Scalable Multi-Core LTL Model-Checking*

J. Barnat, L. Brim, and P. Ročkai

Faculty of Informatics, Masaryk University
Brno, Czech Republic
{barnat,brim,xrockai}@fi.muni.cz

Abstract. Recent development in computer hardware has brought more wide-spread emergence of shared-memory, multi-core systems. These architectures offer opportunities to speed up various tasks – among others LTL model checking. In the paper we show a design for a parallel shared-memory LTL model checker, that is based on a distributed-memory algorithm. To achieve good scalability, we have devised and experimentally evaluated several implementation techniques, which we present in the paper.

1 Introduction

With the arrival of 64-bit technology the traditional space limitations in formal verification may become less severe. Instead, time could quickly become an important bottleneck. This naturally raises interest in using parallelism to fight the "time-explosion" problem.

Much of the extensive research on the parallelization of model checking algorithms followed the distributed-memory programming model and the algorithms were parallelized for networks of workstations, largely due to easy access to networks of workstations. Recent shift in architecture design toward multi-cores has intensified research pertaining to shared-memory paradigm as well.

In [10] G. Holzmann proposed an extension of the `SPIN` model-checker for dual-core machines. The algorithms keep their linear time complexity and the liveness checking algorithm supports full LTL. The algorithm for checking safety properties scales well to N-core systems. The algorithm for liveness checking, which is based on the original `SPIN`'s nested DFS algorithm, is unable to scale to N-core systems. It is still an open problem to do scalable verification of general liveness properties on N-cores with linear time complexity.

A different approach to shared-memory model checking is presented in [13], based on CTL* translation to Hesitant Alternating Automata. The proposed algorithm uses so-called non-emptiness game for deciding validity of the original formula and is therefore largely unrelated to the algorithms based on fair-cycle detection.

In this paper we show a design for a parallel shared-memory model checking tool, that is based on a distributed-memory algorithm due to Černá and

* This work has been partially supported by the Grant Agency of Czech Republic grant No. 201/06/1338 and the Academy of Sciences grant No. 1ET408050503.

Pelánek [7]. The algorithm used is linear for properties expressible as weak Büchi automata, which comprises majority of LTL properties encountered in practice. Although the worst-case complexity is quadratic, the algorithm exhibits very good performance with real-life verification problems. To achieve good scalability, we have devised several implementation techniques, as presented in this paper, and applied them to this algorithm.

We expect, that application of the proposed implementation approaches to several other distributed-memory algorithms for LTL model-checking may bring similar improvements in scalability on N-core systems.

In Section 2 we summarize the existing parallel algorithms for LTL model-checking (*accepting cycle detection*). In Section 3 we present several implementation techniques that were applied to multi-core implementation of the selected algorithm. In Section 4 we report on scalability tests and on comparison with dual-core Nested DFS algorithm.

2 Parallel LTL Model-Checking Algorithms

Efficient parallel solution of many problems often requires approaches radically different from those used to solve the same problems sequentially. Classical examples are list rankings, connected components, depth-first search in planar graphs etc. In the area of LTL model-checking the best known enumerative *sequential* algorithms based on fair-cycle detection are the *Nested DFS* algorithm [8, 12] (implemented, e.g., in the model checker `SPIN` [11]) and *SCC-based algorithms* originating in Tarjan's algorithm for the decomposition of the graph into strongly connected components (SCCs) [19]. However, both algorithms rely on inherently sequential depth-first search postorder, hence it is difficult to adapt them to parallel architectures. Consequently, different techniques and algorithms are needed. Unlike LTL model-checking, the reachability analysis is a verification problem with efficient parallel solution. The reason is that the exploration of the state space can be implemented e.g. using breadth-first search. In the following, we sketch four parallel algorithms for enumerative LTL model checking that are, more or less, based on performing repeated parallel reachability to detect reachable accepting cycles. The reader is kindly asked to consult the original sources for the details.

[MAP] The main idea of the **Maximal Accepting Predecessor Algorithm** [4, 6] is based on the fact that every accepting vertex lying on an accepting cycle is its own predecessor. An algorithm that is directly derived from the idea, would require expensive computation as well as space to store all proper accepting predecessors of all (accepting) vertices. To remedy this obstacle, the MAP algorithm stores only a single representative of all proper accepting predecessor for every vertex. The representative is chosen as the *maximal accepting predecessor* accordingly to a presupposed linear ordering $\prec$ of vertices (given e.g. by their memory representation). Clearly, if an accepting vertex is its own maximal accepting predecessor, it lies on an accepting cycle. Unfortunately, it can happen

that all the maximal accepting predecessor lie out of accepting cycles. In that case, the algorithm removes all accepting vertices that are maximal accepting predecessors of some vertex, and recomputes the maximal accepting predecessors. This is repeated until an accepting cycle is found or there are no more accepting vertices in the graph.

The time complexity of the algorithm is $\mathcal{O}(a^2 \cdot m)$, where a is the number of accepting vertices. One of the key aspects influencing the overall performance of the algorithm is the underlying ordering of vertices used by the algorithm. It is not possible to compute the optimal ordering in parallel, hence heuristics for computing a suitable vertex ordering are used. □

[OWCTY] The next algorithm [7] is an extended enumerative version of the **One Way Catch Them Young Algorithm** [9]. The idea of the algorithm is to repeatedly remove vertices from the graph that cannot lie on an accepting cycle. The two removal rules are as follows. First, a vertex is removed from the graph if it has no successors in the graph (the vertex cannot lie on a cycle), second, a vertex is removed if it cannot reach an accepting vertex (a potential cycle the vertex lies on is non-accepting). The algorithm performs removal steps as far as there are vertices to be removed. In the end, either there are some vertices remaining in the graph meaning that the original graph contained an accepting cycle, or all vertices have been removed meaning that the original graph had no accepting cycles.

The time complexity of the algorithm is $\mathcal{O}(h \cdot m)$ where $h = h(G)$. Here the factor m comes from the computation of elimination rules while the factor h relates to the number of global iterations the removal rules must be applied. Also note, that an alternative algorithm is obtained if the rules are replaced with their backward search counterparts. □

[NEGC] The idea behind the **Negative Cycle Algorithm** [5] is a transformation of the LTL model checking problem to the problem of negative cycle detection. Every edge of the graph outgoing from a non-accepting vertex is labeled with 0 while every edge outgoing from an accepting vertex is labeled with -1. Clearly, the graph contains a negative cycle if and only if it has an accepting cycle.

The algorithm exploits the *walk to root* strategy to detect the presence of a negative cycle. The strategy involves construction of the so called *parent graph* that keeps the shortest path to the initial vertex for every vertex of the graph. The parent graph is repeatedly checked for the existence of the path. If the shortest path does not exist for a given vertex, then the vertex is a part of negative, thus accepting, cycle. The worst case time complexity of the algorithm is $\mathcal{O}(n \cdot m)$. □

[BLEDGE] An edge (u, v) is called a *back-level edge* if it does not increase the distance of the target vertex v form the initial vertex of the graph. The key observation connecting the cycle detection problem with the back-level edge concept, as used in the **Back-Level Edges Algorithm** [1], is that every cycle contains at least one back-level edge. Back-level edges are, therefore, used

as triggers to start a procedure that checks whether the edge is a part of an accepting cycle. However, this is too expensive to be done completely for every back-level edge. Therefore, several improvements and heuristics are suggested and integrated within the algorithm to decrease the number of tested edges and speed-up the cycle test.

The BFS procedure which detects back-level edges runs in time $\mathcal{O}(m+n)$. In the worst case, each back-level edge has to be checked to be a part of a cycle, which requires linear time $\mathcal{O}(m+n)$ as well. Since there is at most m back-level edges, the overall time complexity of the algorithm is $\mathcal{O}(m.(m+n))$. □

All the algorithms allow for an efficient implementation on a parallel architecture. The implementation is based on partitioning the graph (its vertices) into disjoint parts. Suitable partitioning is important to benefit from parallelization.

One particular technique, that is specific to automata based LTL model checking, is *cycle locality preserving* problem decomposition [2, 14]. The graph (product automaton) originates from synchronous product of the property and system automata. Hence, vertices of product automaton graph are ordered pairs. An interesting observation is that every cycle in a product automaton graph emerges from cycles in system and property automaton graphs. Let A, B be Büchi automata and $A \otimes B$ their synchronous product. If $\mathcal{C}$ is a strongly connected component in the automaton graph of $A \otimes B$, then A-projection of C and B-projection of C are (not necessarily maximal) strongly connected components in automaton graphs of A and B, respectively.

As the property automaton origins from the LTL formula to be verified, it is typically quite small and can be pre-analyzed. In particular, it is possible to identify all strongly connected components of the property automaton graph. A partition function may then be devised, that respects strongly connected components of the property automaton and therefore preserves cycle locality. The partitioning strategy is to assign all vertices that project to the same strongly connected component of the property automaton graph to the same sub-problem. Since no cycle is split among different sub-problems it is possible to employ localized Nested DFS algorithm to perform local accepting cycle detection simultaneously.

Yet another interesting information can be drawn from the property automaton graph decomposition. Maximal strongly connected components can be classified into three categories:

Type F: (*Fully Accepting*) Any cycle within the component contains at least one accepting vertex. (There is no non-accepting cycle within the component.)

Type P: (*Partially Accepting*) There is at least one accepting cycle and one non-accepting cycle within the component.

Type N: (*Non-Accepting*) There is no accepting cycle within the component.

Realizing that vertex of a product automaton graph is accepting only if the corresponding vertex in the property automaton graph is accepting it is possible

to characterize types of strongly connected components of product automaton graph according to types of components in the property automaton graph. This classification of components into types N, F, and P can be used to gain additional improvements that may be incorporated into the above given algorithms.

3 Implementation Techniques

It is a well known fact, that a distributed-memory, parallel algorithm is straightforwardly transformed into a shared-memory one. However, there are several inefficiencies involved in this direct translation. Several traits of shared-memory architecture may be leveraged to improve real-world performance of such implementations. In this section, we present our approaches at the challenges of shared-memory architecture and its specific characteristics.

3.1 Shared-Memory Platform

We work with a model based on threads that share all memory, although they have separate stacks in their shared address space and a special thread-local storage to store thread-private data. Our working environment is POSIX, with its implementation of threads as lightweight processes. Switching contexts among different threads is cheaper than switching contexts among full-featured processes with separate address spaces, so using more threads than there are CPUs in the system incurs only a minor penalty.

Critical Sections, Locking and Lock Contention. In a shared-memory setting, access to memory, that may be used for writing by more than a single thread, has to be controlled through use of mutual exclusion, otherwise, race conditions will occur. This is generally achieved through use of a "mutual exclusion device", so-called mutex. A thread wishing to enter a critical section has to lock the associated mutex, which may block the calling thread if the mutex is locked already by some other thread. An effect called resource or lock contention is associated with this behaviour. This occurs, when two or more threads happen to need to enter the same critical section (and therefore lock the same mutex), at the same time. If critical sections are long or they are entered very often, contention starts to cause observable performance degradation, as more and more time is spent waiting for mutexes.

Role of Processor Cache. There are two fairly orthogonal issues associated with processor cache. First, cache coherence which is implemented by hardware, but its efficiency is affected by programmer, and cache efficiency, which mostly depends on data structures and algorithms employed.

Cache coherence poses an efficiency penalty when there are many processors writing to same area of memory. This is largely avoided by the distributed algorithm, however, locking and access to shared data structures have no other choice. Cache coherence on modern architectures works at a level of cache lines, roughly 64 byte chunks of memory that are fetched from main memory into cache at once.

Modern mutex implementations ensure that the mutex is the only thing present on a given cache line, so it does not affect other data, and, more importantly, it ensures that two mutexes never share a cache line, which would pose a performance penalty.

Recent development in multi-core platforms deals with cache coherence problem in a different, more efficient manner, namely, by sharing the level two cache among two or more cores, therefore reducing the cache coherence overhead significantly. Yet, with the current state of technology, this still does not mitigate the overhead completely.

3.2 Implementing Algorithms in Shared-Memory

The above considerations bring us to the actual algorithm implementation and the associated techniques we came up with. They are all designed to reduce communication overhead, exploiting traits of shared-memory systems that are not available in distributed-memory environments. Consequently, the main goal is to improve scalability of the implementation, which is inversely proportional to communication overhead and its growth with increasing number of threads. That said, keeping in mind the possibility to scale beyond shared-memory systems, we try to keep the implementation in a shape that would make a combined tool to work efficiently on clusters of multi-CPU machines achievable.

When we venture into a strictly shared-memory implementation, one may pose a question, whether a different approach of using a standard serial algorithm modified to allow parallelisation at lower levels of abstraction would give a scalable, efficient program for multi-CPU and/or multi-core systems. Our efforts at extracting such a micro-parallelism in our codebase have been largely fruitless, due high synchronisation cost relative to amount of work we were able to perform in parallel. Although we intend to do more research on this topic, we do not expect significant results.

In the following sections, we explore the possibilities to build on existing distributed-memory approaches, in the vein of statically-partitioned graphs, reducing the overhead using idioms only possible due to locality of memory.

3.3 Communication

Generally, in a distributed computation, all communication is accomplished by passing messages – eg. using a library like MPI for cluster message passing. However, in communication-intensive programs, or those sensitive to communication delay, using general-purpose message passing interface is fairly inefficient.

In shared-memory, most of the communication overhead can be eliminated by using more appropriate communication primitives, like high-performance, contention- and lock- free FIFOs (First In, First Out queues). We have adopted a variant of the two-lock algorithm – a decent compromise between performance on one hand and simplicity and portability on the other – presented in [17]. Our modifications involve improved cache-efficiency (by using a linked list of memory-continuous blocks, instead of linked list of single items) and only using

a single write-lock, instead of a pair of locks, one for reading and one for writing, since there is ever only one thread reading, while there may be several trying to write.

Every thread involved in the computation owns a single instance of the FIFO and all messages for this thread are pushed onto this single queue. This may introduce a source of resource-contention, where many processes are trying to append messages to a single queue, but considering the message distribution in our system, this turns out to be a negligible problem in practice. With different patterns of communication, a complete lock-free design may be more appropriate (one is given in [17]).

```
type FIFO of T:
    type Node:
        buffer: array of T
        next: pointer to Node
        read, write: integer
    nodeSize: integer (size of buffer)
    head, tail: pointer to Node
    writeLock: mutex
```

Fig. 1. FIFO representation

```
Require: f is a FIFO of T instance, x of type T is an element to enqueue
Ensure: f contains x as its last element
  lock( f.writeMutex )
  if f.tail.write = f.nodeSize then
    t ← newly allocated Node, all fields 0
  else
    t ← f.tail
  t.buffer[t.write] ← x
  t.write ← t.write + 1
  if f.tail ≠ t then
    f.tail.next = t
    f.tail = t
  unlock( f.writeMutex )
```

Fig. 2. FIFO enqueue

Representation and pseudo-code for enqueue and dequeue algorithms are found in Figures 1, 2 and 3, respectively. The correctness, linearizability and liveness proofs as given in [17] are straightforwardly adapted to our implementation and thus left out.

Alternatives to our implementation, which may be more appropriate in different settings, include a ring-buffer fifo implementation (if there is a bound on

Require: f is a non-empty FIFO instance
Ensure: front element of f is dequeued and then returned
  if $f.head.read = f.nodeSize$ **then**
    $f.head \leftarrow f.head.next$
  $f.head.read \leftarrow f.head.read + 1$
  return $f.head.buffer[f.head.read - 1]$

Fig. 3. FIFO dequeue

the amount of in-flight data known beforehand, the ring-buffer implementation may be more efficient) and possibly an algorithm based on swapping incoming and outgoing queues (which could be easily implemented as a pointer swap). The latter gives results comparable to the described FIFO method, although the code and locking behaviour is much more complex and error-prone, which made us opt for the simpler FIFO implementation.

3.4 Memory Allocation

In a distributed computation, every process has simply its own memory which it fully manages. In a shared-memory, however, we prefer to manage the memory as a single shared area, since an equal partitioning of available memory and separate management may fall short of efficient resource usage. However, this poses some challenges, especially in allocation-intensive environment like ours.

First, a naïve approach of protecting the allocation routines with a simple mutual exclusion is highly prone to resource contention. Fortunately, modern general-purpose allocator implementations refrain from this idea and have a generally non-contending behaviour on allocation. However, releasing memory back for reuse is more complex to achieve without introducing contention, in a setting where it is often the case that thread other than the one allocating the chunk tries to release it.

There are known general-purpose solutions to this problem, eg. [16], however they are currently not in widespread use, therefore we have to refrain from the above-mentioned pattern of releasing memory from different than allocating thread, in order to avoid contention and the accompanying slowdown.

The message-passing implementation we employ is pointer-based, in other words, the message sent is only a pointer and the payload (actual interesting message content) is allocated on the shared heap and it may be either reused or released by the receiving thread. Observe however, that releasing the associated memory in the receiving thread will introduce the situation which we are trying to avoid.

We side-step the issue by adding a new communication FIFO to each thread (recall that our communication induces only low overhead and virtually no contention). When a receiving thread decides that the message content needs to be disposed of, instead of doing it itself, sends the message back to the originating thread using the second FIFO. The originating thread then, at convenient inter-

vals, releases the memory in a single batch, having an interesting side-effect of slightly improving cache-efficiency.

3.5 Efficient Termination Detection

Since our algorithms rely on work distribution among several largely independent threads, similarly to a distributed algorithm, we need a specific algorithm for shared-memory termination detection, that would pose minimal overhead and minimal serialisation.

One possible solution is presented in [15], which does not use locking and relies on the system to provide an enqueue-with-wakeup primitive. However, in our system, we have primitives available that support a somewhat different approach: implementation of sleeping/wakeup primitives already relies on locking and we leverage this inherent locking in our termination detection algorithm.

The POSIX threading library offers a mechanism called "condition signalling", which we use to implement thread sleeping and wakeup. A "condition" is a device that allows to be waited-for by its owning thread and "signalling a condition" from another thread will cause the waiting thread to wake up and continue execution. However, this device in itself is race-prone, since the condition may be signalled just before the owner goes to sleep, leading to a deadlock – another signal may never come. Therefore, the condition is always protected by a mutex, which is always locked through the execution of the owner thread and is only atomically unlocked when the thread enters sleep state and atomically reclaimed before waking up.

Since the available mutex implementation allows a lock-or-fail behaviour, as opposed to lock-or-wait which is usually employed for protecting critical sections, we can use the condition device to implement an efficient termination detection algorithm.

Observe, that at any time when a thread is idle, its condition-protecting mutex is unlocked and conversely, whenever the thread is busy, this mutex is locked. So the termination detection algorithm first tries to lock condition mutexes of all worker threads, one by one, using the lock-or-fail behaviour. Then, it proceeds to check the queues. If it succeeded locking all threads and all queues are empty, termination has occurred. Pseudo-code for the algorithm is shown in Figure 4.

We run the termination detection in a dedicated scheduler thread, which also wakes up threads that have pending work. Ie if it has successfully grabbed any locks, queues belonging to those locked threads are checked, and if any is found to be non-empty, the thread is awakened. After every run, all grabbed locks are released again.

Moreover, although this algorithm works correctly as-is, it is rather inefficient if left running in a loop. Therefore, the scheduler thread goes to sleep after every iteration, and is woken up by any worker thread that goes idle. This requires a slight modification to the algorithm above, since it adds a race-condition, where the last thread going to sleep wakes up the scheduler, which then runs the algorithm before the calling thread manages to go to sleep, assuming termination

```
Require: threads: array of Thread, Thread contains idleMutex and idleCondition,
  fifo
Ensure: termination has occurred iff true is returned
  mutex: Mutex, cond: Condition, held: array of Boolean
  busy ← false
  for t in threads do
    if trylock(t.idleMutex) then
      held[t] ← true
    else
      held[t] ← false
      busy ← true
  for t in threads do
    if not empty( t.fifo ) then
      busy ← true
      if held[t] then
        signal(t.idleCondition)
  for t in threads do
    unlock( t.idleMutex )
  return not busy
```

Fig. 4. Termination Detection in Shared-Memory

did not happen and going to sleep, at which point the system deadlocks, as everyone is idle.

An alternative approach would be to synchronously execute the termination detection algorithm in the thread that has become idle, but due to the nature of the system, the above is more practical code-wise and only incurs very insignificant overhead.

3.6 Implementing OWCTY in Shared-Memory

```
Require: initial is initial state
  S ←Reachability(initial)
  old ← ∅
  while S ≠ old do
    old ← S
    S ←Reset(S)
    S ←Reachability(S)
    S ←Elimination(S)
  return S ≠ ∅
```

Fig. 5. OWCTY Pseudo-code

As can be seen from the pseudo-code (refer to Figure 5), the main OWCTY loop consists of few steps, namely, reachability, elimination and reset. All of them

can be parallelised, but only on their own, which requires a barrier after each of them. Only reachability and elimination run in parallel in the current code, reset is to be implemented.

The algorithm uses a BFS state space visitor to implement both reachability and elimination. The underlying BFS is currently implemented using a partition function, ie, every state is unambiguously assigned to one of the threads. The framework in which the algorithm is implemented offers a multi-threaded BFS implementation based on this kind of state-space partitioning. The algorithm itself is only presented with resulting transition and node-expansion events, unconcerned with the partitioning or communication details.

The barriers are implemented using the termination detection algorithm presented – the computation is initiated by the main thread and the termination detection is then executed in this same thread, which also doubles as a scheduler. When the step terminates, the main thread prepares the next step, spawns the worker threads and initiates the computation again. Since the hash table is always thread-private, ie owned exclusively by a single thread, the main thread has to transfer the hash table among different threads in the serial portion of computation. This is nonetheless done cheaply (few pointer operations only) so is probably not worth parallelising.

4 Experiments

4.1 Methodology

The main testing machine we have used is a 16-way AMD Opteron 885 (8 CPU units with 2 cores each). All timed programs were compiled using gcc 4.1.2 20060525 (Red Hat 4.1.1-1) in 32-bit mode, using -O3. This limits addressable memory to 3GB, which was enough for our testing. The machine has 64GB of memory installed, which means that none of the runs were affected by swapping.

For this paper, our main concern is speed and scalability, therefore we focus on these two parameters. Measurement was done using standard UNIX `time` command, which measures real and cpu times used by program.

All the models we used as inputs to the model checking algorithms, come from BEEM database [18]. The models are in the DVE modeling language, as used in DiVinE [3], for `SPIN` we have used state-space equivalent models in the ProMeLa language.

4.2 Results

First, we have measured run-times of algorithms presented in Section 2 that were implemented using DiVinE framework and `mpich2` library compiled for shared-memory architecture.

These implementations do not exhibit desired scalability as shown in Figure 6. Some algorithms have scaled up to 4 cores. On the other hand, using more cores did not bring any speedup and, as a matter of fact, slowed the computation down due to communication overhead introduced by the MPI library.

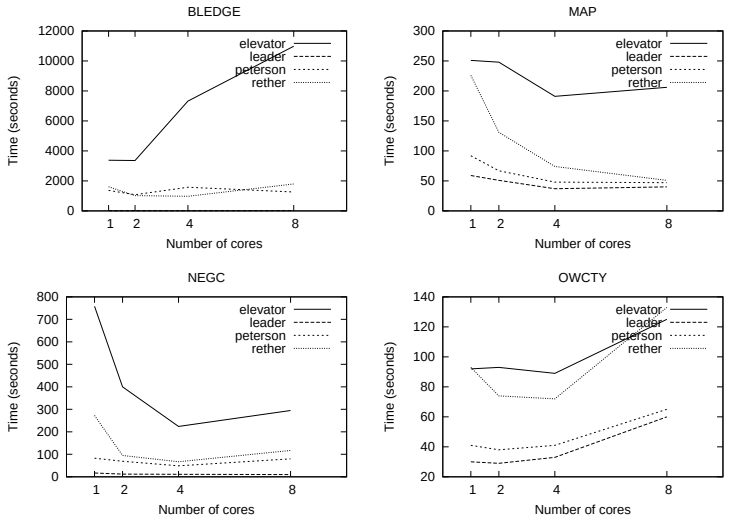


Fig. 6. Run-times of algorithms implemented using DiVinE and MPI.

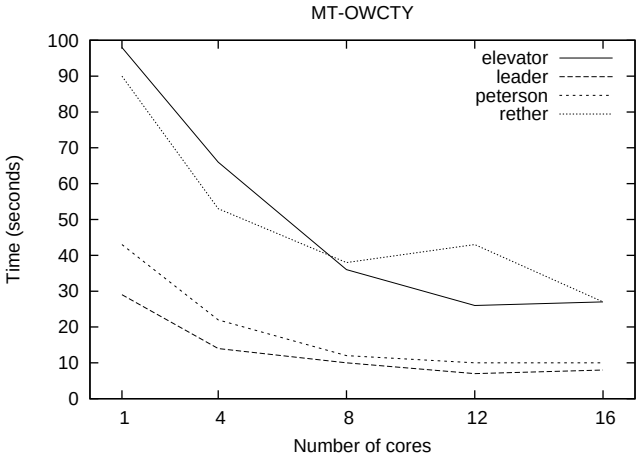
We have performed more experiments to evaluate the efficiency of techniques introduced in Section 3. We have implemented parallel breadth-first search based reachability and the OWCTY algorithm. Run-times of the thread-optimized BFS reachability are given in Figure 8, while the run-times of the thread-optimized implementation of OWCTY algorithm are reported in Figure 7.

The thread-optimized implementations display better scalability behavior, since adding cores reduces computation time at least up to 12 cores, for some models even to 16 cores. Between 12 and 16 cores, the communication overhead reaches a limiting threshold, so adding more does not bring any further speedup and may even impede a slight performance setback.

4.3 Comparison with SPIN

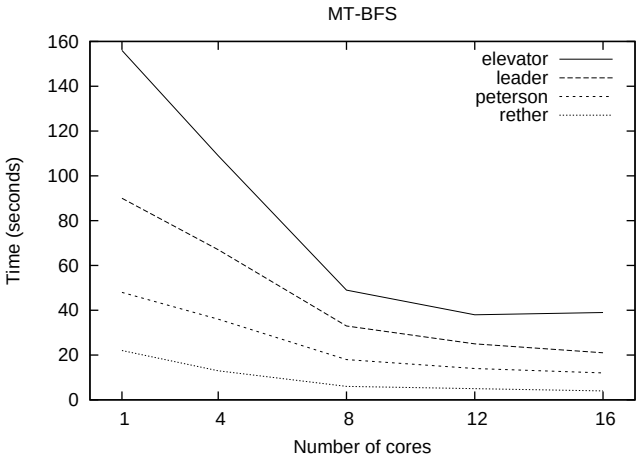
Since the multi-core version of SPIN was not publicly available, in order to make a direct comparison, we run a single reachability on the product automaton graph with SPIN. As SPIN was running only the first procedure of the Nested DFS algorithm we get a good lower bound on runtime of the multi-core SPIN implementation. SPIN was used with parameters -m10000000 -w27 to get the best performance. We have not observed any performance penalty from using bigger stack or hash table than strictly necessary.

We have also measured run-times of a dual-core Nested DFS algorithm as proposed in [10], that was implemented using DiVinE state generator. The run-times are reported in Table 1.



Model	1	4	8	12	16
elevator2.3a.prop4	1:38.7	1:6.1	0:35.2	0:26	0:26.8
leader-filters.5.prop2	0:28.6	0:13.9	0:9.7	0:6.8	0:7.9
peterson.4.prop4	0:42.5	0:22.1	0:12.3	0:9.3	0:9.2
rether.5.prop5	1:30	0:52.7	0:37.5	0:43.3	0:27.2

Fig. 7. Scalability of multi-threaded OWCTY



Model	1	4	8	12	16
anderson.6	2:35.7	1:48.8	0:48.9	0:38.3	0:38.9
elevator2.3	1:29.7	1:7.2	0:33	0:24.9	0:21
elevator2.3a.prop4	0:47.5	0:35.9	0:18.1	0:13.6	0:12.1
peterson.4.prop4	0:21.7	0:13	0:6.3	0:4.7	0:4.2

Fig. 8. Scalability of multi-threaded BFS reachability

Table 2 gives run-times for SPIN, multi-threaded BFS reachability, and OWCTY cycle detection algorithm, both performed on 16 cores.

Model	real	cpu
elevator2.3a.prop4	0:53.4	1:16
leader-filters.5.prop2	0:9.7	0:18.1
peterson.4.prop4	0:24.1	0:33.1
rether.5.prop5	0:45.3	1:5.9

Table 1. Parallel Nested DFS in DiVinE.

Model	SPIN reachability	BFS reachability	OWCTY
elevator2.3a.prop4	0:14.4	0:12.1	0:26.8
peterson.4.prop4	0:7.4	0:4.2	0:9.2

Table 2. Comparison with SPIN

5 Conclusions

We observe, that the algorithms employed by DiVinE, when augmented with the shared-memory-specific techniques, scale fairly well on multiple cores. Our current OWCTY-based, multi-threaded implementation scales up to 12, and for some models, even to 16 cores, which is a definite improvement over the MPI version.

This basically fulfills the goal of implementing a scalable parallel model checker. Thanks to the algorithm used, it has a linear time complexity for majority of LTL properties verified in practice and achieves scalability that makes it practical to use on machines with several CPU cores available.

From the experimental data we see that SPIN outperforms DiVinE in raw speed, but due to SPIN's usage of the Nested DFS algorithm, even if using a parallel nested search, it is bound to execute primary DFS on a single core, which severely limits its scalability potential.

From the profiling work we have done, it is clear that the main bottleneck of DiVinE is its state generator. Improvements in this area should reduce the absolute running times, but will likely negatively affect relative scalability. Therefore, we will continue to work on reducing parallel execution overhead, to maintain or even improve current scalability.

In the pursue of scalability, we also intend to explore alternative approaches to state-space partitioning, non-partitioning approaches and usefulness of load-balancing in this context.

References

1. J. Barnat, L. Brim, and J. Chaloupka. Parallel Breadth-First Search LTL Model-Checking. In *Proc. 18th IEEE International Conference on Automated Software Engineering*, pages 106–115. IEEE Computer Society, 2003.
2. J. Barnat, L. Brim, and I. Černá. Property Driven Distribution of Nested DFS. In *Proceedinfs of the 3rd International Workshop on Verification and Computational Logic (VCL'02 – held at the PLI 2002 Symposium)*, pages 1–10. University of Southampton, UK, Technical Report DSSE-TR-2002-5 in DSSE, 2002.
3. J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček. DiVinE – A Tool for Distributed Verification (Tool Paper). In *Computer Aided Verification*, volume 4144/2006 of *LNCS*, pages 278–281. Springer Berlin / Heidelberg, 2006.
4. L. Brim, I. Černá, P. Moravec, and J. Šimša. Accepting Predecessors are Better than Back Edges in Distributed LTL Model-Checking. In *5th International Conference on Formal Methods in Computer-Aided Design (FMCAD'04)*, volume 3312 of *LNCS*, pages 352–366. Springer-Verlag, 2004.
5. L. Brim, I. Černá, P. Krčál, and R. Pelánek. Distributed LTL model checking based on negative cycle detection. In *Proc. of Foundations of Software Technology and Theoretical Computer Science (FST TCS 2001)*, volume 2245 of *LNCS*, pages 96–107. Springer, 2001.
6. L. Brim, I. Černá, P. Moravec, and J. Šimša. How to Order Vertices for Distributed LTL Model-Checking Based on Accepting Predecessors. In *Proceedings of the 4th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2005)*, pages 1–12, 2005.
7. I. Černá and R. Pelánek. Distributed explicit fair cycle detection (set based approach). In T. Ball and S.K. Rajamani, editors, *Model Checking Software. 10th International SPIN Workshop*, volume 2648 of *Lecture Notes in Computer Science*, pages 49 – 73. Springer Verlag, 2003.
8. C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory-Efficient Algorithms for the Verification of Temporal Properties. *Formal Methods in System Design*, 1:275–288, 1992.
9. K. Fisler, R. Fraer, G. Kamhi, M. Y. Vardi, and Z. Yang. Is there a best symbolic cycle-detection algorithm? In *Proc. Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 420–434. Springer-Verlag, 2001.
10. G. Holzmann. The Design of a Distributed Model Checking Algorithm for SPIN. In *FMCAD, Invited Talk*, 2006.
11. G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
12. G. J. Holzmann, D. Peled, and M. Yannakakis. On Nested Depth First Search. In *The SPIN Verification System*, pages 23–32. American Mathematical Society, 1996. Proc. of the 2nd SPIN Workshop.
13. C. Inggs and H. Barringer. Ctl* model checking on a shared memory architecture. *Formal Methods in System Design*, 29(2):135–155, 2006.
14. A. L. Lafuente. Simplified distributed LTL model checking by localizing cycles. Technical Report 00176, Institut für Informatik, University Freiburg, Germany, July 2002.
15. Ho-Fung Leung and Hing-Fung Ting. An optimal algorithm for global termination detection in shared-memory asynchronous multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):538–543, 1997.

16. M. M. Michael. Scalable lock-free dynamic memory allocation. *SIGPLAN Not.*, 39(6):35–46, 2004.
17. M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Symposium on Principles of Distributed Computing*, pages 267–275, 1996.
18. R. Pelánek. BEEM: BEnchmarks for Explicit Model checkers. `http://anna.fi.muni.cz/models/index.html`, February 2007.
19. R. Tarjan. Depth First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, pages 146–160, Januar 1972.