Reconciling Distributed and Shared Hash Tables Approaches for Parallel State Space Construction

Rodrigo Tacla Saad, Silvano Dal Zilio, Bernard Berthomieu

Reconciling Distributed and Shared Hash Tables Approaches for Parallel State Space Construction

Rodrigo T. Saad, Silvano Dal Zilio and Bernard Berthomieu
CNRS; LAAS; 7 ave. Colonel Roche, F-31077 Toulouse, France
Université de Toulouse; UPS, INSA, INP, ISAE; LAAS; F-31077 Toulouse, France
{*rsaad, dalzilio, bernard*}*@laas.fr*

***Abstract*—We propose an algorithm for parallel state space construction based on an original concurrent data structure, called a *localization table*, that aims at better space and temporal balance. Our proposal is close in spirit to algorithms based on distributed hash tables, with the distinction that states are dynamically assigned to processors; i.e. we do not rely on an a-priori static partition of the state space.**

In our solution, every process keeps a share of the global state space. Data distribution and coordination between processes is made through the localization table, that is a lockless, thread-safe data structure that approximates the set of states being processed. The localization table is used to dynamically assign newly discovered states and can be queried to return the identity of the processor that own a given state. With this approach, we are able to consolidate a network of local hash tables into an (abstract) distributed one without sacrificing memory affinity – data that are "logically connected" and physically close to each others – and without incurring performance costs associated to the use of locks to ensure data consistency.

We evaluate the performance of our algorithm on different benchmarks and compare these results with other solutions proposed in the literature and with existing verification tools. At a more general level, the usefulness of localization tables goes beyond the domain of formal verification, since it provides an efficient substitute for concurrent hash maps. For instance, our experiments show that our solution performed very well against an industrial strength, lockless hash table (taken from the Intel-TBB).

I. Introduction

Model-checking is a very demanding activity in terms of computational resources. As a result, the extensive need for memory and computation power has resulted in the design of model checking algorithms that target parallel and distributed machines. Variations between these algorithms are often explained by differences between the targeted architectures – shared-memory versus distributed memory, clusters, ... – or differences on the criteria to optimize – achieving better spatial balance between processes, lowering synchronization costs, ...

We propose an algorithm for parallel state space construction intended for shared memory, multiprocessor machines. (We focus here on the exhaustive generation of the state space of finite-state transition systems, often a preliminary step for model-checking.) The basic idea behind a state space construction algorithm is pretty simple: take a state that has not been explored (a fresh state); compute its successors and check if they have already been found before; iterate. Hence, a key point for performance is to use an efficient data structure for storing the set of generated states and for testing membership in this set.

In our approach, the goal is to build the state space of a system concurrently in such a way that: (1) the share of state space build by each processor be as uniform as possible; and (2) the processor occupancy is maximal. Our algorithm builds on previous work [23] and is based on the same simple design: the global state space is stored in a set of local containers (e.g. hash tables), each controlled by a different processor, while only a small part of the shared-memory is used for coordinating the state space exploration. This is close in spirit to algorithms based on distributed hash tables, with the distinction that we choose to dynamically assign states to processors, that is, we do not rely on an a-priory static partition of the state space.

Data distribution and coordination between processes is made through a *localization table* (*LT*), that is a lockless, thread-safe data structure that approximates the set of states being processed. The localization table is used to dynamically assign newly discovered states and behaves as an associative array that returns the identity of the processor that own a given state. With this approach, we are able to consolidate a network of local hash tables into an (abstract) distributed one without sacrificing memory affinity – data that are "logically connected" and physically close to each others – and without incurring performance costs associated to the use of locks to ensure data integrity.

The paper is organized as follows. In Section II we review the related work. Section III details our algorithm and defines the data structure for *localization tables*. Before concluding, we report on experimental results obtained on a set of typical benchmarks and compare our approach with solutions already proposed in the literature.

II. Related Work

Model Checking [8] is an automated verification method used to check whether the model of a system meets a given

This work has been supported by the French AESE project Topcased and by région Midi-Pyrénées

specification. This technique relies on the exploration of the state space of the model. State space construction can be classified as an irregular parallel problem because state graphs may be highly irregular, see [9] for a discussion on this topic. As a consequence, when parallelizing this problem, special attention should be taken to ensure a good load balancing among processors.

Several approaches have been proposed, since the early 1990s, for parallel and distributed state space exploration. These solutions adopt, in their vast majority, a common paradigm that could be labeled as "homogeneous" parallelism – a Single Program Multiple Data (SPMD) programming style – such that each processor performs the same steps concurrently. Most of these solutions, to mention a few [1, 7, 10, 18, 20, 24], were intended for distributed computers and rely on slicing functions – that is functions that statically assign a state to a processor – and basically only differ by the nature of these functions. The choice of slicing functions has a major influence on the load balance and data locality of the algorithms.

A smaller number of solutions target shared memory machines [11, 12, 13, 15, 1]. Like in our case, some of these solutions are based on the use of hash tables to store the states. An example is the version of DiVinE [2] for multicore machines, that is based on a static partitioning scheme and where each process owns a private hash table. Another example is the parallel version of Spin [12, 11] which uses the stack-slicing strategy to share work in combination with a shared hash table protected by a fine-grained locking scheme. This work has been recently extended with a lockless shared hash table based on atomic primitives [15] (CAS – *Compare & Swap*). Additionally, we can mention the work of Inggs et al. [13], which proposes a parallel algorithm based on a *work stealing* scheduling paradigm to provide dynamic load balancing. In their case, the data structure used to store the states is an "unsafe" shared hash table.

In the context of this work, we propose an extension of an algorithm that we defined in [23], which is based on two data structures: a lock-free, shared *Bloom filter* [5] to coordinate the data distribution; and local data containers to explicitly store the data (for instance, our initial implementation was based on AVL trees). The Bloom filter is used to represent, in a very compact way, the set of states that have already been found and to efficiently test whether a given state has already been found. Due to the probabilistic nature of the Bloom Filter, the algorithm is based on multiple iterations in order to perform an exhaustive exploration. In a first phase, the algorithm is guided by the Bloom filter until no new states can be found. During this phase, states found by a processor are stored locally in two dictionaries: one for states that, according to the Bloom filter, have also been found by another processor; the other for fresh states. Since the Bloom filter may, at times, falsely report that a state has already been visited, we need to handle these *collision* states in a second phase of the algorithm. The computation stops when there are no more states to explore and no more collisions.

Our Contributions: The algorithm proposed in this paper improves on our previous design and replaces the Bloom Filter by a dedicated data structure, the *localization table*. Unlike Bloom Filters, this data structure can be used to find the processor that owns a given data item – a state in our case – and not only if the object was already found. This simple addition significantly enhance the performance of our previous algorithm and also simplifies its logic. Indeed, it is now possible to solve possible collisions on-the-fly and to get rid of the collision resolution phase.

Our contributions are twofold. First, in the formal verification domain, we define a new algorithm for parallel state space construction. Our algorithm is able to exploit parallelism in all possible cases and, unlike algorithms based on slicing functions or heuristic rules, is compatible with dynamic load-balancing techniques. Second, in the parallel computing domain, the combination of local hash tables with a localization table provides an interesting implementation for concurrent hash maps that may be useful in other situations.

Our preliminary results are very promising. We observe performances close to those obtained using an algorithm based on lockless hash tables (that may be unsafe) and that outperforms an implementation based on the concurrent, unordered map provided in the Intel Threading Building Blocks [21], an industrial strength lockless hash table.

III. Description of the Algorithm

Our algorithm follows a "homogeneous" parallelization approach, where every processors execute the same program simultaneously and each processor handles its own local view of the state space. Coordination between the processors is based on a *Localization Table* (*LT* for short), that is used to allocate newly discovered states to processors.

The *LT* is used to test whether a state has already been found and, if so, to keep track of the location – the processor *id* – where the state is held. The work performed by each processor is pretty simple: generate a state using the model of the system, 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). This approach has the advantage to isolate the local hash tables; each processor has exclusive write access to its local table, whereas concurrent read access are unrestricted. As a result, we consolidate a network of distributed hash tables into a single, concurrent data structure. Another advantage is that we can easily resize local hash tables, as needed, without blocking the entire exploration.

We describe more precisely how the *LT* is implemented in Section III-A. An advantage of our design is to be thread-safe: operations on a *LT* are simple and can be implemented using atomic actions. Another advantage is the small footprint of the *LT*. To summarize, our goal is to combine the advantages of distributed and shared hash tables for parallel state space construction in a single algorithm.

In the remainder of the text, we use N to denote the number of processors and $1..N$ for the natural interval between 1 and N. We define the localization table in Sect. III-A. The work-sharing techniques used in our algorithm is discussed in Sect. III-B while Sect. III-C gives some pseudo-code and further explanations about the algorithm.

A. Localization Table

Storing the relation $(s, LT(s))$ – associating each state, s, with the processor that owns it – in a single table would require a very large amount of memory. Actually, it would defeat the need to store the state themselves. Instead, the idea is to use a notion of *key* associated to a state and to store the association between keys and processors. In our implementation, keys are computed using hash-functions and we will use a scheme based on multiple keys.

A localization table is essentially a "table" that associates a processor id – a value in $1..N$ – to every key in the table. A straightforward implementation is to use an integer vector for the underlying table. We can implement the table using a vector V of size n and, for computing the key of a state, an independent hash function, h, with image in $1..n$. In this case, we can check if a state s has already been found by looking into the local table of processor $V[h(s)]$. While this implementation is simple, its disadvantage is that it is not possible to ensure a fine dynamic distribution of the states if h is not uniform. Indeed, if processor id_1 finds a new state, s, such that $V[h(s)] = id_2$, then we need to transfer s between the two processors. A solution is to increase the size of the vector – but this has a direct impact on the memory consumption – or to use better hashing functions – but this has an impact on the performances.

We propose another implementation of the localization table that improves upon the choice of a vector. Inspired from our previous experience with *Bloom Filter* (*BF*), the idea is to use a finite family of hashing functions $h_1, \ldots, h_k$. To test if a state s is in the *LT*, we search if the key $h_1(s)$ is in *LT*. If it is not, we know that the state is fresh. If $LT(h_1(s)) = id_1$ then we check if s is in the local table of processor id_1. If s is not owned by id_1, we continue searching with the key $h_2(s)$ and so forth.

This is only a rough description of how the *LT* works. Next, we define more formally the operation of our data structure. In particular, we explain how to deal with states that are not in the processors $LT(h_1(s)), \ldots, LT(h_k(s))$, that we call *collision* states. By convention, our algorithm will route a collision state to the last processor found, that is $LT(h_k(s))$.

A *Localization Table*, L, is defined by two parameters: its size n; and a family of k independant hashing functions $h_1, \ldots, h_k$ with image in $1..m$ (where we choose m such that $n \ll m$). In our implementation, we typically choose 64 bytes hash-functions, that is $m = 2^{64}$, while we choose for n an over-approximation of the state space size (see the discussion on ratio in Sect. IV). Given the performances of our test machine, we generally work with a billion states, that is $n \approx 2^{30}$.

A localization table L of size n is an array of size n containing pairs of the form (p, d), where p is a processor id ($p \in 1..N$) and d is a key ($d \in 1..m$). To look for values inside of L, we use a fixed surjective function map, from $1..m$ to $1..n$. Hence, to check the value associated to the key $h_i(s)$, we look in the array L at index $map(h_i(s))$.

Initially, an empty *LT* is an array initialized with the value $(0, 0)$. Assume that the processor id attempts to insert a state s into L. We define this operation in Fig. 1 using pseudo-code. The function takes as input a state and a processor *id* and returns a pair made of: a status, to determine if the element is new (or old); and the identifier of the processor who owns the state. The insertion operation is performed by looking successively at the elements with index $map(h_i(s))$ in L for all $i \in 1..k$. There is three possible cases at each step:

- if $L[map(h_i(s))] = (0, 0)$ then we know for sure that the state is fresh (it has never been added before). We can stop our iteration and write the pair $(id, h_i(s))$ in L. This can be done using an atomic compare and swap operation;
- if $L[map(h_i(s))] = (id', d)$ and $d \neq h_i(s)$ then we cannot decide if the state s has been found and we continue to the next iteration, with the key function h_{i+1};
- if $L[map(h_i(s))] = (id', h_i(s))$ then we answer that s is in the local table of processor id' with high confidence. With this approach, states with the same hash value are not handled at the *LT* level. In our algorithm, these collisions will be spotted when the processor tries to recover the state from the local table of id'. In order to keep the consistency of the *LT* and to prevent states from being stored more than once, we choose to assign s to the processor id' and this is handled like a collision state.

Finally, in the case where we cannot decide after checking the values of $L[map(h_i(s))]$ for $i \in 1..k$, we also say that s is a collision state and we choose to assign s to the processor id' such that $L[map(h_k(s))] = (id', d)$.

The operation for checking whether a state s is already in the *LT* is very similar to the insertion function. We test successively if there is an index $i \in 1..k$ such that

```
function test_or_insert(s:state, id:1..N) : (status, id)
  (p,d) ← (0,0);
  for i in 1..k do
     (p,d) ← LT[map(h_i(s))];
     if p = 0   // the slot is empty means that s is fresh
     then LT[map(h_i(s))]←(id,h_i(s));
          return (new,id);
     elsif d = h_i(s) // the state s may already be in processor p
     then return (old,p);
     endif
  endfor
  //state s is a collision – assign it to processor d;
  return (old,d);
```

Listing 1. Insertion in the Localization Table

$L[map(h_i(s))] = (id_i, h_i(s))$, stopping if one of the position in L is empty. If this is the case, we know that s is not in the *LT*. If we find no match after k attempts, then we consider that s is a collision state that belongs to id_k.

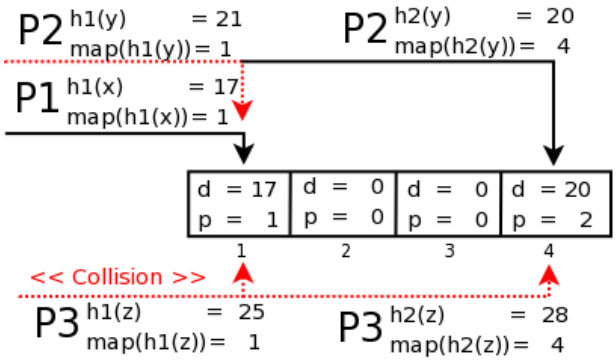

Figure 1. Insertion in a Localization Table.

In Figure 1 we illustrate the insertion and test operation for three data items: x, y, and z; performed in this order by the processors P_1, P_2 and P_3. The figure display a *LT* of size $n = 4$ with two independent hash functions h_1 and h_2. We assume that $k = 2$ and that $m = 31$. The insertion of x requires only one operation since the slot at position $map(h_1(x))$ is initially empty. As a result the slot is associated to processor P_1 for elements with key 17. Element y is inserted at the second attempt, since the slot in position $map(h_1(y))$ is already filled and that $h_1(y) \neq d_1$. Finally, element z cannot be properly inserted – it is a collision – and it is assigned to processor P_2.

B. Work-Sharing Techniques

Our algorithm relies on two different work-sharing techniques to balance the working load between processors. We use these mechanisms alternately during the exploration phase depending on the processor occupancy. First, we use an *active* technique very similar to the work-stealing paradigm of [13]. This mechanism uses two stacks: a private stack that holds all states that should be worked upon; and a shared stack for states that can be borrowed by idle processors. This shared stack is protected by a lock to take care of concurrent access. The second technique can be described as *passive* and has the benefit to avoid useless synchronization and contention caused by the active technique. In the passive mode, an idle processor waits for a wake-up signal from another processor willing to give away some work instead of polling other shared stacks. The shift between the passive and active modes is governed by two parameters:

- the *private minimum workload*, which defines the minimal charge of work that should be kept private. The processor will share work only if the charge in its private stack is larger than this value;
- the *share workload*, which defines the ratio of work that should be added in the shared stack if the load in the private stack is larger than the private minimum workload.

Our implementation of the work-stealing paradigm is interesting in its own right since it differs from [13] by its use of unbounded shared stacks and the use of a "share workload" parameter. The description of this extension is also interesting since it shows that common optimizations and load-balancing techniques are not precluded by our algorithm, which is not the case with algorithms based on static slicing functions.

C. Algorithm

To conclude this section, we give a high-level view of our algorithm that can be described by the pseudo-code of Listing 2. The diagram of Fig. 2 describes the shared and local data structures used in the algorithm. Each processor manages a "private work" stack of unexplored states and a local hash table to store the states assigned to him. The shared values are: the Localization Table; one bitvector of size N to store the state of the processors (idle or busy), used to detect termination; N stacks – one for each processor – for the work sharing technique described in Sect. III-B; and finally N collision stacks used to route collisions states to their correct processors.

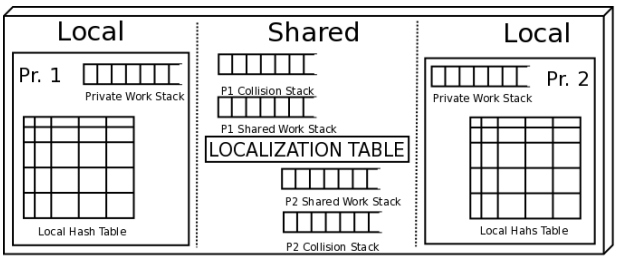

Figure 2. Shared and Private Data.

The state space exploration proceeds until no new states can be added to the *LT* and all stacks are empty. Given a state s, a processor, say my_id, will check the *LT* to

```
while (one process still busy)
   while (Proc[my_id].private_stack not empty)
      do s ← remove state from Proc[my_id].private_stack;
         if s not tagged as collision
         then (status, id) ← LT.test_or_insert(s,my_id);
         else (status, id) ← (new, my_id); //Collision state
         endif
         if status = old
         then if s not in Proc[id].local_table
              then tag s as collision;
                   add s to Proc[id].collision_stack;
              endif
         else
            add s to Proc[my_id].local_table ;
            generate the successors from s
            and put some in Proc[my_id].shared_work_stack;
            ...
         endif
   endwhile
   transfer work from Proc[my_id].collision_stack
      and Proc[my_id].shared_work_stack
      to Proc[my_id].private_stack;
   ...
endwhile
```

Listing 2. Algorithm pseudo-code

test whether s is new and, otherwise, what is the owner of s. This information is returned by a call to the function test_or_insert(s, my_ id) that is defined in Listing 1. During the exploration, states that are labeled as new by the *LT* are stored in the local table of the processor. On the opposite, if the *LT* returns an owner *id*, then the process 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, we can tag it as a *collision state* and add it to the collision stack of processor *id*. Collision states are specifically tagged since they bypass the *LT* membership test and are directly inserted in a local table. When the private work stack is empty, work is transfered from shared work and collision stacks; if they are also empty, the processor may "steal" work from others (as described in Sect. III-B). The *LT* is implemented using an atomic compare and swap primitive, while locks are only used to protect the access to the other shared data structures – the shared work and collision stacks – which are not resource contention points.

Finally, termination can be easily detected by testing the vector recording the states of processors; the algorithm may safely finish if there is no more state to explore, that is if the stacks of all the processors are empty and if all the processors are idle.

IV. Experiments

We implemented our algorithm using the C language with Pthreads [6] for concurrency and the Hoard Library [4] for parallel memory allocation. We developed a library for the *Localization Table* with support for concurrent insertions and we used Bob Jenkins's hash function [14] to generate hash keys from states. The Experimental results presented here are obtained using a Sun Fire x4600 M2 Server, configured with 8 dual core opteron processors and 208GB of RAM memory, running the Solaris 10 operating system. This machine is classified as Non-Uniform Memory Access (NUMA) because the shared memory is physically divided among the processors.

For the benchmark, we used models taken from two sources. We have three classical examples: Dining Philosophers (PH); Flexible Manufacturing System (FMS); and Kanban – taken from [16] – together with 5 Puzzles models: Peg-Solitaire (Peg); Sokoban; Hanoi; Sam Lloyd's puzzle (Fifteen); and 2D Toads and Frogs puzzle (Frog) – taken from the BEEM database [19]. All these examples are based on finite state systems modelled using Petri Nets [17]. This means that, in these cases, a state is a *marking*, that is a tuple of integers. Our algorithm may be adapted to other formalisms, for instance including data, time, etc.

With our computer setup, we are able to tackle examples with approximately 10 billions of states, but we selected models with less than 500 millions of states in order to complete our experiments in reasonable time. (A complete run of our benchmark takes more than a week to finish.)

We study the performance of our implementation on different aspects. While speedup is the obvious criteria, we also study the memory footprint of our approach and the physical distribution of states among processors.

A. Speedup and Physical Distribution

In Fig. 3 we give the observed speedup of our algorithm on a set of examples. We give the absolute speedup, measured as the ratio between the execution time using N processors (T_N) and the time of an optimized, sequential version. Our implementation delivers some promising speedups. The results also show different behaviors according to the model. For instance, our efficiency[1] may vary between 90% (Hanoi model) and 51% (Kanban model), whereas the system occupancy rate[2] is consistently over 95%. Clearly, the algorithm depends on the "degree of concurrency" of the model – it is not necessary to use lots of processors for a model with few concurrent actions – but this is an inherent limitation with parallel state space construction [9], which is an irregular problem.

Concerning the use of memory, we can measure the quality of the distribution of the state space using the *mean standard deviation* (σ) of the number of states among the processors. In our experiments, we observe that the value of σ is quite small and that it stays stable when we change the number of processors (see Fig. 8). For instance, we have $\sigma \approx 1.5\%$ for the Hanoi model and $\sigma \approx 7\%$ for Kanban. The difference between values of σ can be explained by the difference in the "degree of concurrency". It may also be affected by the processor's performance, that is, a processor that handles "simpler states", or smaller work units, may

[1]Efficiency is computed as the ratio between speedup, T_N, and the number N of processors.

[2]The occupancy rate measures the utilization of the machine CPUs

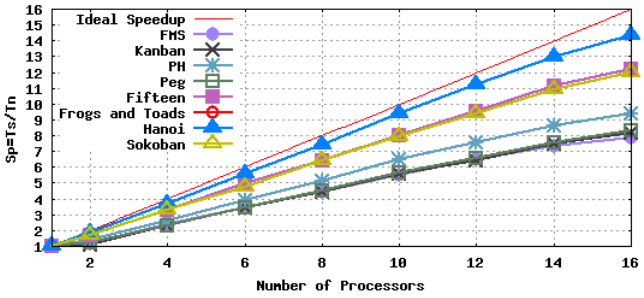

Figure 3. Speedup analysis.

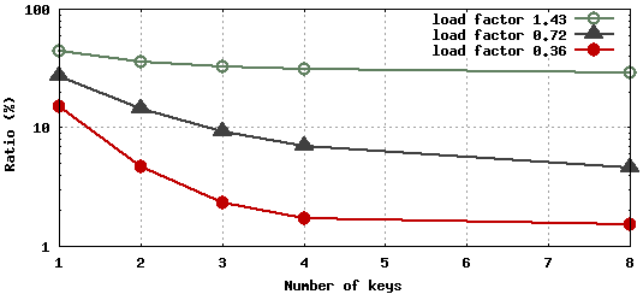

Figure 4. Collisions vs *LT* load factor.

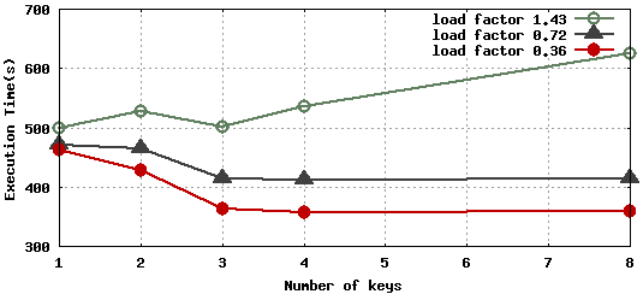

Figure 5. Performance vs *LT* load factor.

dynamically assign more states than others. Finally, our experiments are also affected by the Non-Uniform Memory Access (NUMA) architecture of our machine, where the latency and bandwidth characteristics of memory actions depend on the processor or memory region being accessed.

B. Localization Table Configuration and Memory Footprint

The *LT* data structure is configured using two parameters: its dimension (n) and the number of hash-functions keys (k). The values of these parameters have an impact on the performance. If the dimension is to small, the *LT* will get quickly saturated and the number of collisions will increase. Ideally, a *LT* of size n is sufficient for a space of n states. However, hash functions are not perfect (uniform), which affects our structure just like with standard hash table. In our experiments, we observe that *LT* behaves well for load factors (ratio between the number of states in the *LT* and its dimension) lower than 0.7.

In Fig. 4, we display the ratio (in percentage) between the number of collisions and the size of the state space, on the Kanban model, for three different values of the load factor and for different values of k.

Likewise, in Fig. 5 we show the impact of different load factors (choice of the size of the *LT*) on the execution time of the algorithm for a fixed model. Once again, we observed that our *LT* gives better results when the load factor is in average smaller than 0.7.

For the speedup results given in this Section, we have adjusted the dimension of the *LT* to obtain load factors smaller than 0.5 for every models and we have chosen to limit ourselves to at most four hash-functions keys ($k \leq 4$). We decided to fix these settings beforehand in order to not artificially improve our results and also to show the memory efficiency of our solution. To illustrate this point, we may observe that in the experiments of Fig. 5, the size of the *LT* is of 1GB (that is approximately one billions data items) for a load factor of 0.36, which is the only significant memory overhead used by our solution.

C. Comparison With Other Tools and Other Algorithms

We conclude this section with a comparison with other algorithms. We developed our own implementation of some classical parallel algorithms based on the use of hash tables. In Fig. 6, we briefly describe the different implementations used for this comparison. We decided to skip a comparison with our previous work [23] because our current results are at least twice as best.

Name	Description
LT	Distributed Table instrumented with our Localization Table
Vector	Vector of integers like structure: Localization Table with only one key
Static	States are distributed using a static slicing function
Lockless	Lockless shared hash table as the shared space
TBB	Unordered hash map as the shared space, from Intel-Threading Building Blocks library

Figure 6. Algorithms selected for benchmark comparison.

Figure 7 shows the average speedups, over all models, for the different implementations. The **Lockless** implementation has the best performance but it is an unsafe solution, since states may be skipped [3]. All the other implementations are safe. We include the results for **Lockless** since it provides a good reference for performance. Our algorithm (**LT**) performs better than all the other implementations for all models. As we mentioned earlier, the difference in performance between **Vector** and **LT** is mainly due to the non-uniformity of hash functions. This different is significant especially for Sokoban and Kanban models (see Kanban analysis at Fig 5). Concerning **Static**, an explanation for the better performances is that we exchange less states between

processors: in some experiments with **Static**, we can observe that up to 96% of the states have not been found by their owner. The gain in performance compared to **TBB** (based on an lockless, non-lossy hash table found in the Intel-TBB [21]) may be explained by the adequacy of our data structure to our targeted application (state space generation). Indeed, in this application, we have many more reads than writes (state spaces have more transitions than states). The *LT* has several benefits in this case: (1) it delivers a low complexity mechanism to grant exclusive write access for the local hash-table; (2) the structure is cache-friendly since data are stored in-place (avoiding pointers); and (3) the use of local hash tables improves memory affinity, which is important for NUMA machines.

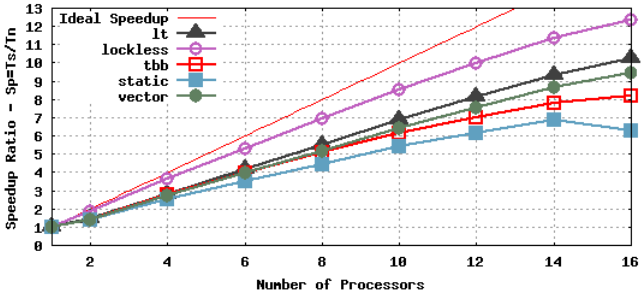

Figure 7. Average Speedup analysis.

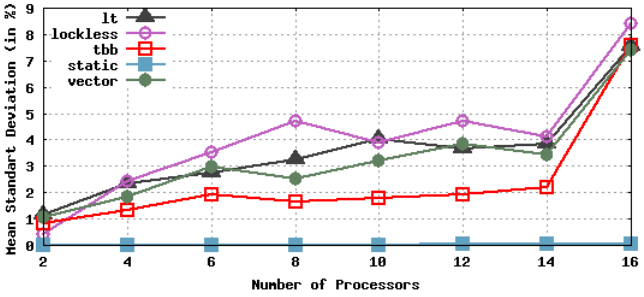

Figure 8. Average Mean-Standard Deviation analysis.

Concerning the memory distribution, we display the average mean-standard deviation for all implementations in Fig. 8. The results show that the best distribution, by far, is from the **Static** version. We can observe that all other implementations have similar distributions. (The anomalous values for $N = 16$ can be explained by the fact that, in this case, we use all the processors of our computer.) In the context of this work, we use no heuristics to ensure an uniform partition of states, so the quality of the distribution depends on the model "degree of concurrency" and the performance of processors. A complete report on the results of our benchmark is available in [22].

We have also compared our implementation with "state of the art" verification tools that provide a parallel implementation. We looked both at the Spin and DiVinE tools. We give some performance results but it is difficult to make a fair comparison. For one thing, it has proved difficult to port available implementations on the configuration used for our experiment. For instance, our benchmark with DiVinE and Spin are based on Linux instead of Solaris, which means that we take advantage of more efficient librairies. On the other hand, a major discrepancy lies in the fact that we compare an algorithm with a tool. For instance, we do not make use of any "general optimizations" techniques, such as local caches, data-alignment optimizations, etc. Also, while Spin work with compiled models, we currently use interpreted models. DiVinE accepts both models but we use for this comparison their interpreted variant. After these words of caution on the significance of the comparison, we give some results obtained on the Sokoban model. Using the parallel version of Spin on our benchmarks, we observe a maximum speedup of 3.6 using 8 cores ($73s$). Nonetheless, the sequential performance of Spin ($264s$) is about 3 times better than our prototype implementation of **LT**. In our experiments, **LT** is marginally faster than Parallel Spin when both are running on 12 cores and is faster using the 16 available cores. The computation for Spin is of $81.2s$ for 12 cores and 82.3 for 16 cores, while we generate the state space in $80s$ with 12 cores and $64s$ with 16 cores using **LT**. Concerning DiVinE – whose sequential performance is about 40% better than our prototype implementation – **LT** matches the performance of DiVinE when both are using 10 cores and outperforms it of about 20% using 16 cores. More precisely, the running time for DiVinE is of $96.5s$ with 10 cores – while **LT** time is of $96s$ – and $84.9s$ with 16 cores. It is possible to connect this result with the comparison given in Fig. 7. Indeed, DiVinE is based on a static slicing function to distribute the states – as in the **Static** implementation of Fig. 6 – and the difference of performance between **LT** and **Static** is of about 30% on 10 cores and of almost 70% for 16 cores. These preliminary results against two of the most popular parallel model-checker are very encouraging since we have a prototype implementation of **LT**.

V. Conclusion

We define a new parallel state space construction algorithm targeted at shared memory machines. The main innovation lies in a new data structure, named *Localization Table*, that is used to coordinate a network of local hash table in order to obtain an efficient concurrent hash map. This new structure replaces the Bloom Filter that was used in one of our previous work [23]. The *LT* is used to dynamically assign newly generated states and behaves as an associative array that returns the identity of the processor that owns a given state.

A first implementation of our algorithm shows promising results as we observed speedups consistently better than with other parallel algorithms. For instance, our experimental results show that efficiency varies between 90% and 50%, depending on the "degree of concurrency" of the model. In

addition, our memory footprint is almost negligible when compared to the total memory used for storing the state space. For example, in the worst case (the Kanban model, with 25GB) we consume less than 4% of the memory for the *LT* and the different stacks used by our algorithm. That is approximately 1GB of memory. The same benchmark also shows that our implementation fares well when compared with related tools. Indeed, our experiments show that our solution performed very well against an industrial strength lockless hash table, the concurrent hash map implementation provided in the Intel-TBB. This may be explained by the fact that we provide a concurrent data structure for encoding sets that is optimized for the case where deletions are very rare and the same item may be inserted several times, whereas the Intel-TBB provide a general implementation. This is a very encouraging since we obtained these results with minimal optimizations (i.e. without resorting to global caches, data-alignment optimizations, etc.), so there is still room for improvements. Altogether, our solution fullfilled our goal of having both the best temporal and spatial balance as possible.

For future works, we are investigating a *probabilistic* version of our current exhaustive algorithm. In this context, the adjective probabilistic stands for an algorithm that builds an underapproximation of the global state space, with a very high probability of building the exact state space (by very high, we mean a probability of failure less than 10^{-30}). The idea, basically, is to use an enhanced *Localization Table* where only potential false positives are stored.

To conclude, we believe that our *Localization Table* can be of great interest outside the domain of parallel model-checking algorithm. For this reason, we are planning to provide a functional API of our distributed hash table, completely self contained, that could be used in other situations and that will only require minimal configuration.

REFERENCES

[1] Allmaier, S., Dalibor, S., Kreische, D.: Parallel graph generation algorithms for shared and distributed memory machines. In: Proceedings of the Parallel Computing Conference PARCO. vol. 97. Citeseer

[2] Barnat, J., Brim, L., Ročkai, P.: Scalable multi-core ltl model-checking. In: Model Checking Software. LNCS, vol. 4595, pp. 187–203. Springer (2007)

[3] Barnat, J., Rockai, P.: Shared hash tables in parallel model checking. Electronic Notes in Theoretical Computer Science 198(1) (2008), proceedings of the 6th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2007)

[4] Berger, E., McKinley, K., Blumofe, R., Wilson, P.: Hoard: A scalable memory allocator for multithreaded applications. ACM SIGPLAN Notices 35(11) (2000)

[5] Broder, A., Mitzenmacher, M.: Network applications of bloom filters: A survey. Internet Mathematics 1(4) (2004)

[6] Butenhof, D.: Programming with POSIX threads. Addison-Wesley (1997)

[7] Ciardo, G., Gluckman, J., Nicol, D.: Distributed state space generation of discrete-state stochastic models. INFORMS Journal on Computing 10(1) (1998)

[8] Clarke, E., Grumberg, O., Peled, D.: Model checking. Springer (1999)

[9] Ezekiel, J., Lüttgen, G.: Measuring and evaluating parallel state-space exploration algorithms. In: Parallel and Distributed Methods in verification. ENTCS, vol. 198 (2008)

[10] Garavel, H., Mateescu, R., Smarandache, I.: Parallel State Space Construction for Model-Checking. In: SPIN workshop on Model checking of software. LNCS, vol. 2057 (2001)

[11] Holzmann, G.: A stack-slicing algorithm for multi-core model checking. Electronic Notes in Theoretical Computer Science 198(1), 3–16 (2008)

[12] Holzmann, G., Bosnacki, D.: The design of a multicore extension of the SPIN model checker. IEEE Transactions on Software Engineering pp. 659–674 (2007)

[13] Inggs, C.P., Barringer, H.: Effective state exploration for model checking on a shared memory architecture. In: Parallel and Distributed Model Checking. ENTCS, vol. 68(4) (2002)

[14] Jenkins, B.: Hash Functions. "Algorithm Alley". Dr Dobb's Journal (1997)

[15] Laarman, A.W., van de Pol, J.C., Weber, M.: Boosting multi-core reachability performance with shared hash tables. In: Proc. of the 10th International Conference on Formal Methods in Computer-Aided Design. IEEE Computer Society (2010)

[16] Miner, A., Ciardo, G.: Efficient reachability set generation and storage using decision diagrams. In: Application and Theory of Petri Nets. LNCS, vol. 1639. Springer (1999)

[17] Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE 77(4), 541 –580 (apr 1989)

[18] Nicol, D., Ciardo, G.: Automated Parallelization of Discrete State-Space Generation* 1. Journal of Parallel and Distributed Computing 47(2), 153–167 (1997)

[19] Pelánek, R.: Beem: benchmarks for explicit model checkers. In: Proceedings of the 14th international SPIN conference on Model checking software. pp. 263–267. Springer-Verlag (2007)

[20] Petcu, D.: Parallel explicit state reachability analysis and state space construction. In: Symposium on Parallel and Distributed Computing. IEEE (2003)

[21] Reinders, J.: Intel threading building blocks. O'Reilly (2007)

[22] Saad, R.T.: Localization table benchmark for parallel state space construction, `http://homepages.laas.fr/rsaad`

[23] Saad, R.T., Zilio, S.D., Berthomieu, B.: A general lock-free algorithm for parallel state space construction (2010), proceedings of the 9th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2010)

[24] Stern, U., Dill, D.: Parallelizing the Murϕ verifier. In: Computer Aided Verification. LNCS, vol. 1254. Springer (1997)