for loop, iterating over the equations, by employing Cilk Plus' parallel version of a for-loop, *cilk_for*. The reasons why we do not require any locking and further modifications are (i) the monotonicity of the Boolean function, as mentioned before, and (ii) the fact that the variable *variablesChanged* indicating a change of LHSs is only reset outside the parallel loop (Algorithm 1, line 4) and set uniformly (only to *true*) inside the parallel loop (Algorithm 1, line 8), i.e., any worker that has observed a changing variable assigns this value, and thus the value cannot become inconsistent.

Many-Core Data Structure. Data structures used for CUDA accelerated computation must be specially designed for this purpose. They must support independent thread-local data processing, and, at the same time, they must

also be compact enough to enable good data locality. This is to avoid high latency device-memory access and generally reduce the usage of device-memory bandwidth, which may otherwise become a performance bottleneck [21].

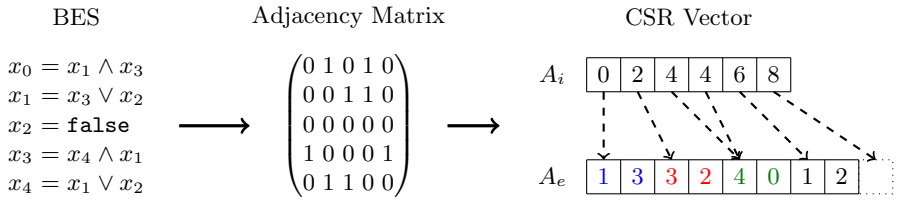


Fig. 2. Generation of adjacency list representation from BES.

A BES may be interpreted as a directed graph where the LHSs are vertices and the dependencies on the RHSs are edges. Such a graph can be encoded as an adjacency matrix and stored using two vectors in *compressed sparse row* (CSR) format, as depicted in Fig. 2. Sine this data structure has been demonstrated to be efficient for graph based algorithms in the context of CUDA accelerated computation [2, 5, 15] we employ it to store BESs. Each vertex stores the following information: a unique index, its Boolean value along with a flag indicating whether the Boolean value is already computed, and the *type* of Boolean operator (conjunction or disjunction).

In more detail, our representation uses two one-dimensional arrays A_i and A_e to encode the directed graph. For all vertices v_0 to v_n, the sum of outgoing edges is stored in A_i, such that the number of outgoing edges from a particular vertex v_j can be computed by $A_i[j+1] - A_i[j]$. The idea of this encoding is that the value of an element $A_i[j]$ serves as an index to the second array A_e. The array A_e is a concatenation of ordered lists of target vertices of outgoing edges from individual graph vertices.

The sizes of the arrays A_i and A_e correspond to the sizes of the vertex set and edge set of the graph, respectively. The array A_i not only stores the indices to the array A_e but also the aforementioned information (index, Boolean value, flag and type). As the on-board memory of GPUs is very limited, we store this additional information in unused bits of A_i, reducing the space requirement to 4 bytes per vertex.

Many-Core Parallelization. For our many-core parallelization we employ the CUDA framework, in which programs consist of two parts (i) *host code* running on the CPU and (ii) *device code* running on the GPU, the so called *kernels*. A kernel is executed concurrently in many independent data-parallel threads, where a group of threads, called a *warp*, executes on the same processor in a lock-step manner. When several warps are scheduled on a processor, memory latencies and pipeline stalls are hidden by switching to the execution of another

Algorithm 2: FIXEDPOINT kernel – run in parallel for every LHS variable

```
   Input  : g(lobal)A_e, g(lobal)A_i, fixedPointFound
 1 tid ← blockId.x * blockDim.x + threadId.x
 2 myVertex ← gA_i[tid]
 3 if myVertex.solved then
 4     return
 5 first ← myVertex.index
 6 last ← gA_i[tid + 1].index
 7 foreach index ∈ first, ..., last do
 8     targetVertex ← gA_e[index]
 9     mySucc ← gA_i[targetVertex]
10     if mySucc.value ≠ myVertex.type then        // type ∨ ≡ 0 and type ∧ ≡ 1
11         break
12 if myVertex.value ≠ mySucc.value then
13     myVertex.solved ← true
14     myVertex.value ← mySucc.value
15     gA_i[tid] ← myVertex
16     fixedPointFound ← false
```

warp. The CUDA framework is optimized for large numbers of simple parallel computations without explicit scheduling of threads.

For this reason the work-flow of our CUDA accelerated fixed point computation is divided into two parts. The host code, executing on the CPU, iterates over the outer loop, i.e., the loop over all BES blocks, and calls the CUDA kernels executing on the GPU from within this loop. Each of the kernels is computing the solution for one LHS, i.e., evaluating one RHS. The CUDA kernel is invoked as long as LHSs change. Its pseudo code is provided in Algorithm 2.

This approach exposes fine grained data-parallelism, requiring a dedicated thread to be executed for every vertex (LHS) of the graph (each item of Array A_i). Each thread first loads the data of a vertex from Array A_i (stored in global memory) into a local copy (line 2) and checks if the corresponding LHS has already been solved (line 3). Then, it processes all immediate successors (loop on line 7), representing the RHS of the corresponding equation. The algorithm employs a lazy evaluation of the equations. In case that a value within a RHS immediately determines the value of the LHS (i.e., the RHS is a purely disjunctive term where at least one variable is *true*, or a purely conjunctive term and at least one variable is *false*), the loop is broken (line 11). Finally, the Boolean value of the evaluation of the RHS (stored in *mySucc.value*) is compared to the Boolean value stored in the corresponding LHS (line 12). If the two values differ the result of the evaluation is assigned to the respective LHS, written back to Array A_i (line 15), and the fixed point flag is set to *false* indicating that the fixed point is not yet reached.

Many-Core Optimizations. For the GPU-based implementation we have experimented with two optimizations.

The first one is the so called "intra-warp fixed point iteration." It is based on the observation that all threads within a warp have to load the required data from global memory into local copies. All operations are performed on the local copies, which are written back to global memory at the end of the execution of the warp. This means that updated LHSs do not become visible to other threads until the next iteration step and, thus, changes can only be propagated one step per iteration. The intra-warp fixed point iteration is intended to increase the number of propagations by performing multiple iterations on the equations bundled in a warp and thereby propagating changes of LHSs within this warp.

The second optimization is an extension to the aforementioned intra-warp fixed point iteration. It utilizes the GPU's shared memory, which provides a fast local memory for single thread or warp, allowing the intermediate storage of data. We use this shared memory to optimize the execution of the kernel by copying the LHS variables contained in a RHS from global memory to shared memory. When the data of a LHS is required by the kernel, the copy in shared memory is utilized instead of the one in global memory. When the kernel returns, the copy is written back from shared to global memory. However, the indirection on line 9 potentially requires further LHSs; this data can either be read from global memory as before or also be copied to shared memory. This reduces access to global memory but requires additional load and store operations before and after each thread invocation.

4 Experimental Evaluation

In this section we experimentally evaluate the scalability of our parallel approach in its CPU and GPU variants and demonstrate the competitiveness of the GPU version when compared to the optimized sequential algorithm, using the VLTS benchmark suite.[7] In order to provide an outlook on the generality of our results, extend this evaluation using randomly generated BESs. Furthermore, we evaluate the structure and density of the BESs generated from the benchmark suite. Besides the run-time based comparison we provide important insights to the specifics of BESs in the context of model checking, i.e., we present heuristics for the order in which equations are to be solved that yield significant speed-ups for BESs resolution in this context.

Benchmark Suite. Our experiments were conducted using the VLTS benchmark suite that was compiled within a joint project of CWI[8] and INRIA[9]. It consists of 40 examples from academia and industry, provided as LTSs with numbers of states ranging from 289 up to 33,949,609. The four largest examples of the benchmark were solved for the first time in 2005 [16].

The background of the benchmark examples varies greatly; thus, different properties could be checked for individual examples. For our evaluation we use

[7] http://www.inrialpes.fr//vasy/cadp/resources/benchmark_bcg.html
[8] http://www.cwi.nl/
[9] http://vasy.inria.fr/

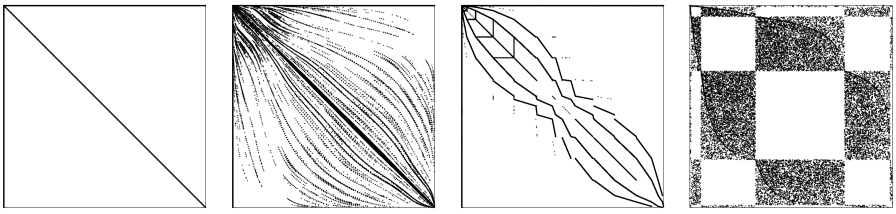
Table 1. μ-Formulae of Properties

Property	μ-formula
Deadlock freedom	$\nu X.([-]X \wedge \langle - \rangle true)$
Livelock	$\mu X.(\langle - \rangle X \vee \nu Y.(\langle \tau \rangle Y))$

two representative properties, namely deadlock freedom and livelock, which can be checked for all examples of the benchmark suite (cf. Table 1 for their formalization). For these properties results are also provided by the authors of the benchmark, thus allowing a direct verification of the correctness of the results of our implementations.

The images of some exemplary BESs, as depicted in Fig. 3, show the significant variance in structure and density of the LTSs provided in the benchmark. The images are visualizations of the adjacency matrices of the respective BESs, with the origin, i.e., the LTSs initial state, on the top left.

In contrast to intuition, our experiments suggest that this information about structure and density does not usefully correlate with the scalability and/or runtime performance of our approach. This is the case for the following reasons: (i) the run-time generally depends on the question whether the property, which the LTS is checked for, is fulfilled or violated; (ii) the fact that our approach does not favor local propagation of changing variables, but globally propagates all possible changes during an iteration; (iii) the fact that our approach performs best in cases that expose large numbers of concurrent changes rather than sequential chains of changes. Unfortunately, none of these factors can be estimated sensibly and extracted from a BES's structure.



(a) Example 10 (b) Example 22 (c) Example 34 (d) Rnd2

Fig. 3. Visualization of benchmark examples as adjacency matrices.

Hardware. Our experiments were carried out on different hardware platforms for (i) the CPU and (ii) the GPU version of the implementation: (i) Two interconnected Intel XEON E7-4830 Processors @ 2,13 GHz, each with 8 physical cores and Hyper-Threading enabled (i.e., a total of 32 logical PUs) and 64 GB DDR3 RAM @ 1333 MHz, running Windows 7 64-bit, and (ii) one AMD Phenom II X4 940 Processor @ 3,0 GHz, 8 GB DDR2 RAM @ 1066 MHz along with (a) one

NVIDIA GeForce GTX 280 GPU with 1 GB of global memory, 16KB of shared memory per multiprocessor, providing 240 CUDA cores, and (b) one NVIDIA GeForce GTX 480 GPU with 1.5 GB of global memory, 48KB of shared memory per multiprocessor, providing 480 CUDA cores, running Debian 6.0 64-bit on kernel 2.6.39.

Although the systems use different CPU types this fact does not affect our results as we did not evaluate a hybrid approach, but only pure CPU and GPU versions of the respective algorithms.

Table 2. Overview of Run-Times for CPU- and GPU-based implementations [ms]

Algorithm		Benchmark Example									Random	
		10	21	22	31	32	33	34	35	39	Rnd1	Rnd2
CPU	(i) sequential	1	19	18	573	475	737	1	704	901	3891	7801
	(ii) parallel	2538	77	611	1564	1786	2764	279	4325	8170	7966	40576
GPU GTX 280	(iii) unoptimized	1336	17	68	217	113	359	51	242	290	350	1840
	(iv) intra-warp	104	22	69	320	149	528	52	404	344	493	2594
GPU GTX 480	(iii) unoptimized	703	6	33	75	46	105	6	98	125	178	992
	(iv) intra-warp	40	7	28	109	63	157	6	152	158	248	1391
	(v) shared mem	38	40	59	659	341	862	48	190	227	315	1800

Overview. Table 2 provides an overview of the run-times of the following algorithms: (i) the optimized sequential work-set-based CPU implementation (the baseline for our comparison), (ii) the parallel Cilk Plus based CPU implementation, (iii) the unoptimized GPU implementation without any optimization, (iv) the GPU implementation with intra-warp iteration, and (v) the GPU implementation utilizing shared memory. In case of the GTX 280 GPU, we omitted the results for (iv), the shared memory implementation, since this GPU does not provide a sufficient amount of shared memory for this optimization. Note that in the case of parallel CPU implementation we list the best runtimes available among the numbers of cores that have been utilize.

Because of space limitations, we restrict selection of benchmark examples in Table 2 to those for which the run-time of the GPU implementation is sensibly measurable, i.e., larger than 5 [ms]; nonetheless we conducted our experiments for the entire benchmark suite. The numbering of the benchmark examples refers to their position in the table provided on the VLTS website[10], which is sorted in ascending order relative to the number of states of the LTS, thus, Example 10 is *vasy_25_25*, Example 21 is *vasy_166_651*, Example 22 is *cwi_214_684*, Example 31 is *vasy_2581_11442*, Example 32 is *vasy_4220_13944*, Example 33 is *vasy_4338_15666*, Example 34 is *vasy_6020_19353*, Example 35 is *vasy_6120_11031* and Example 39 is *vasy_12323_27667*. In this naming scheme

[10] http://cadp.inria.fr/resources/benchmark_bcg.html#section-5

the first number is the number of states divided by 1000, and the second number is the number of transitions divided by 1000.

Furthermore, all examples in Table 2 are checked for the deadlock freedom property as only eight of the 40 LTSs contain livelocks. Nonetheless, our general statements about scalability and competitiveness have been evaluated and are valid for the entire benchmark suite. In order to extrapolate our results and obtain more insight to the specific structure of the benchmark's LTSs, we extend our evaluation to randomly generated BESs. We evaluate a total of five examples with the number of states ranging from 1 to 10 million; Rnd1 and Rnd2 are two representatives illustrating our observations for this class of BESs.

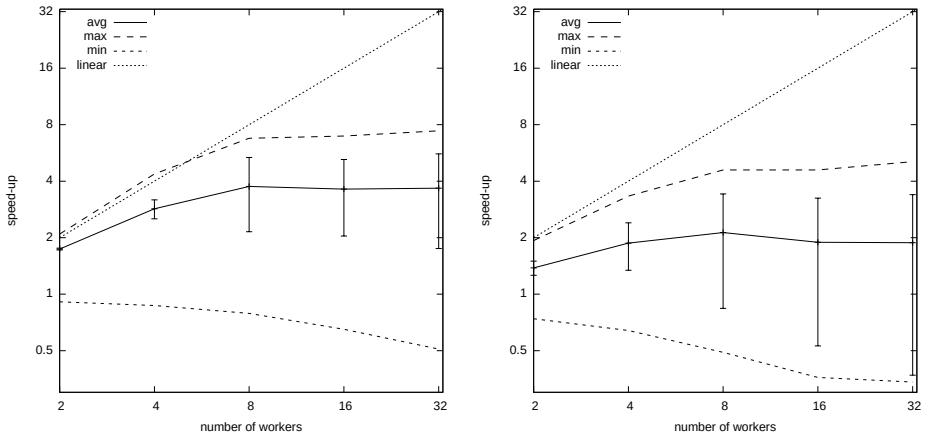
We omit memory consumptions of our implementations in the table, as (i) our parallel versions operate on a static data structure that is linear in the size of the input BES (ranging from approximately 90 KB up to 4.5 GB) and (ii) it is not our aim to evaluate or optimize memory efficiency in the scope of this paper, especially since all benchmark examples easily fit our systems memory.

Multi-Core Performance. The results in Table 2 clearly show that our multi-core implementation is outperformed significantly by the optimized sequential baseline. The reason for this is the low total number of parallel PUs (32 logical cores) and thus, the computational overhead of the fixed point iteration is too large compared to the amount of productive work and cannot be compensated by parallel processing power. This observation is supported by the two graphs in Fig. 4, which show the overall scalability of our CPU-based approach for an increasing number of parallel workers. This result is in accordance with [27] and extends their results to our much larger benchmark suite.

The data for the two graphs in Fig. 4 is averaged over all 40 benchmark examples, but separately evaluated for the two properties: deadlock freedom (Fig. 4(a)) and livelock (Fig. 4(b)). It shows that the average scalability is below linear for both properties, but observable for up to eight workers, which corresponds to the number of physical cores of one CPU in our system. It is important to remark that the shape of the two graphs, suggesting better scalability for LTSs that have been checked for the no deadlock property, is affected by the fact that there are 20 examples containing deadlocks, while only 8 examples contain livelocks. In the case of the trivial examples, i.e., those that do not contain deadlocks/livelocks, the algorithm needs to perform only one iteration, which has the significant impact on the scalability.

The super-linear speed-up in Fig. 4(a) can be explained by the parallel execution of workers. As the Cilk Plus framework may schedule the evaluation order of equations differently from the original one in the BES, this may lead to a faster propagation of updated LHSs, requiring less iterations and thus result in the seemingly above linear boost in performance.

Many-Core Performance. The evaluation of our many-core implementation is aimed more at the competitiveness of our approach when compared to the optimized sequential baseline than at its scalability. The scalability analysis of our many-core implementation is much more difficult than for the multi-core implementation as we had to use different GPU devices that are not comparable



(a) Deadlock freedom (b) Livelock

Fig. 4. Scalability of our multi-core implementation.

with respect to some important specifications. Not only did the number of CUDA cores double from the GTX 280 to the GTX 480, but also the clock rate and the available amount of memory increased significantly. For this reason we did not evaluate the scalability aspect beyond the scope provided in Table 2, which shows a significant boost in performance for the GTX 480. Further evaluations of scalability, e.g., on clusters of GPUs, are subject to future work.

The main limitation of the GPU parallelization is the length of the chain of propagations of LHS values. The example 10 in the benchmark suite contains an artificially long chain of dependencies from the initial state to the last state (cf. Fig. 3(a)). For this example, the number of iterations is equal to the number of states for the unoptimized version of our many-core implementation, yet it is a prime candidate to benefit from the intra-warp iteration as the changes can be propagated ideally within the equations of a warp.

However, the remaining benchmark examples do not have such an extreme structure and therefore the intra-warp iteration, on average, does not provide any advantage, but rather induces overhead as the comparison of run-times in Table 2 shows.

Since the efficiency of our shared memory optimization is tightly coupled to the intra-warp iteration, it can only improve the performance of the many-core implementation in those cases in which the intra-warp iteration actually works. Due to this reason, the results for this optimization in Table 2 are, not surprisingly, even worse than for the intra-warp iteration because the transfer times from and to shared memory degrade the run-time performance even further. Moreover, in order to use the shared memory, the required data (the part of a BES corresponding to a block) has to fit the limited size of the GPUs shared memory. The size of the data that has to be stored in the shared memory is given by the block size - a number of vertices in one block and by the number

of their successors. In the case the average out-degree (the average number of RHS variables per equations) is high, we have to decrease the group size. This can lead to underutilization or low occupancy of the individual multiprocessors and thus significantly reduces the performance of the algorithm.

As documented in Table 2 we have shown that our GPU-based implementation of the resolution of BESs provide significant speed-ups for most cases of the benchmark examples and especially for the randomly generated BESs. Surprisingly, the GPU implementation with no optimizations yields the best results, since in most of cases the structure of the inspected BESs does not allow to benefit from the designed optimizations.

Impact of Ordering Heuristics on the Number of Iterations.

Table 3. Impact of Heuristics [Total Number of Iterations]

Heuristic	Benchmark Example																			
	4	5	7	10	15	16	18	19	21	22	25	27	30	31	32	33	35	37	38	39
Forward	64	19	7	25219	19	33	23	18	33	208	24	7	56	32	23	34	33	20	29	29
Vectorized	64	19	7	25219	23	37	23	19	37	213	24	7	56	37	25	37	34	20	29	29
Reverse	2	4	3	2	7	8	8	5	8	8	10	2	3	7	4	6	4	5	5	5
Random	20	9	5	25219	10	12	6	11	11	63	7	3	6	8	5	8	16	10	9	9

Table 3 provides a comprehensive overview of the total number of iterations for those examples of the benchmark, that have been checked for deadlocks, and for which the initial approximation is not equal to the final solution, i.e., the total number of iterations is larger than one. Even though the available number of PUs continuously increases with each hardware generation, it is still far from the point where a full iteration step can be computed completely in parallel. Thus, the order of equations processing within a block has a significant influence on the total number of iterations needed to compute the fixed point. Yet, our evaluation yields an interesting insight for an ideal "vectorized" parallelization, assuming that a fully parallel iteration step is possible; we model this by delaying the visibility of a changed LHSs until the next iteration step. Our evaluation shows that this limitation does not increase the total number of iterations significantly, when compared to the "original" ordering, where equations are evaluated in their initial order and changes of LHSs are directly visible in the following computations of the iteration (cf. Table 3). This result shows that the penalty for a fully parallel computation is negligible, with respect to the total number of iterations needed to reach the fixed point.

As the application of advanced heuristics would require preprocessing of the data – causing a potentially high computational overhead – we restrict our evaluation to two simple cases that do not introduce any overhead. The first heuristic to carry out the evaluation of equations within a BES-block, namely taking the reverse order as proposed in [27], yields a significant improvement with respect

to the total number of iteration needed to compute the fixed point (cf. Table 3). Yet, according to our observations, this heuristic only works for the examples generated from the benchmark's LTSs, but not for randomly generated BESs.

The second heuristic is the randomized evaluation of equations within a BES-block. In our observations this heuristic leads to a decrease in the number of iterations needed to solve a BESs when compared to the "original" ordering. This result is of practical relevance as our parallel implementations rely on parallelizations in which the order of RHS evaluations is not under our control, but it is determined by the runtime environment of CUDA and Cilk Plus. Thus, we expect an additional performance boost rather than a degradation, due to the parallelization frameworks.

5 Conclusions and Future Work

We have implemented and evaluated an approach to the parallel resolution of BESs on multi- and many-core systems, with respect to scalability and run-time performance in comparison to an optimized sequential algorithm. Our results of the experimental evaluation confirm the scalability results from [27] for the multi-core implementation, yet its overall performance is not competitive, compared to our optimized sequential implementation. The utilization of many-core hardware yields a significant speed-up and can outperform the optimized sequential implementation for most instances of the benchmark by almost an order of magnitude. Furthermore, the scalability of our approach, with respect to increasing numbers of PUs, has been shown in our evaluation by (i) comparing the multi-core and many-core implementations and (ii) evaluating the many-core implementation for two GPU cards with 240 and 480 CUDA cores, respectively.

Future work will include further evaluation of the scalability results of the many-core implementation, e.g., by its distribution over a cluster of GPUs. Since BESs are not restricted to model checking, it is also promising to evaluate input BESs from other applications, such as data-flow analysis [9]. Furthermore, the recently proposed many-core parallelization of graph algorithms [25] should be evaluated with respect to its suitability and potential impact on our work.

References

1. H. R. Andersen. Model Checking and Boolean Graphs. *Theoretical Computer Science*, 126(1):3–30, 1994.
2. J. Barnat, P. Bauch, L. Brim, and M. Češka. Computing Strongly Connected Components in Parallel on CUDA. In *IPDPS*, pages 544 –555. IEEE, 2011.
3. J. Barnat, P. Bauch, L. Brim, and M. Češka. Designing Fast LTL Model Checking Algorithms for Many-Core GPUs. *To app. in J. of Par. and Distrib. Comp.*, 2012.
4. J. Barnat, L. Brim, and P. Ročkai. Scalable Multi-core LTL Model-Checking. In *SPIN*, volume 4595 of *LNCS*, pages 187–203. Springer, 2007.
5. J. Barnat, L. Brim, M. Češka, and T. Lamr. CUDA Accelerated LTL Model Checking. In *ICPADS*, pages 34–41. IEEE, 2009.
6. B. Bollig, M. Leucker, and M. Weber. Local Parallel Model Checking for the Alternation-Free μ-Calculus. In *SPIN*, volume 2318 of *LNCS*, pages 128–147. Springer, 2002.

7. J.C. Bradfield. The Modal mu-Calculus Alternation Hierarchy Is Strict. *Theoret. Comp. Sc.*, 195(2):133 – 153, 1998.
8. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.
9. M. d. M. Gallardo, C. Joubert, and P. Merino. On-the-Fly Data Flow Analysis Based on Verification Technology. In *COCV*, volume 190 of *ENTCS*, pages 33–48, 2007.
10. M. Dam. CTL* and ECTL* as Fragments of the Modal mu-Calculus. *Theoret. Comp. Sc.*, 126(1):77–96, 1994.
11. E. A. Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 995–1072. Elsevier Science, 1990.
12. J. Ezekiel, G. Lüttgen, and R. Siminiceanu. To Parallelize or to Optimize? *J.of Log. and Comput.*, 21:85–120, 2011.
13. M. Garland and D. B. Kirk. Understanding Throughput-Oriented Architectures. *Commun. ACM*, 53:58–66, 2010.
14. O. Grumberg, T. Heyman, and A. Schuster. Distributed Symbolic Model Checking for μ-Calculus. *Form. Methods Syst. Des.*, 26:197–219, 2005.
15. P. Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *HiPC*, volume 4873 of *LNCS*, pages 197–208. Springer, 2007.
16. F. Holmén, M. Leucker, and M. Lindström. UppDMC: A Distributed Model Checker for Fragments of the mu-Calculus. In *PDMC*, volume 128 of *ENTCS*, pages 91–105, 2005.
17. G. J. Holzmann and D. Bosnacki. Multi-Core Model Checking with SPIN. In *IPDPS*, pages 1–8, 2007.
18. C. Joubert and R. Mateescu. Distributed Local Resolution of Boolean Equation Systems. In *PDP*, pages 264–271. IEEE, 2005.
19. D. Kozen. Results on the Propositional mu-Calculus. *Theoret. Comp. Sc.*, 27:333–354, 1983.
20. A. Laarman, J. van de Pol, and M. Weber. Boosting Multi-Core Reachability Performance with Shared Hash Tables. In *FMCAD*. IEEE, 2010.
21. A. Lefohn, J. M. Kniss, and J. D. Owens. Implementing Efficient Parallel Data Structures on GPUs. In *GPU Gems 2*, pages 521–545. Addison-Wesley, 2005.
22. M. Leucker, R. Somla, and M. Weber. Parallel Model Checking for LTL, CTL*, and L^2_μ. In *PDMC*, volume 89 of *ENTCS*, pages 4–16, 2003.
23. A. H. Mader. *Verification of Modal Properties Using Boolean Equation Systems*. PhD thesis, Technische Universität München, Bertz Verlag, Berlin, 1997.
24. R. Mateescu. CAESAR_SOLVE: A Generic Library for On-the-Fly Resolution of Alternation-free Boolean Equation Systems. *STTT*, 8(1):37–56, 2006.
25. D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU Graph Traversal. In *PPoPP*, pages 117–128. ACM, 2012.
26. B. Nichols and D. Buttlar J. P. Farrell. *Pthreads programming*. O'Reilly, 1996.
27. J. van de Pol and M. Weber. A Multi-Core Solver for Parity Games. In *PDMC*, volume 220 of *ENTCS*, pages 19–34, 2008.
28. A. Sailer. Utilizing And-Inverter Graphs in the Gaussian Elimination for Boolean Equation Systems. Master's thesis, Hochschule Regensburg, 2011.
29. A. Tarski. A Lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific J. of Math*, 5(2):285–309, 1955.