Memory-Efficient Kronecker Algorithms with Applications to the Modelling of Parallel Systems

Anne Benoit$^{(1)}$, Brigitte Plateau$^{(1)}$ and W.J. Stewart$^{(2)}$

$^{(1)}$ Informatique et Distribution, INRIA Rhones Alpes
51, Av. Jean Kuntzman, 38330 Montbonnot, France
$^{(2)}$ Department of Computer Science, NC State University, Raleigh, NC, USA.

Abstract

We present a new algorithm for computing the solution of large Markov chain models whose generators can be represented in the form of a generalized tensor algebra, such as networks of stochastic automata. The tensor structure inherently involves a product state space but inside this product state space, the actual reachable state space can be much smaller. For such cases, we propose an improvement of the standard numerical algorithm, the so-called "shuffle algorithm", which necessitates only vectors of the size of the actual state space. With this contribution, numerical algorithms based on tensor products can now handle much larger models.

Keywords: large and sparse Markov chains, stochastic automata networks, generalized tensor algebra, vector-descriptor multiplication, shuffle algorithm

1 Introduction

Continuous Time Markov Chains (CTMC) facilitate the performance analysis of dynamic systems in many areas of application [1], and are particularly well adapted to the study of parallel and distributed systems, [2–4]. They are often used in a high level formalism in which a software package is employed to generate the state space and the infinitesimal generator of the CTMC, as well as to compute stationary and transient solutions. Several high-level formalisms have been proposed to help model very large and complex CTMCs in a compact and structured manner. For example, stochastic automata networks (SANs), [5, 6], queueing networks, [7], generalized stochastic Petri nets, [8], stochastic reward nets, [9] and stochastic activity nets, [10] are, thanks to their extensive modelling capabilities, widely used in diverse application domains, and notably in the areas of parallel and distributed systems. It is therefore possible to generate the CTMC that represents the system to be studied from formalisms such as these which model entire systems from their interconnected subcomponents. Such formalisms often involve a product state space $\hat{S}$ but inside this product state space, the actual reachable state space S can be much smaller ($S \subseteq \hat{S}$).

In this paper, our concern is with the computation of the stationary probability vector $\pi \in R^{|S|}$, a row vector whose i^{th} element π_i is the probability of being in state i of the CTMC at a time that is sufficiently long for all influence of the initial starting state to have been erased, and where S is the set of states of the CTMC. The vector π is the solution of the system of linear equations $\pi Q = 0$, subject to $\pi e = 1$, where Q is the generator matrix of the CTMC and e is a vector whose elements are all equal to 1.

The primary difficulty in developing software to handle large-scale Markov chains comes from the explosion in the number of states that usually occurs when certain model parameters are augmented. Indeed, CTMCs which model real parallel and distributed systems are usually huge and sophisticated algorithms are needed to handle them. Both the amount of available memory and the time taken to generate them and to compute their solutions need to be carefully analyzed. For example, direct solution methods, such as Gaussian elimination, are generally not used because the amount of fill-in that occurs necessitates a prohibitive amount of storage space. Iterative methods, which can take advantage of sparse storage techniques to hold the infinitesimal generator, are more appropriate, even though here also, memory requirements can become too large for real life models.

During the generation of the CTMC from the high-level formalism, particular attention must be paid to the way in which the quantities S, Q and π are stored. Indeed, even though the computational cost of a particular numerical solution method may be relatively high, it is essentially storage considerations that determine whether this method may be used or not. Furthermore, iterative methods often need to compute the product of a probability vector and the generator Q many many times. They therefore depend directly upon the manner in which these data are stored.

Different techniques have been proposed for storing the **set of reachable states** S and for performing efficient searches in this set. In certain cases, it is possible to define a reachability function for the model and this may be used to provide access to the reachable states in constant time, [11]. In other cases, multi-level approaches such as decision diagrams [12] may prove to be efficient. In what follows, we suppose that S is stored in an efficient manner and we will no longer be concerned with this particular aspect of the algorithms.

There have been many techniques proposed to represent the **descriptor** Q and the **probability vector** π. A first approach, independant of the high-level formalism, consists of storing the matrix in a row-wise compact sparse format: the nonzero elements of the matrix and their position in the matrix are kept. The probability vectors are the size of the reachable state space $|S|$. Efficient algorithms are available with which to compute a vector-matrix product when the matrix is stored in this fashion [13, 14]. However, for very large models, it is frequently the case that the matrix is too large to be held in memory.

Sanders and his co-workers [15] proposed to generate the elements of Q as they were needed, "on the fly" from a high level model. However, this approach can be computationally expensive, especially when the model contains immediate transitions. A different proposal is to store Q on a fast disk and to seek the elements as and when they are needed [16]. This obviously requires a disk that is sufficiently fast and sufficiently large.

Stochastic Automata Networks (SANs) were introduced by Plateau et al. ([5, 6]) to keep memory requirements manageable. These allow Markov chains models to be described in a memory efficient manner because their storage is based on a tensor formalism. Another formalism based on Stochastic Petri Nets allows us to obtain a similar tensor formalism, as shown by Donatelli in [17, 18]. However, the use of independant components connected via synchronizations and functions may produce a representation with many unreachable states ($|S| \ll |\hat{S}|$). Within this Kronecker framework, a number of algorithms have been proposed. The first and perhaps best-known, is the shuffle algorithm [5, 6, 19, 20], which computes the product but never needs the matrix explicitly. However, as has been shown previously, this algorithm needs to use vectors $\hat{\pi}$ the size of $\hat{S}$, that we shall call *extended*. We denote this algorithm **E-Sh**, for **extended shuffle**. Some alternative approachs have been proposed in [20–23]. They consist of first computing the state space S, and then solving the model by using iteration vectors π which contains entries only for these states (size of S), that we shall call *reduced*.

Another approach consists of using decision diagrams in order to represent the Markov chain generator Q. In this tree representation only distinct nonzero elements are stored, and they are kept as the leaves of the tree. On initial construction, the branches of the tree must be pruned to keep it from becoming too large. Multi-terminal binary decision diagrams (MTBDDs) and probabilistic decision graphs (PDGs) can be used in model checking, e.g. [24, 25], or in performance evaluation [26]. This is a promising approach, but its efficiency in memory requirements is directly linked to the number of distinct elements in Q. Moreover, access to the nonzero elements requires following a path from the root of the tree to the appropriate leaf, and this must be done every time an element is needed. As far as we are aware, there are no comparative studies on the effectiveness of these algorithms in actually computing solutions of Markov chains.

In order to improve the computation time, another kind of representation is presented in [27, 28]. This representation of the generator Q as a matrix diagram permits quick access to all of their elements, and the solution time is often satisfactory, even for large models. Furthermore, memory requirement to store Q are barely greater than that needed for a descriptor stored in tensor format and remains negligible compared to that needed to store a probability vector. However, techniques linked to a cache, [27] make theoretical estimations of the complexity of the algorithms difficult, whereas good theoretical results may be obtained with the Kronecker approach, [23].

When $|S| \approx |\hat{S}|$, the gain in memory obtained with the use of the tensor formalism can be enormous compared to the standard approach of an explicit sparse matrix storage. For example, if a model consists of K components of size n_i ($i = 1..K$), the infinitesimal generator is full and the space needed to store it with the standard approach is of the order of $(\prod_{i=1}^{K} n_i)^2$. The use of a tensor formalism reduces this cost to $\sum_{i=1}^{K} n_i^2$. The shuffle algorithm E-Sh is very effective in this case, as has been shown in [5, 6, 19, 20].

However, when there are many unreachable states ($|S| \ll |\hat{S}|$), E-Sh is not efficient, because of its use of extended vectors. The probability vector can therefore have many zero elements, since only states corresponding to reachable states have nonzero probability. Moreover, computations are carried out for all the elements of the vector, even those elements corresponding to unreachable states. Therefore, the gain obtained by exploiting the tensor formalism can be lost since many useless computations are performed, and memory is used for states whose probability is always zero. In this case, the approach first described, based on storing all nonzero elements of the generator, performs better since it does not carry out meaningless computations. However, its extensive memory requirements prevent its use on extremely large models.

Due to the large memory requirements of this last approach, it is worthwhile seeking a solution which takes unreachable states into account and at the same time, uses the benefits of the tensor formalism. Thus, we would like to be able to exploit the tensor formalism, even in the presence of an important number of unreachable states. Indeed, the shuffle is an efficient algorithm when $|S| \approx |\hat{S}|$, and we will show how to modify it efficiently for the case in which $|S| \ll |\hat{S}|$. It is interesting to work on this algorithm which conserves its tensor structure and for which we possess certain theoretical results, and which, in addition, is used in diverse domains (signal processing [29], partial differential equations,...).

The use of reduced vectors (of size $|S|$) permits a reduction in memory used, and some needless computations are avoided. This leads to significant memory gains when using iterative methods such as Arnoldi or GMRES which can possibly require many probability vectors. A modification to the E-Sh shuffle algorithm permits the use of such vectors. However, to obtain good performance at the computation-time level, some intermediate vectors of size $|\hat{S}|$ are also used. The algorithm Act-Sh-JCB of [23] transforms the reduced vector π into a vector $\hat{\pi}$ of size $|\hat{S}|$ before calling E-Sh.

In summary, the algorithm E-Sh is an efficient algorithm when there are few non reachable states in the system. Modifications to reduce memory needs when $|S| \ll |\hat{S}|$ have been proposed, but the new algorithms use also intermediate data structures of size $|\hat{S}|$ [23]. Our goal is to develop algorithms that maintain the desirable complexity of E-Sh, while eliminating the inconvenient data structures of size equal to the size of the product space, which, to our knowledge, has not been done before. We hope that this will minimize memory requirements at the cost of only a small increase in computation time.

In the next section we present the shuffle algorithm E-Sh for the multiplication of a vector and a SAN descriptor. We then present (in Section 3) two new versions of the shuffle algorithm to save both computations and memory. The first one is **partially reduced** (use of intermediate vectors of size $|\hat{S}|$) and we denote the corresponding algorithm **PR-Sh**. It focuses on computation time, but the saving in memory turns out to be somewhat insignificant. Another version of the algorithm concentrates on the amount of memory used, and allows us to handle even more complex models. In this new algorithm, all the intermediate data structures are stored in reduced format. We refer to this as **fully reduced** and denote the corresponding shuffle algorithm **FR-Sh**. A series of tests comparing theses shuffle algorithms is presented in Section 5. These algorithms were incorporated into the software package PEPS [11] and tested by means of this package.

2 The Shuffle Algorithm

In SANs [6, 19] and in SGSPNs [18], it has been shown that the transition matrix can be expressed as:

$$Q = D + \bigoplus_{i=1}^{N}{}_{g}\, Q_l^{(i)} + \sum_{e=1}^{E} \bigotimes_{i=1}^{N}{}_{g}\, Q_e^{(i)} = \sum_{j=1}^{(N+2E)} \left[\bigotimes_{i=1}^{N}{}_{g}\, Q_j^{(i)} \right] \qquad (1)$$

Here N is the number of automata (resp. sub Petri Nets) in the network and E is the number of synchronizing events (resp. transitions). In the first representation, D is the diagonal of the descriptor; the tensor sum corresponds to the analysis of the local events and is called the local part of the descriptor; the tensor product corresponds to the analysis of the synchronizing events and is called the synchronizing part of the descriptor. The second representation shows that the basic operation of interest in computing the steady state vector of the Markov chain using iterative methods is the product of a row vector $\hat{\pi}$ and a tensor product[1] $\bigotimes_{g\,i=1}^{N} Q^{(i)}$. This term is composed of a sequence of N matrices denoted $Q^{(i)}$ with $i \in [1 \dots N]$, each associated with an automaton $\mathcal{A}^{(i)}$. We begin by introducing some definitions concerning finite sequences of matrices:

- n_i is the order of the i^{th} matrix in a sequence;
- $nleft_i$ is the product of the order of all the matrices that are to the left of the i^{th} matrix of a sequence, i.e., $\prod_{k=1}^{i-1} n_k$ with the special case: $nleft_1 = 1$;
- $nright_i$ is the product of the order of all the matrices that are to the right of the i^{th} matrix of a sequence, i.e., $\prod_{k=i+1}^{N} n_k$ with the special case: $nright_N = 1$;

We first assume that there are no functional dependencies. According to the decomposition property of tensor products [6], every tensor product of N matrices is equivalent to the product of N normal factors. To compute the multiplication of a vector by the term $\bigotimes_{i=1}^{N} Q^{(i)}$ it is therefore sufficient to know how to multiply a row-vector $\hat{\pi}$ and a normal factor i: $\hat{\pi} \times I_{nleft_i} \otimes Q^{(i)} \otimes I_{nright_i}$. Furthermore, the property of commutativity between normal factors allows the multiplication of the normal factors in any desired order. The matrix

[1]The indices j have been omitted from the matrices $Q_j^{(i)}$ in order to simplify the notation.

$I_{nleft_i} \otimes Q^{(i)} \otimes I_{nright_i}$ is a block diagonal matrix in which the blocks are the matrix $Q^{(i)} \otimes I_{nright_i}$. We can treat the different blocks in an independent manner, which suggests the possibility of introducing parallelism into the algorithm, [30]. There are $nleft_i$ blocks of the matrix each of which is to be multiplied by a different piece of the vector, which we shall call a *vector slice*. The vector is divided according to I_{nleft_i} and so we shall call these vector slices, l-slices (for left) and denote them by $p_0, \ldots, p_{nleft_i - 1}$, each of size $nright_i \times n_i$.

Thus, the E-Sh algorithm consists of a loop over the $nleft_i$ l-slices, and at each iteration it computes the resulting l-slice, $r_l = p_l \times Q^{(i)} \otimes I_{nright_i}$. The computation of an element of the resulting l-slice corresponds to the multiplication of p_l by a column of the matrix $Q^{(i)} \otimes I_{nright_i}$. Due to the structure of this matrix, the multiplication therefore boils down to the repeated extraction of components of p_l (at distance $nright_i$ apart), forming a vector called z_{in} from these components and then multiplying z_{in} by a column of the matrix $Q^{(i)}$. Notice carefully that z_{in} is composed of elements of p_l which may not be consecutive. In a certain sense, it is a slice of p_l. The slice z_{in} corresponds to elements of p_l which must be multiplied by the elements of the column of $Q^{(i)}$. The structure of the matrix $Q^{(i)} \otimes I_{nright_i}$ informs us that we must consider $nright_i$ slices z_{in}. Thus, we number the z_{in} from 0 to $nright_i - 1$. Extracting a z_{in} is the same as accessing the vector p_l and choosing the elements at intervals of $nright_i$ positions in the vector.

Once a z_{in} has been obtained, we can compute an element of the result by multiplying the z_{in} by a column of $Q^{(i)}$. The multiplication of a z_{in} by the entire matrix $Q^{(i)}$ therefore provides several elements of the result, a slice of the result, called z_{out}. The positions of the elements of z_{out} in r_l correspond to the positions of the elements of z_{in} in p_l. We number the z_{out} in the same way as we did for z_{in}. Therefore, the multiplication of the z_{in} n° k by $Q^{(i)}$ gives z_{out} n° k.

A pseudo-code of the E-Sh can be found in [20].

Complexity: The complexity of the algorithm for multiplying a vector and a classic tensor product may be obtained by observing the number of vector–matrix multiplications that are executed [6, 19]. If we assume that the matrices $Q^{(i)}$ are stored in a sparse compacted form, and letting nz_i denote the number of nonzero elements in the matrix $Q^{(i)}$, then the complexity of E-Sh is given by

$$\prod_{i=1}^{N} n_i \times \sum_{i=1}^{N} \frac{nz_i}{n_i} = |\hat{S}| \times \sum_{i=1}^{N} \frac{nz_i}{n_i}$$

This should be compared to the multiplication of a vector and $Q = \bigotimes_{i=1}^{N} Q^{(i)}$ which consists in first computing Q, and then multiplying the result with the vector. This has complexity of the order of $|\hat{S}|^2$ when the matrices $Q^{(i)}$ are full, and $\prod_{i=1}^{N} nz_i$ for a sparse compacted format. So, E-Sh is better than this last multiplication for $nz_i \geq n_i.N^{\frac{1}{N-1}}$. The complexity of the algorithm handling functional dependencies can be found in [6, 20].

The E-Sh has two drawbacks: it uses vector data structure of size $|\hat{S}|$ and computes all vector elements, even the zero entries out of $\hat{S}$. In what follows, we remove these two drawbacks.

3 Reduced Memory Shuffle Algorithm

The improvement of the shuffle algorithm is based on the fact that all unreachable states have probability zero in the initial vector and remain zero after each product with the infinitesimal generator. Buchholz et al. [23] present algorithms in which the vectors are the size of S, but the shuffle algorithm can not be treated in this fashion, because it is possible that unreachable states have a nonzero entry in temporary, intermediate vectors. The algorithm Act-Sh-JCB therefore transforms the reduced vector into a vector of size $|\hat{S}|$ before calling E-Sh.

Data Structures Used: The fact that we shall store only the values of the vector that correspond to reachable states implies that we must keep track of the positions of these elements in the corresponding vector in $\hat{S}$. We shall assume that the set of reachable states in already known. This may be done either by applying an algorithm that explores all reachable state [12, 21], or by asking the user to provide a function which represents the set of reachable states of the system [11].

We shall let $pos_{\hat{\pi}}$ denote the set of states which have nonzero entries in any vector $\hat{\pi}$. The **probability vectors** are such that $pos_{\hat{\pi}} \subseteq S$, and we represent them with the help of two arrays of size $|S|$: the array $\hat{\pi}.vec$ contains the entry values and the array $\hat{\pi}.positions$ contains the positions of the reachable states in the corresponding vector in $\hat{S}$. It is however important to notice that processing the multiplication of a vector and a normal factor corresponding to a synchronizing event (with matrix $Q_e^{(i)}$ in Equation 1) may lead to an **intermediate vector** $\hat{\pi}$ for which we do not have $pos_{\hat{\pi}} \subseteq S$. We can equally well store these intermediate vectors in two arrays $\hat{\pi}.vec$ and $\hat{\pi}.positions$ of size $|S|$ if $|pos_{\hat{\pi}}| \leq |S|$. In this case, the positions no longer correspond to the reachable states, but they correspond to $pos_{\hat{\pi}}$. Notice that most often $|pos_{\hat{\pi}}| \leq |S|$ since the intermediate vectors are obtained via multiplication with synchronization matrices, matrices which are generally very sparse. These vectors therefore contain many zero elements. However, it may happen that $|pos_{\hat{\pi}}| > |S|$. In this case, it becomes necessary to dynamically reallocate memory in which to store the vector[2].

The principal difficulty in applying ideas of algorithm E-Sh with the new reduced vector data structure lies in extracting the slices of the vector z_{in} from an l-slice. Indeed, the previous algorithm used a skipping procedure to extract the vector slices. When the vector is stored in a reduced structure, it is not possible to perform these skips.

A first improvement focuses on the computation time. Then we will see how to reduce the memory cost.

3.1 Improvements in Computation Time (PR-Sh)

In this algorithm, all the z_{in} corresponding to an l-slice are stored in an array. The z_{in}'s are similar to those used in the extended shuffle algorithm E-Sh; they are stored in a vector of length at most $|\hat{S}|$. Thus, for each l-slice (stored in reduced format), we build an extended vector which contains the sequence of z_{in}. Each element of the l-slice belongs to a given z_{in}, and it is in a specific location in this z_{in}. The numbering of the z_{in} goes from 0 to $nright_i - 1$, and corresponds to the numbering introduced in algorithm E-Sh. The position in z_{in} simply indicates the position of an element in a given z_{in}. The data structure that holds the z_{in} is a two-dimensional array.

The different steps of the multiplication $w = v \times I_{nleft_i} \otimes Q^{(i)} \otimes I_{nright_i}$ for the partially reduced PR-Sh algorithm are detailed below. We differentiate between the nature of w (probability or intermediate vector) to optimize the performance of the algorithm.

Notice first that an l-slice is a set of consecutive elements and a single sweep over the vector v allows us to determine the limits of l-slices.
For $l = 0, \ldots, nleft_i - 1$, the l-slice p_l contains elements with index x such that $n_i \times nright_i \times l \leq v.positions(x) < n_i \times nright_i \times (l+1)$. We handle the l-slices in a sequential manner (but it can be done in parallel).

For each l-slice, we begin by extracting all the z_{in}. To do this, we perform an integer division on the values in the array $v.positions$, which gives, for j from 0 to $|S|$, the number of the z_{in} to which belongs the corresponding element ($v.positions[j]\ mod\ nright_i$), and its place in z_{in} ($v.positions[j]\ div\ nright_i$). An integer division by $nright_i$ boils down to recuperating the elements that are spaced $nright_i$ apart in the extended vector.

Once we have the z_{in}, it remains to perform the multiplications of a z_{in} by a column of $Q^{(i)}$, and then store the results in the correct place in the vector w. It is also necessary to evaluate the matrices in cases when they are functional.

When w is a **probability vector**, we know which elements will be nonzero in the solution vector obtained, and we compute only the value of these elements. When there is no functions, we traverse the resulting vector and we compute the probability for each reachable state. When extending algorithm PR-Sh to a generalized tensor product, it is necessary to proceed in a different way than a linear traversal, otherwise we need to have one evaluation of matrix per element of S. We perform the multiplication with only one evaluation per z_{in}, as it is done in E-Sh. To do this, we store some intermediate information in the following arrays:

- the array **used** specifies which z_{out} contain at least one reachable state. It is an array of booleans of size $nright_i$.
- the array **index** is a counter to keep track of how many reachable states are present in each z_{out}.
- the array **useful** indicates the position in z_{out} of an element to be computed. Only places corresponding to reachable states are noted, and they are numbered from 1 to $index[r]$ for z_{out} number r. Element $useful[r][k]$ corresponds to the position in z_{out} number r of the k^{th} reachable state of this z_{out}.
- the array **place** holds the place in the resulting vector of the elements of z_{out}. Once a value has been computed, it must be stored in the vector. Element number k in z_{out} number r will therefore be placed at $place[r][k]$ in the resulting vector.

It is then possible to compute the nonzero elements as before, but not in the same order (we first compute all those corresponding to the same z_{in}). To do this, we perform a loop on $nright_i$, and we perform the evaluation of the matrix only if this is necessary. The evaluation must be performed if we have to compute at least one corresponding element of z_{out}, i.e., if the corresponding value of *used* is *true*. Finally, we compute each element corresponding to a reachable state of this z_{out}, by carrying out a loop from 0 to $index[r] - 1$. The array *place* lets us know where to store the result in the vector, and the array $useful$ lets us know the column of the matrix to be used in the multiplication.

[2]We have not yet implemented dynamic reallocation in PEPS. This happens very rarely in our experience and is the reason why we have not had need of dynamic reallocation for the moment.

When w is an **intermediate vector**, we don't know any more which elements will be nonzero in the solution vector obtained. No longer can we perform the computation of a z_{in} by using only a column of a matrix which gives a result corresponding to a reachable state, because we do not know where the nonzero values will be in the resulting vector. In this case, we perform the multiplication of a z_{in} by the entire matrix, and we store all the nonzero elements as a (position, value) pair. When the multiplications of an l-slice have been completed, we store the nonzero elements obtained in the resulting vector, and update the arrays *positions* and *vec*.

A pseudo-code of the PR-Sh can be found in [20].

Complexity of the algorithm: This algorithm performs $|S|$ multiplications of a vector slice by a column of a matrix when w is a probability vector, instead of the $|\hat{S}|$ multiplications required by E-Sh. However, when we have an intermediate vector, we cannot reduce the number of computations. From a memory point of view, there is no real gain even if we use vectors the size of S, because the algorithm uses intermediate data structures the size of $\hat{S}$.
What we do in the following is to improve the shuffle algorithm from the perspective of memory usage, so that we have no need of any data structure of size $|\hat{S}|$.

3.2 Fully Reduced Memory Shuffle Algorithm (FR-Sh)

The idea of this new algorithm is the following: for each normal factor, the nonzero elements of the vector are reordered so that the elements of a z_{in} are consecutive.

To do this, a somewhat similar method can be adopted by traversing the vector and extracting all the z_{in} for the l-slice being treated. For this, we use an intermediate structure[3] that consists of a set of triplets (number, place, index), each triplet corresponding to an element of the vector $\hat{\pi}$: **number** represents the number of the z_{in} containing this element; **place** represents the place of the element in the z_{in}; **index** represents the index (the position) of the element in the reduced vector ($1 \leq index \leq |S|$), it allows us to find the value of the element as $\hat{\pi}.vec[index]$.

The different steps of the multiplication $w = v \ \times I_{nleft_i} \otimes Q^{(i)} \otimes I_{nright_i}$ for the FR-Sh algorithm are detailed below. We differentiate between the nature of w (probability or intermediate vector) to optimize the performance of the algorithm.

We handle the l-slices in a sequential manner as in PR-Sh (but it can be done in parallel): for each l-slice, we begin by collecting the necessary information for an efficient execution of the remainder of the algorithm. The set of triplets *infozin* contains informations on the z_{in} of the l-slice, while *infozout* give informations on the z_{out} when available (the slice to consider in this case is a l-slice of w).

Treatment of an l-slice:
1. Filling *infozin*: We first perform a traversal of the l-slice of *v.positions*, to construct a set of triplets called *infozin*. For each element, we obtain *number* and *place* by performing an integer division of $v.positions[index]$ by $nright_i$:
$number \ = \ v.positions[index] \ mod \ nright_i$ and
$place \ = \ v.positions[index] \ div \ nright_i$.
An integer division by $nright_i$ boils down to fetching the elements that are spaced $nright_i$ apart in the extended vector.
2. Filling *infozout*: When w is a probability vector, the array *w.positions* is initialized with the positions of the states of S. This means that we can perform a traversal of the l-slice of *w.positions*, to construct a set *infozout* similar to *infozin*, but which takes into account the positions of the vector w (these may be different from those of v if v is an intermediate vector). Thus, the fields *number* and *place* respectively represent for each element of w, the number of the z_{out} and the place of the element in the z_{out}. When w is an intermediate vector, we have no information on w so there is no *infozout*.
3. Sorting *infozin* and *infozout*: We need to perform a sort[4] on the sets of triplets according to *number* so that all the elements of the same z_{in} (or z_{out}) end up in adjacent positions.
4. Treatment of a z_{in}:

a. Extraction of the z_{in}: The elements of the same z_{in} are now placed consecutively and a single traversal of *infozin* allows us to get the z_{in} one after the other. When w is a probability vector, we extract the z_{in} only if at least one of the states of the corresponding z_{out} is reachable. This is obtained by a single traversal of *infozout*, because the elements of the same z_{out} are now placed consecutively. So we need not necessary to extract all the z_{in} as it is done in E-Sh.
b. Multiplication: When w is a probability vector, *infozout* informs us of which elements need to be multiplied, and we can finally perform the multiplication of the z_{in} by the corresponding

[3]To implement this structure, we used the Standard Template Library (STL) *containers*.
[4]The STL sort used is *introsort*, a variant of *quicksort* which offers a complexity of $O(N \log N)$ in the worst case.

columns of the matrix. In this case, we do not have to perform all the products as in E-Sh. On the other hand, when w is an intermediate vector, we do not know a priori the position of the nonzero elements, so we perform the sparse multiplication of z_{in} by the entire matrix, and we store all the nonzero elements as a (position, value) pair.

5. Storing the result: When w is a probability vector, we perform the multiplication for each reachable state and the information contained in *infozout* tells us where to store the value obtained ($w.vec[index]$). The storage therefore takes place as and when the computations are performed. On the other hand, when w is an intermediate vector, we must wait until the end of the computations with all of the z_{in}, and then sort the nonzero elements by increasing position order. A linear traversal of these elements is sufficient to fill the tables $w.positions$ and $w.vec$[5].

When all the z_{in} of a given l-slice have been treated, then the following l-slice is treated.

A pseudo-code of the FR-Sh can be found in [20].

Complexity of the algorithm: Notice first that we no longer use arrays of size $|\hat{S}|$. All the arrays are of size $|S|$ since we assume that there is no intermediate vector $\hat{\pi}$ such that $|pos_{\hat{\pi}}| > |S|$. As far as computation time is concerned, we reduce the number of multiplications to the order of $|S| \times \sum_{i=1}^{N} \frac{nz_i}{n_i}$ when w is a probability vector (where we assume that the number of nonzero elements per column is uniform). However, we introduce some supplementary costs, most notably, the cost of a sort which could reach $O(|S| log(|S|))$.
When the percentage of nonreachable states is high ($|S| \ll |\hat{S}|$), the improvement is significant. The computation and memory cost are somewhat higher than E-Sh when $|S| \approx |\hat{S}|$. We have therefore fulfilled our objective of not using any structure of size $|\hat{S}|$ while maintaining an efficient algorithm when $|S| \ll |\hat{S}|$.

4 Numerical results

Now that the algorithms have been presented, we shall study their performance and compare them with one another on the basis of both memory needs and execution time.

All numerical experiments were performed using the software package, PEPS ([11]), into which we implemanted the new algorithms. All execution times were measured to within a tenth of a second on a 531 MHz Pentium III PC with 128 MB of memory. Convergence was verified to an absolute precision of ten decimal places, i.e., the results have a tolerance of the order of 10^{-10}. In all experiments, the initial vectors were chosen to be equiprobable, and we used the *unpreconditioned power method*. Furthermore, since the new algorithm does not alter the speed of convergence of the methods, we only provide the total time needed during execution and not the number of iterations required for convergence.
The memory use is taken from the system during execution. It represents the totality of memory used by PEPS during its execution (i.e., during the solution of a model). This includes the data, memory structures reserved by the procedure, and also the process stack. The only parameters that change from one algorithm to the next are the memory structures reserved by the algorithms (probability vectors, intermediate array structures, and so on).

We present results obtained from two classical parallel systems chosen from the literature ([6, 19, 31]). The first model, called **mutex1**, performs resource sharing with the use of functions; the second, **mutex2**, represents the same model, but with functions replaced by synchronizing transitions. The results obtained from a queuing network model can be found in [20].
In both models, N distinct clients share P identical units of a common resource. A customer requests use of a resource at a rate λ and frees up resource at a rate μ. For the tests, we use the values $\lambda = 6$ and $\mu = 9$. The values of N and P are varied according to the experiment.

Mutex 1

For this first model, a function is used to represent the mechanism by which access to the units of resource is restricted. The semantics of this function is as follows: *access permission is granted if at least one unit of the resource is available*. Freeing up resource, as opposed to acquiring resource, occurs in an independent manner.

The SAN product state space for this model is of size 2^N. Notice that when $P = 1$, the reachable state space is of size $N + 1$, which is considerably smaller than the product state space, while when $P = N$ the reachable state space is the entire product state space. Other values of P give rise to intermediate cases.

The first results we give are for the case in which $N = 16$, and in varying P. In all cases, we have $|\hat{S}| = 65,536$, and, naturally $|S|$ changes with P.

[5]It is at this moment that a dynamic reallocation may be needed.

Model			E-Sh		PR-Sh		FR-Sh	
P	$\|S\|$	$\|S\|/\|\hat{S}\|$	Time sec.	Mem Kb	Time Sec	Mem Kb	Time Sec	Mem Kb
1	17	0.03%	27.2	4208	1	4476	3.5	2124
4	2517	3.8%	112	4208	11.5	4584	22.1	2304
6	14893	22.7%	178.2	4208	70	5116	93.1	3128
8	39203	59.8%	235.4	4208	206.6	6176	267.4	4748
10	58651	89.5%	292.7	4212	348.6	7016	478.9	6040
12	64839	98.9%	332	4212	419.6	7280	612.9	6448
16	65536	100%	27	4212	84.9	7832	377.8	6516

We notice, as expected, that PR-Sh is faster than FR-Sh. Both reduced algorithms are better than E-Sh in terms of execution time so long as the percentage of reachable states remains reasonable (less than 50%). During execution, we notice that algorithm FR-Sh permits a reduction in memory needs with respect to algorithm E-Sh as long as the percentage of reachable states is sufficiently high. Indeed, when we are working in reduced vector format, we use structures of the size of $|S|$, but an element of the vector needs additional information which is also stored. Thus, once we exceed 50% of reachable states, we should not hope to produce a gain in memory since a vector is stored with the help of two arrays of size $|S|$, which is greater than $|\hat{S}|$. The intermediate structures used in PR-Sh make its memory use greater in all cases than that of algorithm E-Sh.

We performed additional experiments on larger models to determine the limits and the possibilities of the algorithms developed. When $N = 24$, only algorithm FR-Sh is successful. With $P = 10$, we have $|\hat{S}| = 16,777,216$ and $|S| = 4,540,386$. The solution is obtained in 101,348 seconds and requires 307,768 Kb of memory.

The limits of this algorithm are however reached when the percentage of nonreachable states is high. When $P \geq 12$, the memory needs of algorithm FR-Sh become excessive and the solution cannot be computed.

Mutex 2

Let us now look at how this same system may be modeled without using functional transitions. One possibility is to introduce an additional automaton, a resource pool automaton, which counts the number of units of resource available at any moment. The action of a process in acquiring a resource could then be represented as a synchronizing event requiring the cooperation of the demanding process and the resource pool. A further synchronizing event would be needed for a process to return resource to the resource pool.

The SAN product state space for this model is of size $2^N * (P + 1)$, and the reachable state space size is identical to that of *Mutex1*.

This model allows us to test the efficiency of the new algorithms in the presence of synchronizations, and hence intermediate vectors w which do not satisfy $pos_w \subseteq S$.

Model			E-Sh		PR-Sh		FR-Sh	
P	$\|\hat{S}\|$	$\frac{\|S\|}{\|\hat{S}\|}$	Time sec.	Mem Kb	Time Sec	Mem Kb	Time Sec	Mem Kb
1	131072	0.01%	13.6	6480	6.1	7928	2.2	2368
4	327680	0.77%	85.4	12672	40.1	21784	11.1	2628
6	458752	3.25%	174.9	16800	101.3	37440	47.9	3628
8	589824	6.65%	296.5	20928	219.8	59860	150.3	5564
10	720896	8.14%	563.1	25060	449.8	84604	350.9	7116
12	851968	7.61%	1002.5	29188	774.6	112564	603.5	7628
16	1114112	5.88%	1861.5	37452	1350	179872	918.2	7780

The first thing to note in these tests is the real inefficiency of PR-Sh which is worse than FR-Sh both from the point of view of memory needs and from execution time. The difference in memory needs is obvious since PR-Sh uses intermediate data structures the size of $\hat{S}$. As for execution time, algorithm PR-Sh is not efficient in presence of intermediate vectors, because we don't know which elements of the result really need to be computed, and hence we perform $|\hat{S}|$ multiplications of vector slices by a column of the matrix. FR-Sh carries out tests in order to know if a z_{in} contains at least one nonzero element before evaluating the matrix, and only then performs the multiplication of the z_{in} by the matrix. These tests reduce the number of multiplications, which in turn occasions an improvement in time when compared to algorithm PR-Sh. It what follows, we no longer use algorithm PR-Sh in presence of intermediate vectors, but use FR-Sh indeed.

We observe that FR-Sh remains efficient with intermediate vectors. Furthermore, the use of synchronizations implies a large number of nonreachable states. Algorithm FR-Sh is therefore the most appropriate when the model contains synchronized transitions, both from the point of view of memory needs as well as from the point of view of execution time.

To verify that the memory required by the new algorithm FR-Sh really depends on $|S|$ and no longer on $|\hat{S}|$, we have drawn some curves that depict memory used by the two algorithms (Fig. 1). The memory used by FR-Sh is proportional to $|S|$, whereas this is not the case for algorithm E-Sh. For E-Sh we get a straight line in plotting memory use as a function of $|\hat{S}|$, which goes to show that the memory used is indeed proportional to $|\hat{S}|$. The origin of the straight line corresponds to the minimum memory needed for the execution of algorithm FR-Sh on this model (2400 Kb).

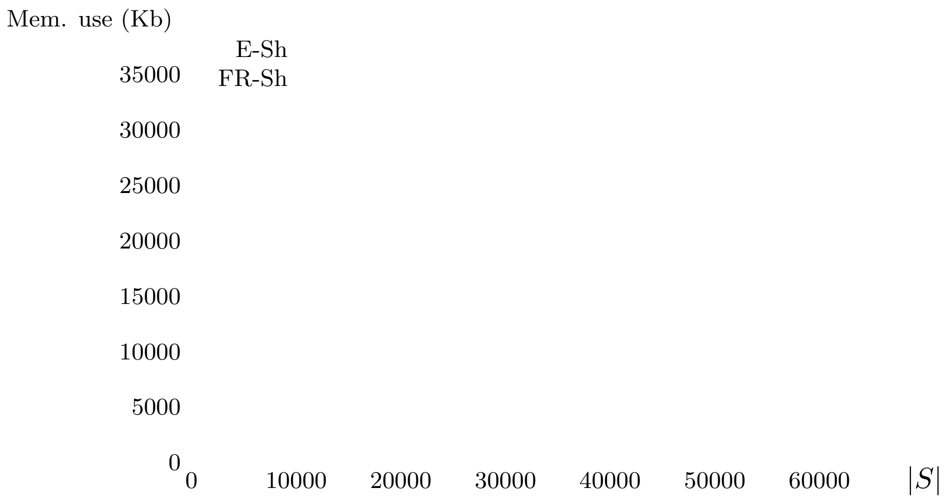

Figure 1: Memory use for the mutex2 model

5 Conclusions

In this paper we presented new algorithms based on the shuffle algorithm. The comparative tests that were performed allow us to reach the following conclusions:

- **Algorithm E-Sh, or the extended shuffle algorithm** is the most efficient when the percentage of reachable states is large. So, whenever more than half the states are reachable, this is the preferred algorithm.
- **Algorithm PR-Sh, or the partially reduced shuffle algorithm** takes advantage when the percentage of unreachable states is high to improve the performance of the vector-descriptor multiplication when there is no intermediate vectors. However, it uses intermediate structures that are of size $|\hat{S}|$, which limits its applicability on very large models. Furthermore, when there are some intermediate vectors, the performance is worse than FR-Sh, and we do not recommend its use in this case.
- **Algorithm FR-Sh, or the fully reduced shuffle algorithm** keeps all data structures at the reduced size. This permits us to handle very large models, models which the other algorithm cannot handle. We notice however a loss in computation time and in memory needs with respect to E-Sh when there are few unreachable states. This is the price we pay in order to be able to handle the largest models. A formal study of the complexity as a function of the percentage of reachable states and of the problem size is part of our future work.

We shall also compare our results with those obtained in generating the global matrix (in Harwell-Boeing (HB) format) and then performing a vector-matrix multiplication using standard sparse matrix multiplication. When it is possible to obtain the global matrix, there is no doubt that time-wise this algorithm performs best. It is from the memory point of view that it is limited. Therefore, the new algorithm allows us to solve models that can not be solved with the HBF algorithm. In the SANs formalism, the use of functions allows a decrease in the size of the product state space. The use of a generalized tensor algebra ([6, 31]) permits tensor operations on matrices to have functional characteristics. However, the cost of matrix evaluation is high and so we try to limit their number. Some techniques have been developed in order to decrease the number of matrix evaluations in the shuffle algorithm **E-Sh** [6, 19]. One possibility that we considered is a reordering of the automata. Some details are provided in [20] on the manner in which the newly presented algorithm need to be modified in order to handle the reordering of automata. Automata grouping ([6]) is another technique that may be used to decrease the number of function evaluations, but this technique is not presented here because the reduced storage of vectors does not cause any change to these procedures.

Finally, the new algorithms was only compared to the extended shuffle algorithm and the algorithm that generates the global matrix in HB format. A comparison with other classical algorithms, and notably algorithms that use Petri nets, [22, 32], were not included. Such a comparison will be performed in future work. We also propose to perform comparisons with our algorithms and matrix diagrams. Indeed, Ciardo in [27] presents a comparison between these two approaches, but he does not use the most recent algorithm developed for the Kronecker approach. His results show that matrix diagrams have a substantial advantage on the Act-Sh-JCB of [23], but it is now necessary to perform new experiments with FR-Sh in order to have comparisons with an algorithm which improves both the memory needs and the computation time when there is a lot of unreachable states.

These improvements to the shuffle algorithm will allow us to model and analyze even larger parallel and distributed systems, something that seems to us to be of primary importance in the evolution of such systems.

References

[1] W. J. Stewart, An introduction to numerical solution of Markov chains, Princeton University Press, New Jersey, 1994.

[2] S. Alouf, F. Huet, P. Nain, Forwarders vs. centralized server: an evaluation of two approaches for locating mobile agents, Performance Evaluation 49 (2002) 299–319.

[3] H. Garavel, H. Hermanns, On Combining Functional Verification and Performance Evaluation using CADP, Tech. rep., Rapport de recherche INRIA n. 4492, France (July 2002).
URL `http://www.inria.fr/rrrt/rr-4492.html`

[4] R. Stansifer, D. Marinescu, Petri net models of concurrent Ada programs, Microelectronics and Reliability 31 (4) (1991) 577–594.

[5] B. Plateau, De l'Evaluation du parallélisme et de la synchronisation, Ph.D. thesis, Université de Paris XII, Orsay (France) (1984).

[6] P. Fernandes, Méthodes Numériques pour la solution de systèmes Markoviens à grand espace d'états, Ph.D. thesis, Institut National Polytechnique de Grenoble, France (1998).

[7] E. Gelenbe, G. Pujolle, Introduction to Queueing Networks, second edition, John Wiley & Sons, 1998.

[8] M. Marsan, G. Balbo, G. Conte, S. Donatelli, G. Franceschinis, Modelling with generalized stochastic Petri nets, John Wiley & Sons, 1995.

[9] J. K. Muppala, G. Ciardo, K. S. Trivedi, Modeling Using Stochastic Reward Nets, in: MASCOTS'93 International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, San Diego, CA, IEEE Comp. Soc. Press., 1993, pp. 367–372.

[10] W. H. Sanders, J. F. Meyer, Reduced Base Model Construction Methods for Stochastic Activity Networks, IEEE Journal on Selected Areas in Communications 9 (1) (1991) 25–36.

[11] J.-M. Fourneau, K.-H. Lee, B. Plateau, PEPS: A package for solving complex Markov models of parallel systems, in: R. Puigjaner, D. Potier (Eds.), Proceedings of the Fourth International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Palma, Spain, 1988, pp. 291–305.

[12] G. Ciardo, A. Miner, Efficient reachability set generation and storage using decision diagram, in: In proc. 20th int. Conf. Application and Theory of Petri Nets, LNCS 1639, Springer, 1999.

[13] Y. Saad, Iterative Methods for Sparse Linear Systems, PWS Publishing Company, Boston, 1996.

[14] W. Stewart, MARCA: Markov chain analyzer, a software package for Markov modelling, in: W. Stewart (Ed.), Numerical Solution of Markov Chains, Marcel Dekker, 1991.

[15] D. Deavours, W. Sanders, "On-The-Fly" Solution Techniques for Stochastic Petri Nets and Extensions, in: Proceedings of the Seventh International Workshop on Petri Nets and Performance Models, IEEE Computer Society, Saint Malo, France, 1997, pp. 132–141.

[16] D. Deavours, W. Sanders, An Efficient Disk-Based Tool for Solving Large Markov Models, Performance Evaluation 33 (1) (1998) 67–84.
URL `citeseer.nj.nec.com/deavours98efficient.html`

[17] S. Donatelli, Superposed Stochastic Automata: a class of stochastic Petri nets with parallel solution and distributed state space, Performance Evaluation 18 (1) (1993) 21–36.

[18] S. Donatelli, Superposed Generalized Stochastic Petri nets: definition and efficient solution, in: Valette, R. (Ed.), Lecture Notes in Computer Science; Application and Theory of Petri Nets 1994, Proceedings 15th International Conference, Zaragoza, Spain, Vol. 815, Springer-Verlag, 1994, pp. 258–277.

[19] P. Fernandes, B. Plateau, W. Stewart, Efficient Descriptor-Vector Multiplications in Stochastic Automata Networks, JACM 45 (3) (1998) 381–414.

[20] A. Benoit, B. Plateau, W. Stewart, Memory Efficient Iterative Methods for Stochastic Automata Networks, Tech. rep., Rapport de recherche INRIA n. 4259, France (Sept. 2001).
URL `http://www.inria.fr/rrrt/rr-4259.html`

[21] P. Kemper, Reachability analysis based on structured representations, in: W. R. J. Billington (Ed.), Proc. 17th int. conf. Application and Theory of Petri Nets, Springer LNCS 1091, 1996, pp. 269–288.

[22] P. Kemper, Numerical Analysis of superposed GSPNs, IEEE Trans. on Software Engineering 22 (9).

[23] P. Buchholz, G. Ciardo, S. Donatelli, P. Kemper, Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models, INFORMS Journal on Computing 12 (3) (2000) 203–222.

[24] J.-P. Katoen, M. Kwiatkowska, G. Norman, D. Parker, Faster and Symbolic CTMC Model Checking, Lecture Notes in Computer Science 2165.
URL `citeseer.nj.nec.com/479458.html`

[25] M. Bozga, O. Maler, On the Representation of Probabilities over Structured Domains, in: N. Halbwachs, D. Peled (Eds.), Proc. CAV'99, Vol. 1633 of LNCS, Springer, 1999, pp. 261–273.
URL `http://www-verimag.imag.fr/ maler/Papers/pdg.ps.gz`

[26] H. Hermanns, J. Meyer-Kayser, M. Siegle, Multi Terminal Binary Decision Diagram to Represent and Analyse Continuous Time Markov Chains, in: Proc. 3rd int. workshop on the numerical solution of Markov chains, Prensas Universitarias de Zaragossa, Zaragossa, Spain, 2000.

[27] G. Ciardo, A. Miner, A data structure for the efficient Kronecker solution of GSPNs, in: In proc. 8th Workshop on Petri Nets and Performance models, IEEE CS Press, Zaragossa, 1999.

[28] A. Miner, Efficient solution of GSPNs using Canonical Matrix Diagrams, in: Proc. PNPM'01, 9th International Workshop on Petri Nets and Performance Models, Aachen, Germany, 2001, pp. 101–110.

[29] J. Granata, M. Conner, R. Tolimeri, Recursive fast algorithm and the role of the tensor product, IEEE Transactions on Signal Processing 40 (12) (1992) 2921–2930.

[30] C. Tadonki, B. Philippe, Parallel Multiplication of a Vector by a Kronecker product of Matrices, Journal of Parallel and Distributed Computing and Practices 2 (4).

[31] A. Benoit, P. Fernandes, B. Plateau, W. Stewart, On the Benefits of Using Functional Transitions in Kronecker Modelling, Submitted to Performance Evaluation .

[32] P. Buchhloz, Hierarchical structuring of Superposed GSPNs, in: Proc. 7th Int. Workshop on Petri Nets and Performance Models PNPM'97, Saint Malo, France. IEEE CS Press, 1997, pp. 81–90.