Coarse-Grained Locking Scheme for Parallel State Space Construction

Silvano Dal Zilio, Rodrigo Tacla Saad, Bernard Berthomieu

Coarse-Grained Locking Scheme for Parallel State Space Construction

Rodrigo T. Saad[1], Silvano Dal Zilio[2] and Bernard Berthomieu[2]
rsaad@das.ufsc.br, {dalzilio, bernard}@laas.fr

[1] DAS-CTC-UFSC; Bairro Trindade, 88040-970
Florianopolis, Brazil
[2] CNRS; LAAS; 7 ave. Colonel Roche, F-31077
Université de Toulouse; UPS, INSA, INP, ISAE; LAAS; F-31077 Toulouse, France

Abstract. We propose a new parallel algorithm for state space construction that is well-suited for modern, multiprocessor architectures with non-uniform memory access times. Our algorithm uses a network of distributed hash tables to store the states locally—we use one hash table per processor (or thread)—and a shared entity, suggestively called the *dispatcher*, to control the distribution of data among all these tables. Conflicts in accessing the same shared memory location simultaneously are resolved by a traditional CREW strategy; we use one lock per hash table to grant write accesses but read accesses are allowed to occur concurrently. With this approach, we combine the simplicity and ease of implementation of distributed hash tables with the dynamic data distribution of concurrent data structures. We evaluate the performance of our algorithm on different benchmarks.

1 Introduction

Model Checking, much like other formal verification techniques, is a very resource-intensive process. This may explain why the implementation of model-checking tools has often followed advances in hardware. In this paper, we propose an algorithm for parallel state space construction that is well-suited for multiprocessor and multicore servers with shared-memory architectures. This algorithm relies on a new coarse-grain locking scheme to reduce synchronization overheads and favors data locality in order to be efficient on NUMA architectures (that is, such that memory access times are non-uniform.)

Our algorithm is based on a previous work on parallel state space generation [5] where we used a "modified" Bloom filter, together with a set of distributed hash tables, to store the states. In this context, the distribution of states among processors is done dynamically and the location of each state is tracked by a shared data structure called a Localization Table, or *LT* for short. Although we obtained some very good results when compared to other parallel algorithms, one drawback is that we need a rough estimate of the state space size to dimension the *LT*. Indeed, the performance of the algorithm degrades severely if the initial table is not large enough to fit all the states. The problem

stems from the fact that the LT cannot be resized without blocking the state space exploration (on the opposite, local hash tables may be resized at any time). An acceptable choice is to use a bound on maximal number of states that can be processed by the computer, for example by dividing the amount of available memory by an estimate of the size of one state. Nonetheless, this may lead to a waste of memory space. (On typical models, the LT occupies almost 5% of the total memory used during state space exploration.)

In this work, we improve on the algorithm of [5] and solve the problems related to the size and the parametrization of the LT. Our new solution is based on the observation that a uniform distribution of states among processors (both in time and space) helps reduce the probability of a collision: two states assigned with the same key at the same time. Our algorithm uses a network of distributed hash tables, one on each processor, but replaces the shared LT by an entity called the *dispatcher*, that is in charge of controlling the distribution of data among all the tables. The dispatcher maps every generated state to one of the hash tables. Conflicts in accessing the same hash table simultaneously are resolved by a traditional Concurrent Read Exclusive Write (CREW) strategy; we use one lock per hash table to grant write accesses but read accesses are not constrained. Our main goal is to provide an algorithm with similar performances but without the need for parametrization and with a lower memory footprint.

Related Work: the use of distributed hash tables with static partition of states is the most widespread solution for parallel state space exploration; an example is the version of DiVinE [1] for multicore machines where each process owns a private EREW hash table. In contrast, Spin [2], LTSmin [4] and PMC [3] use a concurrent data structure with a dynamic distribution of data. Spin uses the stack-slicing strategy to share work in combination with a shared hash table protected by a fine-grained locking scheme. This approach has been extended in [4] with a lockless shared hash table based on atomic primitives (Compare & Swap). Finally, we can cite the work of Inggs and Barringer [3], that use an unsafe shared hash table together with a work stealing strategy to provide dynamic load balancing.

Contributions: we define a new algorithm that combines the simplicity and ease of implementation of distributed hash tables with the dynamic data distribution of concurrent data structures. In addition to a light usage of locks, our algorithm enables the individual resizing of local hash tables without blocking the entire state space exploration. (The dispatcher does not require any setup and its dimension does not depend on the number of elements to be inserted.) Compared to fine-grained locking schemes, our algorithm requires less memory (at most 65 KB) to ensure data consistency. For comparison, the algorithm in [4] stores a spare tuple with a memoization value and a *write status* bit for each state.

Outline: we detail our algorithm in Sect. 2 and propose two implementations for the dispatcher. Before concluding, we give some preliminary results obtained on a set of typical benchmarks.

2 Coarse-Grained Locking Scheme

We start by presenting our original algorithm of [5] before giving the details of our coarse-grained locking scheme. Our original algorithm is based on the use of a set of hash tables, one table on each processor, to store the states. Coordination between the processors is based on a shared data structure, the Localization Table (LT), that tests whether a state has already been found and, if so, keeps track of its location. (When a new state is computed, it is preferably located on the processor that found it.)

The work performed by each processor is pretty simple: compute the successor of a state, say s, and check in the LT where it could have potentially been assigned. If s is a newly discovered state, it will be assigned to the processor who generated it. Otherwise, the LT will return the location where the state s is assigned, say $LT(s)$.

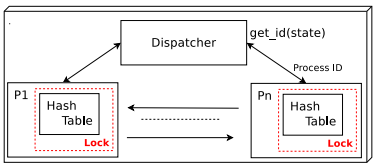

Fig. 1. Algorithm overview.

The architecture of our new algorithm is quite similar (see Fig. 1), except that we replace the LT with a new data structure, suggestively called the dispatcher. We say that we use a *coarse-grained locking* scheme because we need only one lock per hash table to guarantee exclusive write access. Like the LT, the dispatcher returns a processor id for each new state. However, the dispatcher distributes data between hash tables, not between processors like the LT. As a consequence, the dispatcher will always assign the same key to a given state regardless of whether it is a newly discovered state or not, just like with a static hash function. Processors are then allowed to write on a different hash table than the one they house. We present our algorithm in more details in the following sections.

Algorithm Operation and Pseudocode: our algorithm follows an "homogeneous" parallelization approach, where all processors execute the same program simultaneously. We use a work-stealing strategy (see [5]) to balance the work-load among all processors during the exploration; for instance, whenever a thread has no more states to explore, it tries to "steal" non-explored states from other processors.

The dispatcher is implemented as a shared object that supports the *Dispatcher.get_id* operation. It receives as input a state s and returns a processor id (a value in $1 \ldots N$). Each processor manages a *private work* stack of unexplored states and a local hash table to store the states. The shared memory space consists of: one bitvector of size N to store the state of the processors (idle or busy), used to detect termination; N *shared work* stacks—one for each processor—for the work sharing technique; and finally N *forward* stacks used to prevent processors from blocking when they are not granted with the lock for exclusive write access.

We give the pseudo-code of our algorithm in Listing 1.1. The state space exploration proceeds until all stacks are empty. Given a state s, a processor, say my_id, will check over the dispatcher for the owner of s. This information is

returned by a call to the function Dispatcher.get_id(s). From the *id* returned by the dispatcher, the process my_id performs a look-up operation over the local table of processor *id* to check if the state is really there. If the state is not found, it tries to insert the state itself on the local hash table of processor *id*. If process my_id is not able to lock the given hash table for exclusive write, we tag it as a *forward state* and add it to the forward stack of processor *id*. Forward states are specifically tagged since they bypass the dispatcher test and are directly inserted in the local table, forcing the processor to wait until the lock is granted (see line 16). When the private work stack is empty, work is transferred from shared work and forward stacks; if they are also empty, the processor may "steal" work from others. The private stack holds all states that should be worked upon and the shared stack for states that can be borrowed by idle processors.

```
1  while (one process still busy)
2    while (Proc[my_id].private_stack not empty)
3       do s ← remove state from Proc[my_id].private_stack;
4          write_lock_granted ← false
5          if s not tagged as forward
6             then status ← TRY;
7                  id ← Dispatcher.get_id(s);
8          else status ← INSERT; // Forward State
9               id ← my_id;
10         endif
11         if s not in Proc[id].local_table
12              then if status = TRY //Try to get the lock
13                        then if try lock of Proc[id].local_table
14                                then write_lock_granted ← true;
15                             endif
16                   else //Wait for the lock (status = INSERT)
17                        wait and get lock of Proc[id].local_table
18                        write_lock_granted ← true;
19                   endif
20                   if write_lock_granted
21                        then add s to Proc[my_id].local_table
22                             release lock of Proc[id].local_table
23                         generate the successors from s
24                         and put some in Proc[my_id].private_stack;
25                         ...
26                   else tag s as forward;
27                       add s to Proc[id].forward_stack;
28                   endif
29       endif
30   endwhile
31   transfer work from Proc[my_id].forward_stack
32      and Proc[my_id].shared_work_stack to Proc[my_id].private_stack;
33      ...
34 endwhile
```

Listing 1.1. Algorithm pseudo-code

Versions: we tested three variants of our algorithm, they are:

Non-blocking Static: on its simplest form, the dispatcher can be implemented as a hash function h, with image in $1 \ldots |N|$, that maps states to one of the processors identifiers.

Non-blocking Vector: in this version, the dispatcher is essentially a small "table" that associates a processor id to range of keys in the table. The idea is to build incrementally an approximation of the "best" distribution function by exploring the first states. The table will behave as a static hash

function once it is filled. We can implement the table using an integer vector V of size n and, for computing the key of a state, an independent hash function, h. (The best results were achieved using a vector of $n = 65536$ slots, which is the 64 KB level 2 cache memory size of the processors used.)

Blocking Vector: this is the same implementation than previously but without the "forward" stacks. From Listing 1.1, we replace the "try lock" for the "wait lock" operation at line 13, forcing the processor to wait until the write lock is granted. Lines 16 to 18 and 26 to 28 are no longer needed because all locks are resolved and no state is forwarded.

3 Results

We implemented our algorithm using the C language with POSIX threads and atomic primitives. Our experimental results have been obtained using a Sun Fire server with eight dual-core Opteron processors and 208 GB of RAM. We used classical, finite-state problems (Dining Philosophers; Kanban; ...) and puzzles (Peg-Solitaire; Sokoban; ...) for the benchmarks[3]. We only selected medium size models, with number of states varying from a few millions (Sokoban) to a half-billion states (Frogs).

We give the average relative speedup for the three versions of the dispatcher in Fig. 2. In this figure, NB_STATIC stands for Non-Blocking Static, NB_VECTOR for Non-Blocking Vector and B_VECTOR for Blocking Vector. Figure 3 gives the ratio of missed to acquired locks by the number of states inserted per second for a set of 8 experiments using 16 processors (or threads). The number of inserted states per second ranges from 4.10^5 (Hanoi) to 2.10^6 (Frog). These results support our initial assumption that there is a relation between the "uniformity" of the state distribution and the probability of write conflicts; in the worst case, the percentage of missed locks is smaller than 12% of the locks acquired.

Figures 5 and 4 compare the previous version of our algorithm [5] with the new one. Figure 5 shows the average mean-standard deviation (MEAN-STD) of states among all processors and Figure 4 gives the ratio of read to write accesses. We see that, on average, our algorithm is able to match the performance obtained using the LT. The smaller number of memory accesses made by the LT version can be explained by its "more localized" distribution of states. We can also mention the extra read accesses performed by our coarse-grained scheme due to the missed locks. Finally, we see that we can outperform our original algorithm using the NB_STATIC version of the dispatcher and also have a better physical distribution of data.

4 Conclusion

We presented a new coarse-grained locking scheme for parallel, enumerative state space exploration. Our preliminary results are promising. In particular, we obtained performances similar to the one observed with our previous approach [5],

[3] All data collected from our experiments are available at http://goo.gl/e7C2I

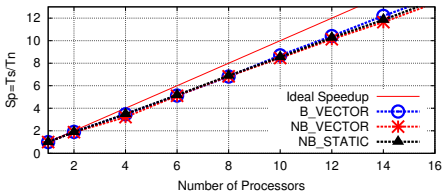

Fig. 2. Speedup Analysis.

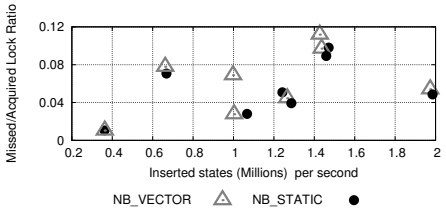

Fig. 3. Lock Misses Analysis.

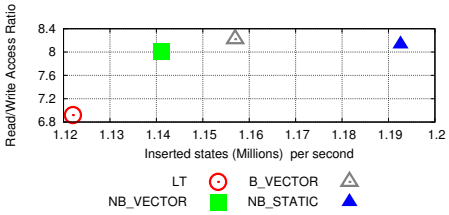

Fig. 4. Read/Write accesses Analysis.

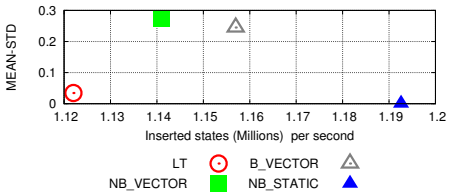

Fig. 5. Standard Deviation Analysis.

but with a smaller memory footprint and without the need to parametrize an auxiliary data structure. This result is encouraging since, in [5], we observed performances close to those obtained using an algorithm based on lockless hash tables (that may be unsafe) and that outperformed an implementation based on the concurrent, unordered map provided in the Intel Threading Building Blocks, an industrial strength lockless hash table. In future works, we intend to perform a broader comparison with state of the art tools. (We already obtained favorable comparison against Spin and LTSmin using our *LT*-based algorithm.)

References

1. J. Barnat, L. Brim, and P. Ročkai. Scalable multi-core ltl model-checking. In *Model Checking Software*, volume 4595 of *LNCS*, pages 187–203. Springer, 2007.
2. G.J. Holzmann. A stack-slicing algorithm for multi-core model checking. *Electronic Notes in Theoretical Computer Science*, 198(1):3–16, 2008.
3. Cornelia P. Inggs and Howard Barringer. Effective state exploration for model checking on a shared memory architecture. In *Parallel and Dist. Model Checking*, volume 68(4) of *ENTCS*, 2002.
4. A. W. Laarman, J. C. van de Pol, and M. Weber. Boosting multi-core reachability performance with shared hash tables. In *Proc. of the 10th Int. Conf. on Formal Methods in Computer-Aided Design*. IEEE Computer Society, 2010.
5. Rodrigo T. Saad, Silvano Dal Zilio, and Bernard Berthomieu. Mixed Shared-Distributed hash tables approaches for parallel state space construction. In *International Symposium on Parallel and Distributed Computing (ISPDC 2011)*, page 8p., Cluj-Napoca, Romania, July 2011. Rapport LAAS 11460.