A Dynamic, Lock free Data Dictionary for Parallel State Space Construction

Rodrigo Tacla Saad, Bernard Berthomieu, Silvano Dal Zilio

To cite this version:

Rodrigo Tacla Saad, Bernard Berthomieu, Silvano Dal Zilio. A Dynamic, Lock free Data Dictionary for Parallel State Space Construction. 2010. hal-00523188v1

HAL Id: hal-00523188
https://hal.science/hal-00523188v1

Preprint submitted on 4 Apr 2011 (v1), last revised 9 Aug 2011 (v3)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Dynamic, Lock free Data Dictionary 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 a novel algorithm for parallel state space construction based on an original 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; we do not rely on an a-priori static partition of the state space. Distribution of the states space and coordination between processes is made through the shared access to a *localization table*, that is designed to be a highly efficient, thread safe data structure. In our solution, every process keeps a share of the global state space. 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 owns a given state. With this approach, we are able to consolidate a network of local dictionaries into an (abstract) distributed data structure without sacrificing memory affinity – data that are "logically connected" are physically close to each others – and without incurring performance costs associated to the heavy use of locks to ensure data integrity.

We evaluate the performance of our algorithm based on experimental results obtained on different benchmarks and compare these results with other solutions proposed in the literature.

I. Introduction

Verification by way of model-checking is a very demanding activity in terms of computational resources. Therefore, it is not surprising that the extensive need for memory and computation power has resulted in the definition 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. Our 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 is uniform; and (2) the processor occupancy is maximal. We are only interested here in 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. A key point is to use an efficient data structure for storing the set of generated states and for testing membership in this set.

Our algorithm builds on previous work [20] and is based on the same simple design: the global state space is stored in a set of distributed dictionaries (e.g. AVL trees or hash tables), each controlled by a single 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 states are dynamically assigned to processors; we do not rely on an a-priory static partition of the state space. Distribution of the states space and coordination between processes is made through the shared access to a novel data structure named *localization table*, that is designed to be a highly efficient thread safe data structure. In our solution, processes keep disjoint shares of the global state space. 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 owns a given state. With this approach, we are able to consolidate a network of local dictionaries into an (abstract) distributed data structure without sacrificing memory affinity – data that are "logically connected" and physically close to each others – and without incurring performance costs associated to the heavy 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. (Experiments have been performed on a shared memory multiprocessors computer, but our algorithm may be adapted to different parallel or distributed architectures.) Our preliminary results are very promising. We observe performances close to those

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

obtained using an algorithm based on lockless hash tables (that may be unsafe) and outperforms an implementation based on an industrial lockless hash table (the Intel Threading Building Blocks[18]).

II. Related Work

Several approaches have been proposed, since the early 1990s, for exhaustive parallel state space exploration. These early 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 approaches were intended for execution on parallel computers.

Subsequent proposals have addressed the problem of using shared memory machines [1, 6]. Allmaier et al. [1] were among the first to implement a parallel state space construction algorithm for shared memory systems. In their design, states are stored in a concurrent balanced-tree and data consistency is solved by using locks together with a "splitting-in-advance" scheme to reduce contention. Later, with the advent of fast network connections, commodity desktop machines were used to create affordable clusters, delivering not only more memory but also more computational power. This shift in hardware industry trends was accompanied by a very active period for algorithmic research on formal verification. Many of the solutions proposed in this period [1, 15, 7, 9, 10, 17, 21] 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.

Since the mid 2000s, research have been impacted by another shift in the computer hardware industry, with the advent of affordable multicore processors. As a consequence, new parallel algorithms for model checking have been proposed. We can mention the work of Inggs et al [12], which propose a parallel algorithm based on a *work stealing* scheduling paradigm to provide dynamic load balancing without a blocking phase. In their case, the data structure – the dictionary – used to store already visited states is a global hash table. Unlike [1], access to the dictionary is not protected by locks, hence it is not possible to ensure data integrity (see [2] for a discussion on using shared hash tables for model-checking). Another interesting work is [11] – the first, to our knowledge, that specifically focus on processor – which uses the notion of irreversible transitions to divide the state graph into disjoint sub-graphs.

In the context of this work, we propose an extension of an algorithm that we defined in [20], which is based on two data structures: (1) a lock-free, shared Bloom filter to coordinate the data distribution ; and (2) local dictionaries – we use AVL trees in our implementation – to explicitly store the data. 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, deterministic, state space exploration. In the first phase (*exploration*), 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, in rare cases, falsely report that a state has already been visited (what is called a false positive), we need to handle these *collision* states in a special way. This is done in the second phase of the algorithm (*collision resolution*) where collisions are analyzed to find possible false positive. The algorithm stops when there are no more states to explore and no more collisions.

The algorithm proposed in this paper follows a similar design but 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 state. 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. While this addition may seem minor, the real contribution lies in the definition of the localization table, that provides an efficient implementation of a safe, lock free, distributed dictionary.

A. Contributions

Our contributions are of interest for two broad domain of computer science. First, in the formal verification community, we define a new algorithm for parallel state space construction able to outperforms related solutions. Our algorithm is based on an efficient lockless solution that is able to exploit parallelism in all possible cases without compromising data integrity (it is thread safe). Moreover, unlike algorithms based on slicing functions or heuristic rules, our solution is compatible with dynamic load-balancing techniques. Second, in the parallel computing community, we present a new implementation for a (safe) lock free, distributed, data-balanced, dictionary. Indeed, we believe that our definition of localization table may be interesting in its own right as a space-efficient data structure. In particular, our experiments show that localization table performs very well against industrial strength lockless hash table (the Intel Threading Building Blocks[18]

III. Description of the Parallel Algorithm

Our new algorithm fallows the same guidelines then our previous one, it is elaborated on the work-stealing paradigm and the "homogeneous" parallelization approach, where each processor performs the same steps concurrently. Work is distributed homogeneously between processors and each processor handles its own local view of the state space. Coordination between the processors is based on our new data structure called Localization Table (sec III-A). This structure is primarily used to dynamically distributes the newly discovered states among the processors according to the exploration pace; second, it behaves like an associative array for states already assigned, returning the identify of the processor that holds this state.

Figure 1 succinctly presents the algorithm structure. In Brief, the main concept of our novel solution is the encapsulation of local dictionaries through the shared Localization Table. The basic idea behind is pretty simple, take a state and ask to the Localization Table if it is **new** or **old**. If it is a newly discovered state, it will be assigned to the discoverer processor, otherwise, the Localization Table will return the identifier of the process owner. According to this, it is possible to recover this state by accessing the respective local dictionary, in other words, the dictionary that belongs to the process addressed by the given identifier. This procedure enables us to isolate each local dictionary and grant exclusive writing privilege(but concurrent read access are allowed). As a result, this isolation raises the requirement of critical sections for writing without compromise data integrity.

The first advantage of this design is a legacy of previous work, that is the absence of heavy locks. Our Localization Table is almost as simple as the Bloom Filter and its implementation can be easily accomplished using atomic actions, ensuring secure access to data. Second,the Localization Table is a cache friendly data structure because it stores all data in-place. Hence, such a structure will benefit state spaces that have more transitions than states, that is to say, models where the Localization Table is used mostly to recover the process *id* that holds a given state.

Our Localization Table is presented at Section III-A. Section III-B introduces the memory disposition scheme of our algorithm. In Section III-C, we discuss the work-sharing techniques used in our algorithm. The pseudo-code and further explanations about the algorithm is given at section III-D.

In the remainder of the text, we assume that there are N processors and that each processor is given a unique *id*, which is an integer in the interval $0..N-1$.

A. Localization Table

Our method is based on a novel data structure that assigns states and keeps a directory of this distribution. This structure is basically a simple "table" where processors id are correlated with hash keys. Its operation is simple, it receives as input a state and returns an *id*. It would be simple to implement an exact directory using a Vector of integers, correlating every position (hash key) to a unique processor identifier. Although this may be very straightforward to implement and use, it will not ensure a fine dynamic distribution of states due to the high number of collisions, for instance, states that have the same hash key. Of course, it could be circumvented by a long vector but it would result in extra memory consumption. Consequently, inspired by the *Bloom Filter*, we developed a new space-efficient probabilistic data structure in order to have a better dynamic distribution without increase the vector size. This new structure is an enriched *Bloom Filter* named *Localization Table* because for every state already member of the set, it returns a list of processors identifiers (*id* numbers).

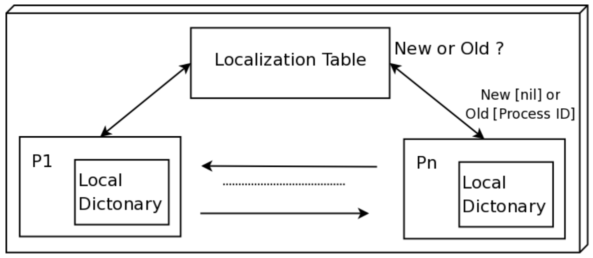

Figure 1. Algorithm abstract view.

In brief, a *Bloom filter* ([4]) is a space-efficient data structure for encoding set membership that supports two operations: insertion of an element in the set and test that an element is in the set. A filter B of size n is implemented as a vector of n bits and is associated with a series of k independent hash function $(h_i)_{i\in 1..k}$ with image in the interval $1..n$. An empty set is represented by a vector with all bits set to 0. Insertion of the element x in B is performed by setting the bits $h_i(x)$ of the vector to 1 for all i in $1..k$. Reciprocally, to query whether an element y is in B, we test that the bits $(h_i(y))_{i\in 1..k}$ are all set to 1 in the vector. If it is not the case, then we are sure that y is not in the set encoded by B. If all these bits are set to 1, then we only have a probabilistic result: in the case where y is actually not in the set, we say we have a *false positive*. The probability of false positive is a function of the size, n, number of hash functions, k, and number of elements inserted so far. Hence the parameters n and k should be carefully chosen in an implementation. Figure 2 illustrates insertion and query operations on a Bloom filter with size $n = 16$ and $k = 3$. Starting from an empty set (above), we show the result after the insertion of two elements, x and y. Element z is an example of false positive.

Our *Localization Table* inherits the same operations and properties from a normal *Bloom Filter*. It differs only because it is implemented as a vector of bytes instead of bits, providing the capacity of storing small pieces of information for each one of k independent hash functions $(h_i)_{i\in 1..k}$. This information is the unique *id* number used to identify a given processor. The insertion of an element x over the

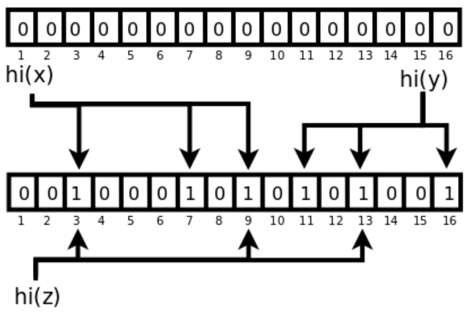

Figure 2. Illustration of some operations on a Bloom filter.

Localization Table LT is performed by setting the bytes $h_i(x)$ with the processor *id* value for all i in $1..k$. We define two types of insertion: *complete*, when all bytes are set (has the same *id*) and *incomplete*, when at least one of the bytes had already been set by a previous insertion operation (another *id*). The test operation is not only used to know if a given element is already a member of the set, it also returns the possible processors *id* that may hold it. To query whether an element y is in LT, we test that the bytes $(h_i(y))_{i \in 1..k}$ are all set. If an element had been add through a complete insertion operation, the test operation is going to return only one *id*. On the other hand, if it had been add through an incomplete insertion operation, it will return as many different *id* it had found. In both cases, complete or incomplete, only one of the processors has the element. With respect to *false positive* elements, our Localization Table has the same probability as a normal Bloom Filter and, by convention, our algorithm will route all false positive elements to the first *id* found (see sec III-D). Figure 3 illustrates the insertion and test operation over three elements (x, y, z) for three processors ($N = 3$,which are $\{P_1, P_2$ and $P_3\}$) using three independent hash function ($k = 3$).

With respect to its limitations, it is not possible to guarantee that duplications will never happen. Although it is implemented using atomic actions for an individual hash-function insertion, it is impossible to control the data race condition when two (or more) processors are inserting the same state. It can be explained by the fact that it will imply the insertion of the same sequence of k hash-functions, characterizing a classical race condition.

B. Shared and Local Data

Figure 4 illustrates the shared and local data structures used in the algorithm. It presents almost the same architecture from our previous work. In addition to the shared Localization Table and the private local dictionaries, each processor also manages two stacks of unexplored states for work-sharing: one for local work; the other for sharing work with idle processors. With respect to false positives states, they had been been separated into a different stack because they represent element collisions. Accordingly, these elements are not part yet of the set and they are going to be forward to the first processor *id* returned by the Localization Table. Such procedure will handle these elements as static assignments, forwarding it every time a different processor found one of these elements. Finally, in order to detect termination, we also manage a shared vector that stores the current state of processors (either idle or busy).

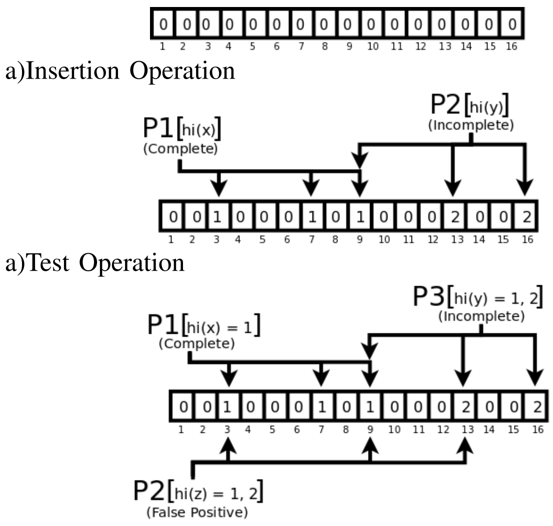

Figure 3. Illustration of some operations on a Localization Table.

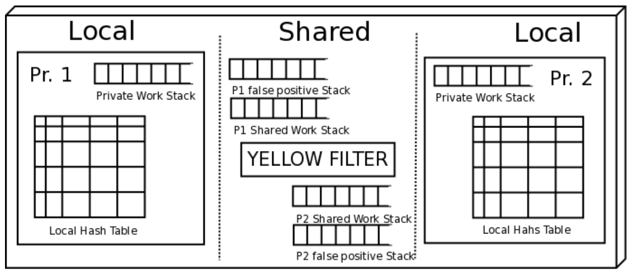

Figure 4. Shared and Private Data Model Scheme.

C. 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 in accordance with the processor occupancy. First, we use an *active* technique very similar to the work-stealing paradigm of [12]. This mechanism uses two stacks: a private stack that holds all states that should be worked upon; a shared stack for states that can be borrowed by idle processors. The 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 (*pr_work_load*), 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 *pr_work_load*;
- the share workload (*sh_work_load*), which defines the ratio of work that should be added in the shared stack if the load in the private stack is larger than *pr_work_load*.

Our implementation of the work-stealing paradigm differs from [12] by its use of unbounded shared stacks and the *sh_work_load* parameter.

D. Algorithm

As mentioned before, our solution makes use of our shared Localization Table to test whether a state may have already been discovered before, otherwise, to recovery which processor had stored it (processor *id*). To overcome the problem of false positives inherited from our inspired data structure (Bloom Filter), these elements are forward to a given processor according to the first *id* returned by the Localization Table. This procedure will prevent others processors from storing this state many times, handling these states as static assignments. The rest of the algorithm works like a common exploration engine, that is, if the membership test fails for a given state, it explores its successors until exhaustion.

In the remaining of this section, we define our algorithm using pseudo-code. The data structures used in the algorithm are composed of shared and local elements. Shared variables are: (1) the Localization Table *LT*, used to test whether a state had already been discovered or not; (2) the bitvector *V*, that stores the state of the processor (0 for idle and 1 for busy); (3) the shared stacks *Shared_Stack*[0], $\ldots$, *Shared_Stack*[N-1]; and (4) the false positive stacks *Positive_Stack*[0], $\ldots$, *Positive_Stack*[N-1]. Processor-local variables are the private stack, *private_stack*, of unexplored states and one Hash Table, *local_table*, to store states discovered by this processor.

1) Pseudo-code: The state space exploration proceeds until no new states can be added to the Localization Table *LT*. During the exploration, all states appointed by *LT* as newly discovered states are stored locally in the *local_table*. On the opposite, every time *LT* returns a list of *id's*, the process performs look-up operations over the *local_table* of the processors identified by these *id's*. If the state had not been found in one of these processors, it will be labeled as a *false positive* and forward, through the *Positive_Stack*, to the first id returned by *LT*. These states are marked with a special tag because they bypass the *LT* membership test and are directly inserted at the *local_table*. Figure 5 depicts the algorithm pseudo-code.

2) Termination Detection: Termination detection is ensured through a simple test over the bit vector *V*, which holds the states of processors, and consumes no resources. The algorithm may safely finish if all stacks of all processors are empty, in other words, if there is no more states to explore.

IV. Experiments

We implemented our algorithm using the C language with Pthreads [5] for concurrency and the Hoard Library [3] for parallel memory allocation. We developed our Localization Table with supports for concurrent insertion. The library makes use of Bob Jenkins's hash function [13]. Experimental results presented in this section were obtained on a Sun Fire x4600 M2 Server, configured with 8 dual core opteron processors and 208GB of RAM memory, running the Solaris 10 operating system.

The finite state systems chosen for our benchmarks were took from two sources. We experiment three classical examples of Petri Nets from [14] together with 5 Puzzles models from the BEEM database [16]. Figure 6 presents all selected models highlighting their respective version. The first column illustrates the abbreviations used fallowed by a brief description. The version chosen for each model was motivated by the number of states, we selected models with less than 5.10^8 states. The last columns gives the source from where each model was extracted. We give several results detailing the performance of our implementation. While speedup is the obvious criteria when dealing with parallel algorithm, we also study the memory footprint of our approach and the physical state distribution over all processors.

Model States	Description	Parameter	Source
PH 14.10^7	Dining Philosophers	*13 subnets*	[14]
FMS 24.10^7	Flexible Manufacturing System	*initial marking weight 8*	[14]
Kanban 38.10^7	Kanban System	*initial marking weight 9*	[14]
PEG 18.10^7	Peg-Solitaire Game	*version=2, crossways=1*	[16]
Sokoban 7.10^7	Computer Maze Game	*version=2*	[16]
Hanoi 38.10^7	Tower of Hanoi puzzle	*n=17*	[16]
Fifteen 23.10^7	Sam Lloyd's fifteen puzzle	*cols=4, rows=3*	[16]
Frog 53.10^7	2D Toads and Frogs puzzle	*n=6, m=5*	[16]

Figure 6. Benchmark Examples.

A. Localization Table Configuration and Memory Footprint

The Localization Table (LT) is configured by setting two parameter: its dimension (n) and the number of hash-functions keys (k). These setting must take into account the

```
while least one process
          is busy do
     while private_stack is not empty do
          s := pop(private_stack);

          if s is not tagged
               // Question LT about s
               result := LT.test_or_insert_with_my_id(s,my_id)
          else
               // False Positive, bypass LT
               result := 0
          endif

          if list_size(result) <> 0
               // Old Value, iterate over i processors
               // Initialize bool variable named found
               found := false
               for each i of result
                    if search s over local_table of processor i
                         // s found at processor i
                         found := true
                    endif
               endfor
               if not found
                    // False Positive state, forward it
                    mark s with special tag
                    // Get the first id from the list
                    first_id := list_head(result)
                    insert s in the false positive stack of processor first_id
               endif
          else
               // New Value, insert at local table
               search_and_insert s into local_table;
               let s1,..,sj,...,sn = successors(s) where
                    j = shared_work_load x n
                    if size(private_stack)
                                   > private_work_load then
                         // Share a percentage of new work
                         // Protected action by locks
                         insert s1,...,sj in my shared_stack
                         insert sj+1,...,sn in my private_stack
                         if some processor is sleeping
                              wake him up
                         endif
                    else
                              insert s1,...,sn in my private_stack
                    endif
               endlet
          endif
     endwhile
     // private stack empty
     if my shared stack is not empty then
          transfer work from my shared to my private stack
     else if my false_positive stack is not empty then
          transfer work from my false_positive to my private  stack
     else
          look for a non empty shared_stack
                    to transfer work ;
          if all shared_stacks are empty
                    and at least one processor busy
          then enter into sleep mode
          endif
     endif
endwhile
// Everybody is idle
wake up all processors and Terminate
```

Figure 5.   Algorithm pseudo-code.

size of the state space in order to prevent the saturation of LT before the exploration is finished. If the LT got saturated - almost all bytes are set, it will increase the number of false positive states and, by consequence, the size of the false positive stacks (*Positive_Stack*). As mentioned at section III-D, false positive states are handled as static assignments and therefore the performance will be severely affected. For this benchmark, we worked with a LT with the size of 1GB ($n = 1.10^9$ bytes) and *up* to 18 hash-functions ($k = 18$). Our LT had been adapted to automatically adjust the number of hash-functions keys up to a user defined number, as long as the seeds are provided. In all of ours experimentation, we never needed to change these settings. In practice, this means that users do not need to adjust any parameter of the tool before using it. Indeed, these configuration is related to the available memory space, in our case, we dimensioned our benchmark according to our machine which represents examples with 5.10^8 states.

With respect to the memory footprint of our algorithm, the only significant extra memory come from the memory space occupied by Localization Table itself. As an illustration, we have an extra memory usage of 1GB for the configuration employed in this benchmark.

B. Speedup

Figure IV-B gives the observed speedup of our algorithm when generating the state space for the models presented at Figure 6 with different number of processors. We give the relative speedup, measured as the ration between the execution time using N processors (T_N) and the time of the same algorithm on one processor. Note that, although our algorithm delivers some promising speedups, it shows different behaviors according to the model employed. While we obtained an efficiency[1] of 97% for the Hanoi model, for the Kanban example we observed a efficiency of 52%. Clearly, the algorithm is very dependent on the "degree of concurrency" of the model: it is not necessary to use lots of processors for a model with few concurrent actions. This is an inherent limitation of parallel state space construction algorithm [8].

C. State Distribution

The parallel state space construction must take into account the state space distribution, that is to say, an homogeneous distribution of states per processor. Our algorithm primes for the dynamic distribution according to the exploration pace. In addition to the "degree of concurrency" of the model, the state space distribution may also be affected by the processor performance, that is, a processor that handles simple states, smaller work units, may assign more states than others. Our experiments are also affected by the Non-Uniform Memory Access (NUMA) architecture of our

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

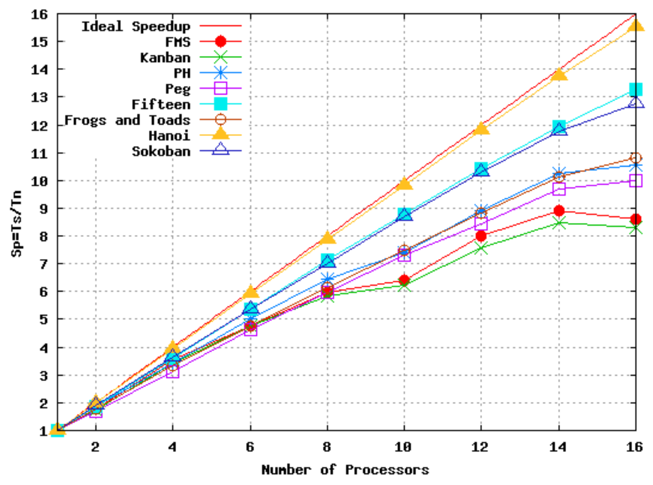

Figure 7. Speedup analysis.

machine, where the latency and bandwidth characteristics of memory actions depend on the processor or memory region being accessed. Practically, this means that the shared, addressable memory space is divided into several regions, reachable through physically different buses. Consequently, the distribution of states is by nature non-deterministic and there is no guarantee for an homogeneous distribution.

In order to evaluate the quality of the distribution, Figure IV-C depicts the mean standard deviation in % for every experimentation. This percentage shows how much variation there is from the "average" and it is obtained by dividing the standard deviation $\alpha = \sqrt{\sum_{i=1}^{N}(s_i - \overline{s})^2/N}$ by the mean value $\overline{s} = S/N$, where S is the size of the state space. The analysis of this measure shows that the distribution is affected by the concerned model and it is not possible to guarantee an homogeneous distribution. To demonstrate this variation, Figure IV-C presents the distribution of states over 16 processors for the **Sokoban** and **Hanoi** models, these are the best and the worst distribution obtained, respectively. As you can notice, the **Sokoban** model presents a complete irregular distribution where the first two processors have, in average, 40% more states then the mean value ($\overline{s}$). By contrast, the **Hanoi** model presents a regular distribution with a mean standard deviation smaller than 3%.

D. Comparison

We conclude this section on experimental results with a comparison with previously existing algorithms. It has proved difficult to port available implementations on the configuration used for our experiment. As a result, we developed our own implementation of some classical algorithms described in the literature. To start with, we implemented a variant of our code using a Vector of integers as the Localization Table to compare which one holds a better dynamic distribution using the same amount of memory (see

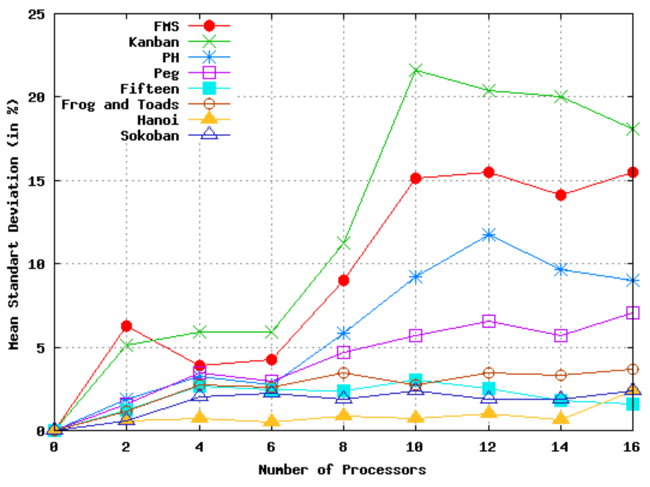

Figure 8. Mean Standard Distribution Analysis.

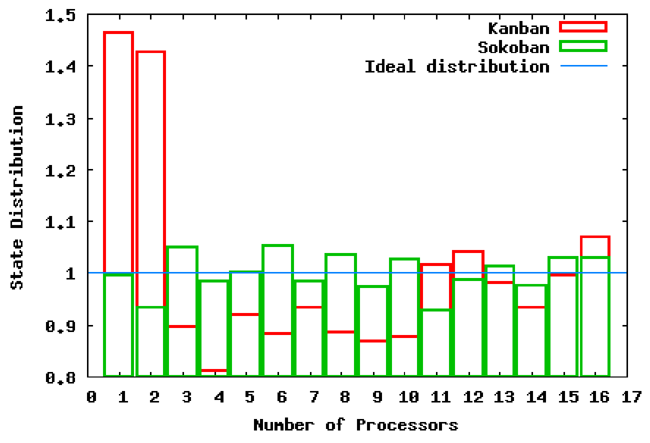

Figure 9. Physical State Distribution over 16 processors.

section III-A). In the second place, we have the well known static hash partition solution, which had been widely used by the community because of its simplicity and satisfactory results. Moreover, even if the lockless shared hash table as the shared space is not a safe solution, it is still part of our comparison because it is a good reference for performance comparison. We conclude our benchmark with an implementation using an industrial lockless non-lossy hash table (Intel TBB [18]) to demonstrate all the potential or our solution. It is important to mention that we decided to skip a comparison with our previous work because our results are at least twice as better. This gain is performance was achieved by removing the phased characteristic of the algorithm, which implied in superfluous overheads. Figure IV-D briefly illustrates all implementations with their respective abbreviations and descriptions.

Figure 11 depicts the results obtained from Kanban and Hanoi models for all 5 algorithm versions. The rest of the

Name	Description
Dist	Distributed Table instrumented with our Localization Table
Vector	Localization Table replaced by a simple Vector of integers
Static	States are distributed according to a static Hash function
Lockless	Lockless shared hash table as the shared space
TBB	Unordered hash map as the shared space, which is part of the Intel-threading blocs library

Figure 10. Algorithms selected for benchmark comparison.

benchmark is presented at appendix A. The first column shows the absolute speedup, which takes into account the time of a sequential optimized version instead of the time of the same algorithm on one processor. We decide to use the absolute speedup rather than the relative in order to provide a fairer comparison. As can be noticed, not only our algorithm (**DIST**) matched the performance of **TBB**, **Static** and **Vector** for all models but also yields a better performance of 10% for 6 (over 8) models from our benchmark. As might be expected, the **Lockless** version holds the best performance for all models. Even so, among the safe solutions presented here, this benchmark position our algorithm as the most effective for parallel state space construction. The second column depicts the mean standard deviation of all algorithm in order to evaluate the quality of the distribution. This second analysis shows that the best distribution is given by far from the **Static** version. Thereafter, it is hard to state which one has the second best distribution. Note that the **Dist** and **TBB** systematically alternate the position of second best distribution. As an illustration, our solution holds the second best distribution (in average) for **Peg**, **Fifteen**, **Frog** and **Sokoban** models. Regarding the **Lockless** version, it consistently holds the worst distribution for all models. In what concerns the number of collisions and false positives of our algorithm, they are practically negligible. To put in figure, in the worst case (**Frog**) we got in average 1 collision for each 25.10^7 states and less then 1000 false positives states for 5.10^8 states. All the information concerning this benchmark is available at [19].

V. Conclusion

We presented a new parallel state space construction algorithm target at shared memory machines. This algorithm is an extension of a previous work [20] based on a singular architecture defined as a small shared space supplemented by local dictionaries. It innovates by proposing a novel data structure, named *Localization Table*, to replace the Bloom Filter previously used for the shared space. This new data structure is used to dynamically assign newly discovered states and behaves as an associative array that

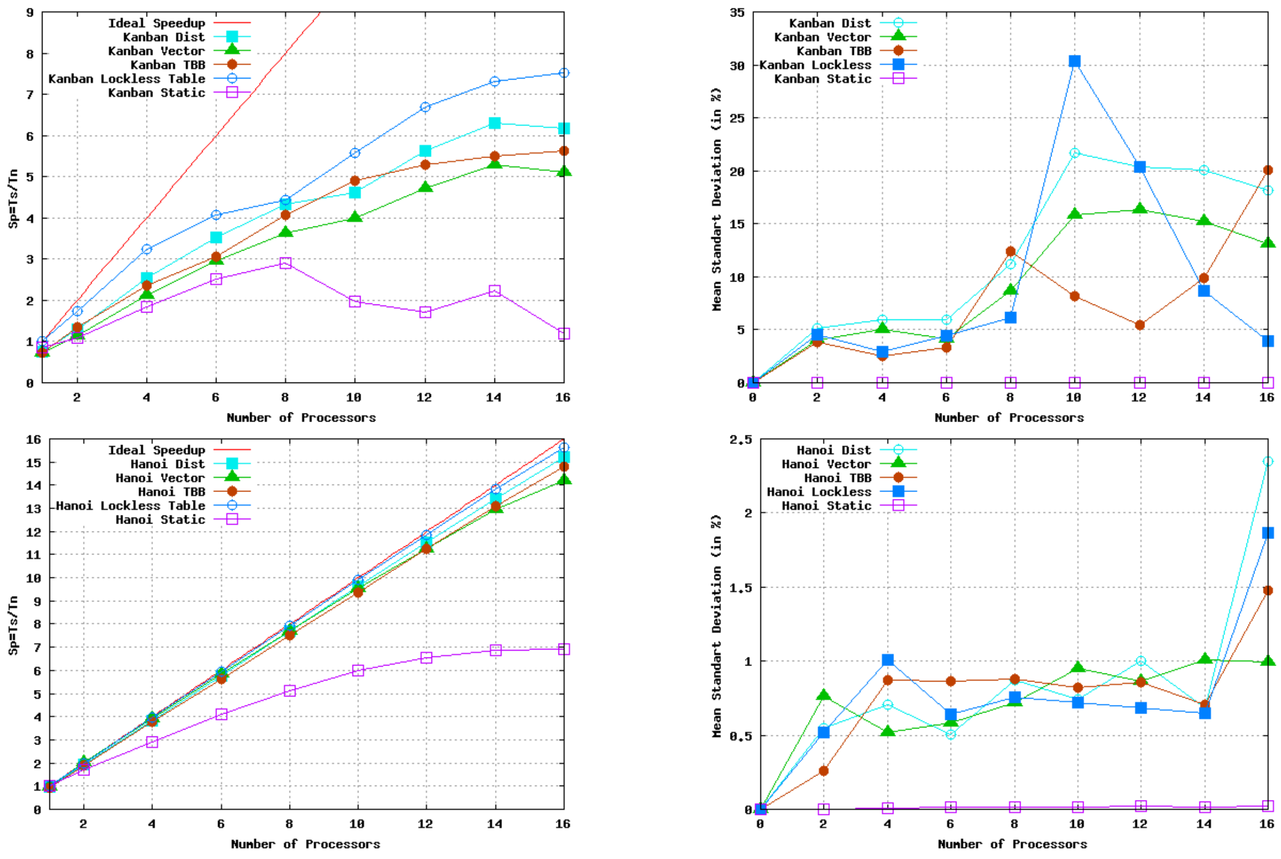

Figure 11. Algorithm Comparison for Kanban and Hanoi models.

returns the identity of the processor that owns a given state. Consequently, it eliminates the need of an phased algorithm because the collisions are solved on-the-fly. Moreover, our algorithm delivers a new type of distributed hash table with the distinction that states are dynamically assigned; without use of locks and open for different types of work-sharing techniques.

From the benchmark performed, our algorithm proved capable of promising speedups whenever was possible. For instance, the results show that our efficiency varies between 95% and 50%, depending on the "degree of concurrence" of the model. In addition, our memory footprint is almost negligible when compared to all memory expanded, for example, it represented in the worst case (Kanban - 60GB) less then 1.7% of the memory used. Regarding the number of duplications, it proved really rare to happen, and when it happened it was in the order of 1 for each 25.10^7 states. This benchmark also showed that our algorithm is well positioned when compared with related solutions. Our algorithm not only matched all safe solution but also yielded a better performance of 10% for 6 (over 8) models from our benchmark. Altogether, it shows that our solution attended our expectations of having 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 Bloom Filter data-structure, like our *Localization Table*, where only potential false positives are stored. Finally, we understand that our *Localization Table* can be of great interesting for a broader community. For this reason, we are planning to provide a functional API of our distributed hash table, completed self contained, where the user will just configure its dimension (n) and the max number of hash-functions keys (k). This configuration will be required only once and it will be related to the amount of available memory.

REFERENCES

[1] Allmaier, S., Kowarschik, M., Horton, G.: State space construction and steady-state solution of GSPNs on a shared-memory multiprocessor. In: Workshop on Petri Nets and Performance Models (1997)

[2] 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)

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

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

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

[6] Caselli, S., Conte, G., Bonardi, F., Fontanesi, M.: Experiences on SIMD massively parallel GSPN analysis. In: Computer Performance Evaluation Modelling Techniques and Tools. LNCS, vol. 794. Springer (1994)

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

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

[9] Flavio Lerda, R.S.: Distributed-memory model checking with spin. In: Theoretical and Practical Aspects of SPIN Model Checking. Springer (1999)

[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., Bosnacki, D.: Multi-core model checking with SPIN. HIPS-TopModels 2007, short paper (2007)

[12] 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)

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

[14] 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)

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

[16] 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, Berlin, Heidelberg (2007)

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

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

[19] Saad, R.T.: Benchmark comparison of safe and unsafe solutions for parallel state space construction. (Aug 2010), `http://homepages.laas.fr/rsaad/`

[20] 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)

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

Appendix

This appendix complement the benchmark analysis of section IV-D. It presents the speed-up and mean standard distribution measures for the **FMS**, **PH**, **Solitaire**, **Fifteen**, **Sokoban** and **Frogs** models. As you can note, except for model **Frog**, our solution holds better performance when compared to related safe algorithms.

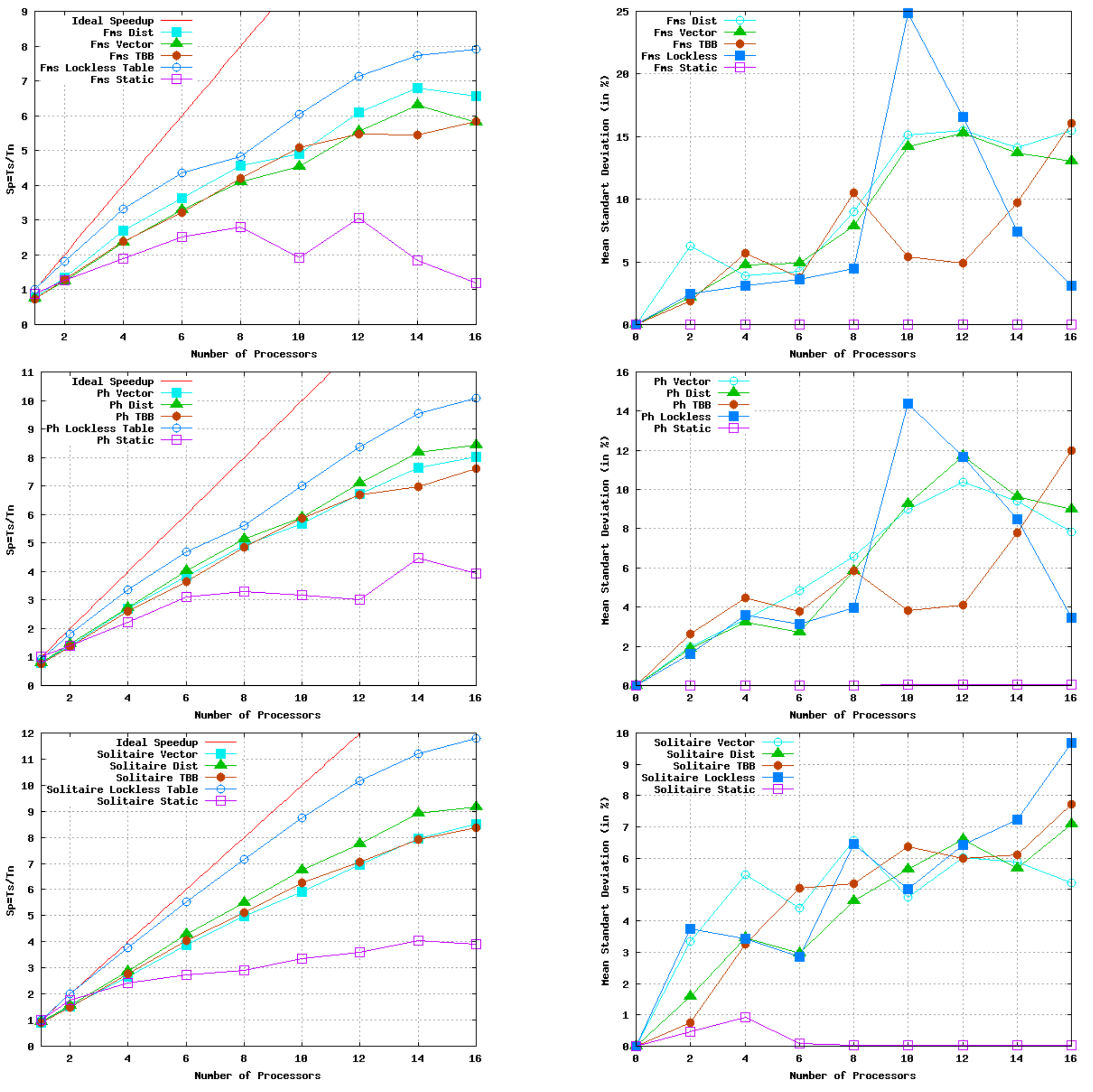

Figure 12. Algorithm Comparison for FMS, PH and Solitaire models

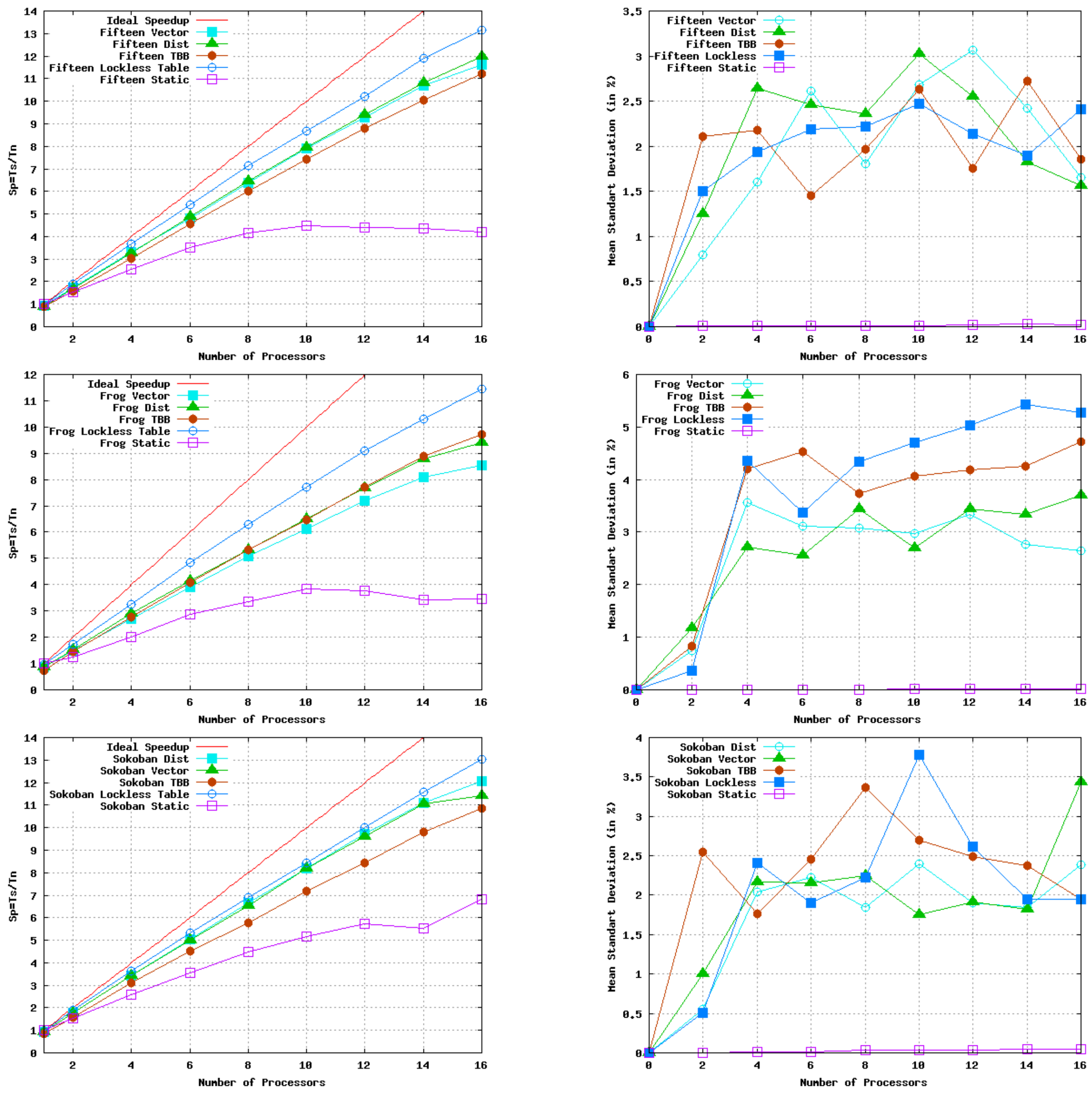

Figure 13. Algorithm Comparison for Fifteen, Frog and Sokoban models