INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Memory Efficient Iterative Methods for Stochastic Automata Networks

Anne Benoit — Brigitte Plateau — William J. Stewart

N° 4259

Septembre 2001

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4259--FR+ENG

INRIA
RHÔNE-ALPES

Memory Efficient Iterative Methods for Stochastic Automata Networks

Anne Benoit *, Brigitte Plateau * , William J. Stewart †

Thème 1 — Réseaux et systèmes
Projet Apache

Rapport de recherche n° 4259 — Septembre 2001 — 55 pages

Abstract: We present new algorithms 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 tensorial structure implies to work on a product space. Inside this product space, the reachable state space can be much smaller. For these cases, we propose two improvements of the standard numerical algorithm (based on tensor products), called "shuffle algorithm", that take as input-output only data structures of the size of the reachable state space. One of the improvements allows a gain on the computation time, and the other one on the memory requirements. With those contributions, the numerical algorithms based on tensor products can deal with larger models.

Key-words: large, sparse Markov chains, stochastic automata networks, generalized tensor algebra, vector–descriptor multiplication, shuffle algorithm.

* Laboratoire Informatique et Distribution, ENSIMAG - antenne de Montbonnot, ZIRST 51, avenue Jean Kuntzmann - 38330 Montbonnot St Martin - France. Research supported by the (CNRS – INRIA – INPG – UJF) joint project *Apache*.

† Department of Computer Science, North Carolina State University , Raleigh, N.C. 27695-8206, USA. Research supported in part by NSF (CCR-9731856).

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 Montbonnot-St-Martin (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Méthodes itératives de résolution des réseaux d'automates stochastiques exploitant le creux du modèle

Résumé : Nous présentons de nouveaux algorithmes pour calculer la solution de grands modèles de Markov structurés dont le générateur peut être exprimé à l'aide d'algèbre tensorielle généralisée. Nous nous plaçons dans le cadre des réseaux d'automates stochastiques. La structure tensorielle implique de travailler sur un espace produit. A l'intérieur de cet espace produit, l'espace des états atteignables peut être très réduit. Pour ces cas, nous proposons deux améliorations de l'algorithme numérique (basé sur les produits tensoriels) standard, appelé "algorithme du shuffle", qui ne prennent en entrée-sortie que des structures de données de la taille de l'espace des états atteignables. L'une des améliorations gagne sur le temps de calcul et l'autre sur la mémoire utilisée. Avec ces apports, l'algorithmique numérique basée sur les produits tensoriels peut traiter numériquement des modèles encore plus grands.

Mots-clés : creux, chaînes de Markov, réseaux d'automates stochastiques, algèbre tensorielle généralisée, multiplication vecteur-descripteur, algorithme du shuffle.

1 Introduction

Continuous time Markov chains (CTMC) facilitate the performance analysis of dynamic systems in many areas of application [16]. 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. In this paper, our concern is with the computation of the stationary probability vector $\pi \in R^{|T|}$, 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 T is the set of states of the CTMC, called the reachable state space (RSS). 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 a software tool 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 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. Furthermore, matrix iterative techniques often need to compute the product of a probability vector and the generator Q many many times. Different possibilities can be found to store the generator and the probability vector in order to form these products efficiently. We shall consider some solutions proposed in the literature, and the advantages and drawbacks of each.

A first approach consists of using decision diagrams in order to represent the Markov chain [3, 4, 10]. This tree representation of the generator Q and of the probability vectors permits quick access to all of their elements, and the solution time is often satisfactory, even for large models. However, this representation of the generator requires a large amount of memory. All the nonzero elements of the matrix must be stored, as well as the tree structure that describes the path needed to find the elements.

Another approach is used in the software package Marca [17]. The idea here is to store the matrix in a row-wise compact sparse format: only 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. Efficient algorithms are available with which to compute a vector-matrix product when the matrix is stored in this fashion. However, for very large models, it is frequently the case that the matrix is too large to be held in memory.

In order to keep memory requirements manageable, Stochastic Automata Networks (SANs) were introduced, [13, 7]. These allow Markov chains models to be described in a memory efficient manner because their storage is based on a tensor formalism. However, the use of independant components connected via synchronizations and functions may produce a representation with many unreachable states. The set of all the states that can be

enumerated by the tensor formalism is called the product state space (PSS), but some of these states may not be valid in the sense that they never occur. For this reason, they are called "unreachable states". Another formalism based on Stochastic Petri Nets allows us to obtain a similar tensor formalism, as shown by Donatelli in [5, 6].

When the reachable state space (RSS) is almost equal to the product state space (PSS), the gain in memory obtained with the use of the tensor formalism can be enormous compared to the two alternative approaches described above. For example, if a model consists of K componants of size n_i, for $i = 1, \ldots, K$, which are full, then the space needed to store the generator with the previous methods is of the order of $(\prod_{i=1}^{K} n_i)^2$. The use of a tensor formalism reduces this cost to $\sum_{i=1}^{K} n_i^2$. However, when there are many unreachable states, the difference with the other approaches is reduced. Also, if the infinitesimal generator is very sparse, the gain may be less substantial. Unfortunately, the gains in memory are often accompanied by an increase in the costs of computing numerical solutions. The greatest obstacle to obtaining solutions for Markov chains stored in a tensor format is the inefficiency of the basic operations, and notably the multiplication of the generator by a vector [7, 8]. Once an efficient version of this operation is available, the application of iterative methods, whether simple or sophisticated, is immediate. When the generator Q is available in tensor form, it is usually called the SAN *descriptor*. Several approaches are possible for computing a vector-descriptor product. The first and perhaps best-known, is the shuffle algorithm [7, 8, 13], which computes the multiplication but never needs the matrix explicitly. However, as has been shown previously, this algorithm needs to use vectors the size of the product state space. Moreover, computations are carried out for all the elements of the vector, even those elements corresponding to unreachable states. In some models, the reachable state space is small compared to the product state space and the probability vector can therefore have many zero elements, since only states corresponding to reachable states have nonzero probability. 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. On the other hand, good performance is obtained with this algorithm when the number of unreachable states is small.

In contrast, if there are many unreachable states, the approaches first described, based on storing all nonzero elements of the generator, perform better because they do not carry out meaningless computations. However, due to the large memory requirements of these approaches, 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. Some methods have already been developed:

1. A first possibility for SANs consists of aggregating the components [9]. In this way, some unreachable states disappear. However, to be sure that all unreachable states are eliminated, it is necessary to aggregate all components and this becomes similar to the use of traditional methods which do not use the tensor formalism of the Markov system. This resurrects the memory limitation problem which we seek to remove. We note in passing that this approach of aggregation can also be adapted for Petri Nets.

2. Another approach consists of introducing a hierarchical structure containing only reachable states and the development of efficient algorithms to exploit this structure. Buchholz presents such results for Petri Nets in [1]. In this case however, the tensor formalism is modified and made more complex. As for the previous approach, these results can be extended to SANs.

3. A new approach has been proposed in [11, 12, 2]. It consists of first computing the reachable state space, and then solving the model by using iteration vectors which contains entries only for these states. Some additional computations are then needed at each iteration in order to obtain the indices of reachable states. Moreover, at each iteration the rows of the generator Q are formed (albeit in sparse format) so that the multiplication is performed for reachable states only. In this approach, the tensor formalism is not exploited as in the shuffle algorithm, because it is necessary to compute the elements of the generator explicitly. Nevertheless, the use of vectors of size RSS permits a reduction in memory used, and some needless computations are avoided.

Our goal is to conceive a new algorithm based on the shuffle algorithm, which exploits the tensor formalism without modifying it, and which never explicitly computes the elements of the generator. We wish however to take reachable states into account in such a way that performance is satisfactory even when the percentage of these reachable states is small. Obviously, in the new algorithms we seek to develop, we would like to use iteration vectors with entries corresponding to reachable states only. This implies that only necessary computations will then be performed.

In the next section we present the shuffle algorithm [13, 7] for the multiplication of a vector and a SAN descriptor. This algorithm exploits the structure of the SAN but it also uses probability vectors that we shall call *extended*. In other words, these vectors are the same size as the product state space. We denote this algorithm **E**, for extended. This algorithm can be improved both in the amount of execution time needed to compute solutions as well as in memory requirements, by taking advantage of the fact that the probability vectors of size PSS are very sparse: only reachable states have non-zero probability.

Two new algorithms are presented. The first exploits a knowledge of which states are reachable to improve the performance of the vector–descriptor multiplication. All the *probability* vectors are stored in an array of size RSS. We shall call vectors of this size *reduced* vectors. However, to obtain good performance at the computation-time level, some work vectors of size PSS (extended vectors) are also used. We refer to this as *partially reduced* and denote the corresponding algorithm **PR**. Unfortunately, the saving in memory turns out to be somewhat insignificant (Section 3). A second algorithm concentrates on the amount of memory used, and allows us to handle even more complex models. In this algorithm, all the intermediate data structures are stored in reduced format. We refer to this as *fully reduced* and denote the corresponding algorithm **FR**. Unfortunately, the result of this decreased need of memory translates into diminished performance for computation time (Section 4).

In the SANs formalism, the use of functions allows a diminution in the size of the state space. The use of a generalized tensor algebra ([7, 8, 15]) permits tensor operations on matrices to have functional characteristics. However, the cost of matrix evaluation is high and so we try to limit the number of evaluations. Some techniques have been developed in order to decrease the number of matrix evaluations in the shuffle algorithm **E** [7, 8]. One possibility that we consider is a reordering of the automata. In Section 5, we shall provide details on the manner in which the newly presented algorithms need to be modified in order to handle the reordering of automata. Automata grouping ([7]) 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 in the algorithms presented does not cause any change to these procedures.

A series of tests comparing the algorithms is presented in Section 6. These algorithms were incorporated into the software package PEPS [14] and tested by means of this package.

2 The Shuffle Algorithm

Equation (1) below exploits the tensor structure of a SAN descriptor to form the product of a vector v by a descriptor Q [7, 8].

$$vQ = v \sum_{j=1}^{(N+2E)} \left[\bigotimes_{i=1}^{N}{}_{g} Q_j^{(i)} \right] = \sum_{j=1}^{(N+2E)} \left[v \bigotimes_{i=1}^{N}{}_{g} Q_j^{(i)} \right] \tag{1}$$

Here N is the number of automata in the network and E is the number of synchronizing events. Notice that the tensor products are *generalized tensor products*. The basic operation of interest to us is therefore

$$v \bigotimes_{i=1}^{N}{}_{g} Q^{(i)}$$

where the indices j have been omitted from the matrices $Q_j^{(i)}$ in order to simplify the notation. This term is composed of a sequence of N matrices denoted $Q^{(i)}$ with $i \in [1 \ldots N]$, each associated with an automaton $\mathcal{A}^{(i)}$. We begin by introducing some definitions concerning finite sequences of matrices[1].

✍ **Let**

n_i be the order of the i^{th} matrix in a sequence;

[1] In the following, *finite sequences of matrices* will be referred to simply as matrix sequences since only finite sequences are considered in this paper.

$nleft_i$ be 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: $nleft_1 = 1$;

$nright_i$ be 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: $nright_N = 1$;

$\bar{n}_i$ be the product of the order of all matrices except the i^{th} in a sequence, i.e., $\prod_{k=1,k\neq i}^{N} n_k$ ($\bar{n}_i = nleft_i\ nright_i$);

First we present the basic algorithm for the multiplication of a vector by a tensor product. This algorithm is used when the tensor products do not have functional elements (classic tensor products). Then we shall consider the constraints imposed when the tensor product does contain functional elements (generalized tensor products).

2.1 The Non-functional case

The simplest case is the multiplication of a vector by a tensor product when this tensor product does not contain functional elements. In this case we must compute

$$v \bigotimes_{i=1}^{N} Q^{(i)}.$$

According to the decomposition property of tensor products [7], every tensor product of N matrices is equivalent to the product of N normal factors. In using this property for the term $\bigotimes_{i=1}^{N} Q^{(i)}$, we have

$$\begin{aligned} Q^{(1)} \otimes Q^{(2)} \otimes \cdots \otimes Q^{(N-1)} \otimes Q^{(N)} = \\ & Q^{(1)} \otimes I_{nright_1} \\ & \times I_{nleft_2} \otimes Q^{(2)} \otimes I_{nright_2} \\ & \times \cdots \\ & \times I_{nleft_{N-1}} \otimes Q^{(N-1)} \otimes I_{nright_{N-1}} \\ & \times I_{nleft_N} \otimes Q^{(N)} \end{aligned}$$

To compute the multiplication of a vector v by the term $\bigotimes_{i=1}^{N} Q^{(i)}$ it is necessary and sufficient to know how to multiply a vector by a normal factor. The vector v must be multiplied by the first normal factor, the result is multiplied by the second normal factor and so on until the last of the normal factors has been multiplied. This is possible thanks to the associativity property of the (usual) multiplication of matrices. Furthermore, the property of commutativity between normal factors allows the multiplication of the normal factors in any desired order.

Multiplication by a Normal Factor

We are interested in performing the multiplication of a row-vector v and a normal factor, i:

$$v \times I_{nleft_i} \otimes Q^{(i)} \otimes I_{nright_i}$$

Because of the associativity of the tensor product, this may be rewritten as

$$v \times I_{nleft_i} \otimes R$$

with

$$R = Q^{(i)} \otimes I_{nright_i}$$

The matrix $I_{nleft_i} \otimes R$ is a block diagonal matrix in which the blocks are simply the matrix R. We can treat the different blocks in an independent manner. 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 $p_1, \ldots, p_{nleft_i}$, as shown in Figure 1.

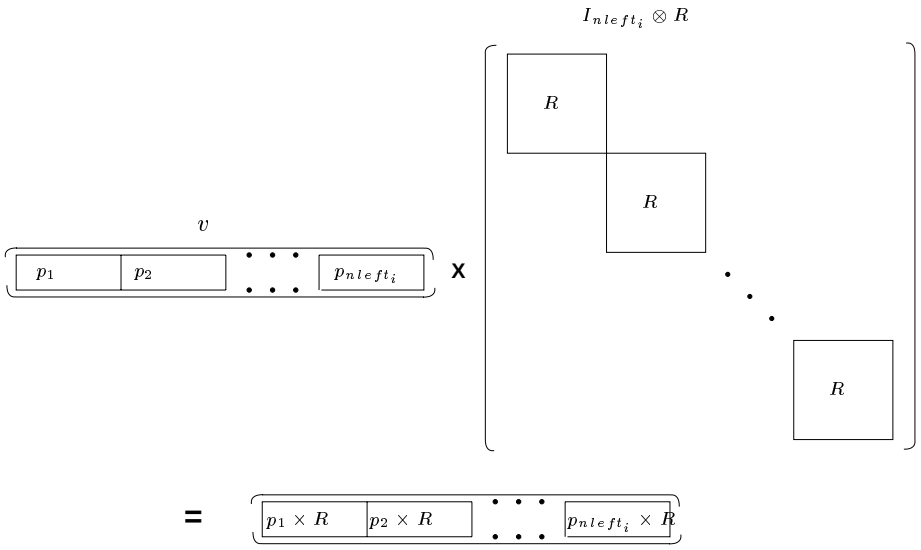


Figure 1: Multiplication $v \times I_{nleft_i} \otimes R$

Thus, we loop over the l-slices, and at each iteration we consider only a part of the vector to multiply. Figure 2 illustrates how the vector is divided into $nleft_i$ l-slices, each

of size $nright_i \times n_i$. We now consider the details involved in multiplying an l-slice. The computation is carried out by looping over the $nleft_i$ l-slices. (There is only one in the case of the first normal factor.)

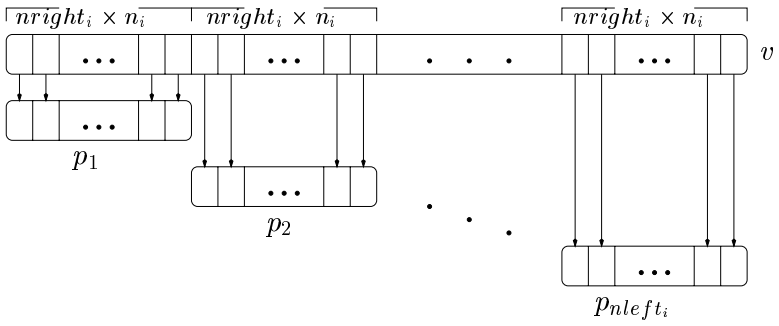


Figure 2: Vector separation into l-slices

Detail of an l-slice

We now need to describe the multiplication of an l-slice p_l with a block of the matrix, (the matrix R defined previously):

$$p_l \times R = p_l \times Q^{(i)} \otimes I_{nright_i}$$

Notice the structure of the matrix $Q^{(i)} \otimes I_{nright_i}$:

$$Q^{(i)} \otimes I_{nright_i} = \begin{pmatrix} q^{(i)}_{1,1} I_{nright_i} & q^{(i)}_{1,2} I_{nright_i} & \cdots & q^{(i)}_{1,n_i} I_{nright_i} \\ q^{(i)}_{2,1} I_{nright_i} & q^{(i)}_{2,2} I_{nright_i} & \cdots & q^{(i)}_{2,n_i} I_{nright_i} \\ \vdots & \vdots & \ddots & \vdots \\ q^{(i)}_{n_i,1} I_{nright_i} & q^{(i)}_{n_i,2} I_{nright_i} & \cdots & q^{(i)}_{n_i,n_i} I_{nright_i} \end{pmatrix}$$

Each matrix $q^{(i)}_{j,k} I_{nright_i}$ is a diagonal matrix of size $nright_i$ in which all the diagonal elements are equal to $q^{(i)}_{j,k}$. We wish to compute the resulting l-slice, $r_l = p_l \times Q^{(i)} \otimes I_{nright_i}$. The computation of an element of the resulting vector corresponds to the multiplication of p_l by a column of the matrix $Q^{(i)} \otimes I_{nright_i}$. However, each column of this matrix is composed of elements of a single column of the matrix $Q^{(i)}$; the other elements are zero. 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 column k of the matrix $Q^{(i)}$ is denoted by $q_{*,k}$.

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. We therefore proceed by jumps within the vector. Figure 3 represents the process of extracting the z_{in}. The k^{th} z_{in} is represented by z_{in} n° k, for $k = 0, \ldots, nright_i - 1$.

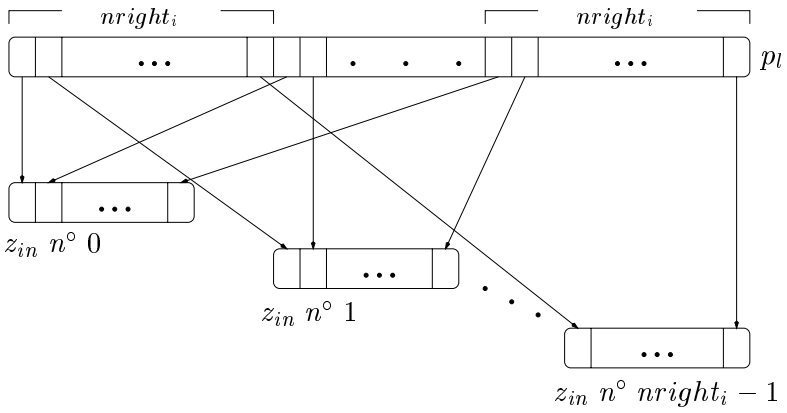


Figure 3: Dividing p_l into z_{in}

Once a z_{in} has been obtained, we can compute an element of the result by multiplying the z_{in} by a column $q_{*,k}$. 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.

Figure 4 illustrates the multiplication for some elements of the resulting vector. The elements computed constitute z_{out} n° 0. Indeed, we carry out the multiplication of z_{in} n° 0 by all the columns of the matrix $Q^{(i)}$ to obtain, successively, the different elements.

To summarize then, how an l-slice is handled (in other words, how we compute $p_l \times Q^{(i)} \otimes I_{nright_i}$), we proceed by carrying out a loop to successively extract $nright_i$ z_{in}. For each of these z_{in}, we carry out the multiplication $z_{out} = z_{in} \times Q^{(i)}$, which gives us pieces of the resulting vector; we then store these pieces (slices) z_{out} into the resulting vector. Since

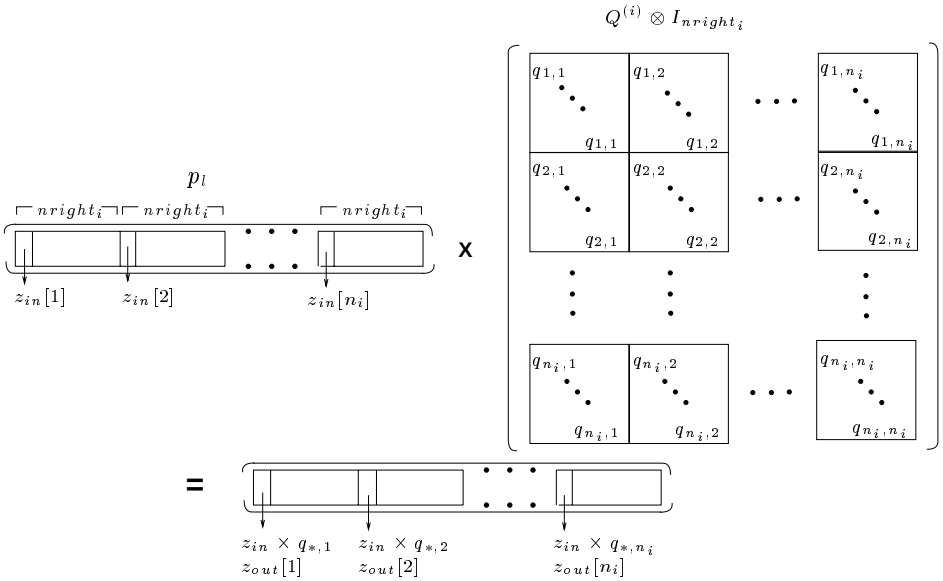


Figure 4: Multiplication $p_l \times Q^{(i)} \otimes I_{nright_i}$: Derivation of z_{out} n° 0.

the positions in z_{out} are the same as those in the corresponding z_{in}, this storage is performed analogously to the extraction of the z_{in} (Figures 3 and 4).

Multiplication Algorithm

Algorithm 2.1 performs the multiplication of a vector v and a tensor product $\otimes_{i=1}^{N} Q^{(i)}$. We shall call this algorithm **E** *without functions* (E for extended) , because the probability vectors are stored in extended format (size equal to PSS) and the algorithm does not handle functional dependencies. In this algorithm, the normal factors are handled in order from the first to the last. Nevertheless, according to the commutativity property of normal factors, any other order could have been used.

Algorithm 2.1

```
1  for i = 1,2,...,N                    // loop over all matrices
2  do base = 0;                         // base: beginning index of the l-slice in the vector
3     for l = 0,1,...,nleft_i - 1       // loop over all l-slices
4     do for r = 0,1,...,nright_i - 1   // l-slice detail: loop over all the z_in
5        do index = base + r;
         // index: index of the current element of z_in in the vector
6           for k = 0,1,...,n_i - 1 // extraction of z_in n° r (jumps of size nright_i)
7           do z_in[k] = v[index];
8              index = index + nright_i;
9           end do
10          multiply  z_out = z_in × Q^(i);        // multiplication
11          index = base + r; // index: the index of the current element of z_out
12          for k = 0,1,...,n_i - 1 // store result z_out (jumps of size nright_i)
13          do v[index] = z_out[k];
14             index = index + nright_i;
15          end do
16       end do
17       base = base + (nright_i × n_i);     // now onto the next l-slice (jumps in v)
18    end do
19 end do
```

Algorithm 2.1: **E** *without functions*

Complexity

The complexity of this algorithm for multiplying a vector and a classic tensor product may be obtained by observing the number of vector–matrix multiplications that are executed (line 10 of Algorithm 2.1). In each i loop of the algorithm, $nleft_i \times nright_i$ vector–matrix products are executed, with matrices of size n_i. If we assume that the matrices $Q^{(i)}$ are

full, the number of multiplications for each vector–matrix product is equal to $(n_i)^2$. From the definition of the sequences of matrices, $\bar{n}_i \times n_i = nleft_i \times nright_i \times n_i = \prod_{i=1}^{N} n_i$. The complexity of Algorithm 2.1 is therefore

$$\sum_{i=1}^{N} \left(\bar{n}_i \times (n_i)^2\right) = \prod_{i=1}^{N} n_i \times \sum_{i=1}^{N} n_i. \tag{2}$$

This should be compared to the multiplication $v \bigotimes_{i=1}^{N} Q^{(i)}$ which consists in first computing $Q = \bigotimes_{i=1}^{N} Q^{(i)}$, and then multiplying the result with the vector v. This has complexity of the order of $(\prod_{i=1}^{N} n_i)^2$.

If the matrices $Q^{(i)}$ are stored in a sparse compacted format, the number of multiplications for each vector–matrix product is generally much less than $(n_i)^2$. In this case, if nz_i is the number of nonzero elements in the matrix $Q^{(i)}$, the complexity of Algorithm 2.1 is:

$$\sum_{i=1}^{N} (\bar{n}_i \times nz_i) = \prod_{i=1}^{N} n_i \times \sum_{i=1}^{N} \frac{nz_i}{n_i}. \tag{3}$$

To get a more precise idea of the cost of this algorithm, let us establish a comparison of this complexity with the multiplication of a vector by a single matrix stored in sparse format. The matrix equivalent to a tensor product term possesses $\prod_{i=1}^{N} nz_i$ nonzero elements and therefore the complexity will be of this order. A comparison between formulas as different as these is difficult. Let us establish a limit for the sparsity of the matrices $Q^{(i)}$:

$$nz_i = n_i (N)^{\frac{1}{N-1}}$$

For this number of nonzero elements, the complexities of Algorithm 2.1 and the multiplication by a sparse matrix are equal. For values of nz_i that are less than this limit, the multiplication by a sparse matrix will be more efficient while for values greater than this limit, the advantage will lie with the proposed algorithm[2]. It should be remembered that these considerations are applicable to the comparison of a single (classic) tensor product term (without functional elements). Practical cases, with descriptors composed of a sum of generalized tensor products, are much more complex and only numerical experimentation can provide some concrete information concerning execution costs.

2.2 Handling Functional Dependencies

According to properties of generalized tensor products concerning their decomposition into normal factors, it is always possible to obtain an order σ for multiplying the normal factors of a term $\otimes_{i=1}^{N} Q^{(i)}(\dots)$ if and only if there are no cycles in the functional dependency graph. The existence of such cycles prevents the direct application of these decomposition properties. The way in which this situation must be handled is described in [7]. In this

[2] For these comparisons, the cost of generating a matrix from a tensor product is not taken into account.

section, we only describe the multiplication of generalized tensor products without functional dependency cycles.

The multiplication of a vector v by a tensor product $\bigotimes_{g\,i=1}^{N} Q^{(i)}(\mathcal{A}^{(i+1)}, \ldots, \mathcal{A}^{(N)})$ without cycles is carried out in a manner that is similar to the case without functional elements. Two modifications must be made to Algorithm 2.1 to accommodate functional elements. We must

- compute an order σ in which the normal factors must be multiplied;
- evaluate the functional elements of matrices before each is used in a multiplication.

2.2.1 Establishing the order of normal factors

A correct ordering may be obtained from the graph of functional dependencies [7, 8], by applying the properties of generalized tensor algebra concerning normal factor decomposition. This order is denoted by a permutation σ on the interval $[1 \ldots N]$, and is called its *decomposition order*. It represents the order in which the tensor product will be decomposed into normal factors.

✍ **Let**

σ be a permutation called σ on the interval $[1 \ldots N]$, which establishes a new order for a sequence of N matrices;

$\sigma(i)$ be the position of the matrix $Q^{(i)}$ in the order represented by the permutation σ;

σ_k be the index of the matrix in position k of the order represented by the permutation σ (if $\sigma_k = i$, $\sigma(i) = k$);

The product $v \times \bigotimes_{g\,i=1}^{N} Q^{(i)}(\ldots)$ must be handled as

$$v \times \prod_{i=1}^{N} \left[I_{nleft_{\sigma_i}} \underset{g}{\otimes} Q^{(\sigma_i)}(\ldots) \underset{g}{\otimes} I_{nright_{\sigma_i}} \right]$$

The modifications that must be made to Algorithm 2.1 to take this change of order of the normal factors into account is indicated in Algorithm 2.2. Notice that this modification requires the computation of a handling order (permutation σ) which is not included in the algorithm for the multiplication of normal factors.

For some tensor products, the *decomposition order* may not be unique. For example, two matrices in the sequence may be constant. In this case, either matrix may be treated before the other. This is due to the fact that the product of normal factors of two constant matrices is commutative. The general rule is that if two (or more) matrices do not have

Algorithm 2.2

1 **for** $i = \sigma_1, \sigma_2, \ldots, \sigma_N$
2 **do** $\cdots$
As for Algorithm 2.1
$\vdots$

Algorithm 2.2: Changing the order of the normal factors

direct or indirect functional dependencies between them, they may freely change position. This rule explains the absence of a defined order for the multiplication of the normal factors of a classical tensor product (the case without functions).

2.2.2 Evaluation of functional elements

Once the *decomposition order* has been computed, the second preoccupation for the multiplication of generalized tensor products is the evaluation of the functional elements of each matrix $Q^{(i)}(\ldots)$ before its multiplication by a slice z_{in} of the vector v (corresponding to line 10 of Algorithm 2.1). In order to simplify the notation, we shall assume that the order of decompositon σ is the order $1 \ldots n$.

With each execution of the multiplication of a slice z_{in} of vector v by the matrix $Q^{(i)}(\ldots)$, this matrix must be evaluated for the state of the automata that are arguments of $Q^{(i)}$. The indices l and r shall represent, respectively, subvectors of states of automata to the left and right of[3] automaton $\mathcal{A}^{(i)}$ (Algorithm 2.1). For the evaluation of functional elements, it is necessary to incorporate the computation of local states of automata that are arguments of $\mathcal{A}^{(i)}$. Since these arguments are, a priori, an arbitrary subset of the SAN, this leads us to compute the local state of each automata other than $\mathcal{A}^{(i)}$. Then it is necessary to perform the evaluation of $Q^{(i)}(\ldots)$ with these local states.

Let us first examine the computation of local states of automata that are to the left of automata $\mathcal{A}^{(i)}$. The loop index l of Algorithm 2.1 provides the current position in the state space $\prod_{j=1}^{i-1} S^{(j)}$, i.e., the subvectors of local states of type $(x^{(1)}, x^{(2)}, \ldots, x^{(i-1)})$. It is possible to think of a subvector of local states as a number in a variable base system, i.e., a number for which each of its digits has its own base. The local state of the automaton $\mathcal{A}^{(j)}$ $(x^{(j)})$ is a *number* which may vary in the interval $[0 \ldots n_j - 1]$. The sequence of these numbers allows us to obtain the position l in the state space $\prod_{j=1}^{i-1} S^{(j)}$ of a subvector

[3] We shall call *left* of a matrix of index i, each matrix of the same term which has indices j less than i ($j \in [1 \ldots i-1]$). Analogously, we call *right* of a matrix of index i, all matrices of the same term which have indices j greater than i ($j \in [i+1 \ldots N]$).

$(x^{(1)}, x^{(2)}, \ldots, x^{(i-1)})$ by means of the formula:

$$l = \sum_{j=1}^{i-1} \left(x^{(j)} \times \prod_{k=1}^{j-1} (n_k) \right)$$

In the same way, we can obtain the subvector $(x^{(1)}, x^{(2)}, \ldots, x^{(i-1)})$ from a given index l, by means of a sequence of integer divisions.

An algorithm which implements these divisions is very costly (there are as many integer divisions as the number of automata to the left of $\mathcal{A}^{(i)}$). A cheaper solution may be found. The first useful thing to note is that all combinations of local states are needed. Furthermore, the combinations will be taken exactly in lexicographical order, (The value of the index l varies from 0 to $nleft_i - 1$). The second interesting remark is that it is cheap to go from one subvector of local states to the next in lexicographical order. This may be performed by a simple incrementation of +1 of a number in a variable base.

As an example, let us consider the local states of three automata ($\mathcal{A}^{(1)}$, $\mathcal{A}^{(2)}$ and $\mathcal{A}^{(3)}$) with sizes $n_1 = 2$, $n_2 = 4$ and $n_3 = 3$. The subvectors of the local states in lexicographical order are:

position (l)	$x^{(1)}$ (l_1)	$x^{(2)}$ (l_2)	$x^{(3)}$ (l_3)	position (l)	$x^{(1)}$ (l_1)	$x^{(2)}$ (l_2)	$x^{(3)}$ (l_3)	position (l)	$x^{(1)}$ (l_1)	$x^{(2)}$ (l_2)	$x^{(3)}$ (l_3)
0	0	0	0	8	0	2	2	16	1	1	1
1	0	0	1	9	0	3	0	17	1	1	2
2	0	0	2	10	0	3	1	18	1	2	0
3	0	1	0	11	0	3	2	19	1	2	1
4	0	1	1	12	1	0	0	20	1	2	2
5	0	1	2	13	1	0	1	21	1	3	0
6	0	2	0	14	1	0	2	22	1	3	1
7	0	2	1	15	1	1	0	23	1	3	2

The change from $\{0, 1, 1\}$ (in position 4), to the next subvector in lexicographical order (in position 5) necessites incrementing the state of the last local state (from 1 to 2). The change of this new combination ($\{0, 1, 2\}$ - position 5) to the next (position 6) does not correspond to incrementing the last local state (which is already at its maximum value $n_3 - 1 = 2$), but rather to incrementing the second-to-last local state and setting the last to zero which then gives $\{0, 2, 0\}$. As a general rule, to proceed through the subvectors with local states of automata to the left of $Q^{(i)}$ we initialize

$$l_1 = 0 \quad l_2 = 0 \quad \cdots \quad l_{i-2} = 0 \quad l_{i-1} = 0$$

With each increment of l, corresponds an addition of +1 to the number in variable base, i.e., an increment of local state l_{i-1}, with carry over if necessary. The sequence of values

obtained is

$$\begin{array}{llllllllllll}
l_1 & = & 0 & l_2 & = & 0 & \cdots & l_{i-2} & = & 0 & l_{i-1} & = & 0 \\
l_1 & = & 0 & l_2 & = & 0 & \cdots & l_{i-2} & = & 0 & l_{i-1} & = & 1 \\
\vdots & & & \vdots & & & & \vdots & & & \vdots & & \\
l_1 & = & 0 & l_2 & = & 0 & \cdots & l_{i-2} & = & 0 & l_{i-1} & = & n_{i-1}-1 \\
l_1 & = & 0 & l_2 & = & 0 & \cdots & l_{i-2} & = & 1 & l_{i-1} & = & 0 \\
\vdots & & & \vdots & & & & \vdots & & & \vdots & & \\
l_1 & = & n_1-1 & l_2 & = & n_2-1 & \cdots & l_{i-2} & = & n_{i-2}-1 & l_{i-1} & = & n_{i-1}-1
\end{array}$$

This same reasoning may be applied to the computation of states corresponding to automata to the right of $\mathcal{A}^{(i)}$ (for index r of Algorithm 2.1).

The Multiplication Algorithm

Algorithm 2.3 implements the modifications necessary for handling terms with functional elements. This new algorithm is called **E** *with functions*. The six operations which have been added are

- line 1: the handling of normal factors in the decomposition order σ;
- line 3: the initialization of the subvector with the local states of automata to the left of the matrix $Q^{(i)}(\dots)$;
- line 5: the initialization of the subvector with the local states of automata to the right of the matrix $Q^{(i)}(\dots)$;
- line 12: the evaluation of the matrix $Q^{(i)}(\dots)$ with arguments computed (local states of automata);
- line 19: the incrementation of a subvector with the local states of automata to the right of the matrix $Q^{(i)}(\dots)$;
- line 22: the incrementation of a subvector with the local states of automata to the left of the matrix $Q^{(i)}(\dots)$;

It should be noted that the subvector of local states of automata to the left and to the right of the matrix $Q^{(i)}(\dots)$ are redundant with respect to the indices l and r, respectively. This redundancy has the goal of avoiding the computation of the subvectors at each incrementation of l and r, which is too expensive. From this point of view, the initializations (lines 3 and 5 of Algorithm 2.3) and the increments (lines 22 and 19) are equivalent to the initializations and increments on l and r executed by the loop in lines 4 and 6, respectively.

Algorithm 2.3

```
1   for i = σ(1), σ(2), ... , σ(N)
2   do  base = 0;
3       initialize    l_1 = 0, l_2 = 0, ... , l_{i-1} = 0;
4       for l = 0, 1, ... , nleft_i − 1          // loop over the l-slices
5       do  initialize    r_{i+1} = 0, r_{i+2} = 0, ... , r_N = 0;
6           for r = 0, 1, ... , nright_i − 1          // detail of an l-slice: loop over the z_in
7           do  index = base + r;
8               for k = 0, 1, ... , n_i − 1          // extract the z_in (jumps of size nright_i)
9               do  z_in[k] = v[index];
10                  index = index + nright_i;
11              end do
12              evaluate    Q^(i)(a^(1)_{l_1}, ... , a^(N)_{r_N});          // matrix evaluation
13              multiply    z_out = z_in × Q^(i);          // multiplication
14              index = base + r;
15              for k = 0, 1, ... , n_i − 1   // store the result z_out (jumps of size nright_i)
16              do  v[index] = z_out[k];
17                  index = index + nright_i;
18              end do
19              next    r_{i+1}, r_{i+2}, ... , r_N;
20          end do
21          base = base + (nright_i × n_i);   // now on to the next l-slice (jumps within v)
22          next    l_1, l_2, ... , l_{i-1};
23      end do
24  end do
```

Algorithm 2.3: **E** *with functions*

Complexity

Algorithm 2.3 does not change the size of the loops that are in Algorithm 2.1. The multiplication of a slice of the vector v by the matrices $Q^{(i)}(\dots)$ also remains identical. This allows us to assert that the complexity of Algorithm 2.3 remains of the same order as Algorithm 2.1. The only additional costs correspond to the six operations already alluded to. Working with the normal factors in the order designated by σ has little impact on the complexity, because it only adds an indirection to the access on i. The increment of the subvector with the local states of automata to the right of matrix $Q^{(i)}(\dots)$ and the evaluation of the matrix $Q^{(i)}$ represent a substantial cost, especially since these operations must be performed in the innermost loop (loop from 0 to $nright_i - 1$). Reordering techniques allow us to reduce the number of matrix evaluations by removing these evaluations from the innermost loop. These techniques are presented in Section 5. At this point we shall present new versions of the shuffle algorithm which exploit the sparse structure of *extended* probability vectors.

3 Improvements in Computation Time

The shuffle algorithm presented in the previous section used extended vectors, i.e., vectors the size of the product state space. However, all nonreachable states have probability zero in the initial vector and remain zero after each product with the infinitesimal generator. Buchholz et al. [2] present algorithms in which the vectors are of this size ($\#PSS$) but the shuffle algorithm can not be treated in this fashion. This is because it is possible that nonreachable states can have a nonzero probability in temporary, intermediate vectors. First we shall examine the vector–descriptor multiplication procedure in detail to detect those instants at which the vector being computed is in the RSS. We say that a vector is *in the RSS* when the probability of each nonreachable state in that vector is zero. We describe new algorithms which take advantage of a reduced data structure (a vector of length $\#RSS$) for storing probability vectors.

3.1 Multiplication details

The Markovian generator, Q, corresponding to the Markov chain associated with a stochastic automata network is defined by the following tensor formula (equivalent to Equation (1)):

$$Q = \bigoplus_{i=1}^{N}{}_g \, Q_l^{(i)} + \sum_{e \in \varepsilon} \left(\bigotimes_{i=1}^{N}{}_g \, Q_{e^+}^{(i)} + \bigotimes_{i=1}^{N}{}_g \, Q_{e^-}^{(i)} \right) \tag{4}$$

where N is the number of automata in the network and ε is the set of identifiers of synchronizing events. The tensor sum corresponds to the local events of the descriptor, and the tensor product corresponds to the synchronizing events. We shall study the two parts separately ($Q = Q_{local} + Q_{synchro}$).

We wish to perform the product $v \times Q = v \times Q_{local} + v \times Q_{synchro}$, v being a vector of the RSS. The multiplication of v by Q boils down to performing all the events (both local and synchronised) on all the automata beginning from states with nonzero probability (states in RSS). The resulting vector $v \times Q$ must therefore be in RSS, because by definition of the reachable states, it is not possible to reach a nonreachable state from a reachable state. On the other hand, it is possible that nonreachable states have nonzero probability during the course of the computation. We first describe the computation of the local part, then the computation of the synchronized part.

Study of the Local Part

From the definition of tensor sum,

$$v \times Q_{local} = v \times \bigoplus_{i=1}^{N}{}_g \, Q_l^{(i)} = \sum_{i=1}^{N} \left[v \times (I_{nleft_i} \underset{g}{\otimes} Q_l^{(i)} \underset{g}{\otimes} I_{nright_i}) \right]$$

For each automaton, we must compute $v \times (I_{nleft_i} \underset{g}{\otimes} Q_l^{(i)} \underset{g}{\otimes} I_{nright_i})$, which means that we must carry out the local events on automaton i. On leaving the reachable states (vector v), we remain in the set of reachable states. Indeed, either the local event has a constant rate, and all its transitions lead to a reachable state, or the event has a functional rate in which certain values are zero if they lead to a nonreachable state. Each $v \times (I_{nleft_i} \underset{g}{\otimes} Q_l^{(i)} \underset{g}{\otimes} I_{nright_i})$ therefore has zero values for all nonreachable elements. The sum of these terms $(v \times Q_{local})$ must also be in RSS, since it is the sum of vectors in RSS.

Study of the Synchronizing Part

For the synchronizing part

$$v \times Q_{synchro} = \sum_{e \in \varepsilon} \left(v \times \bigotimes_{i=1}^{N}{}_g \, Q_{e^+}^{(i)} + v \times \bigotimes_{i=1}^{N}{}_g \, Q_{e^-}^{(i)} \right)$$

For each event $e \in \varepsilon$, we need to compute

$$w = v \times \bigotimes_{i=1}^{N}{}_g \, Q_{e^+}^{(i)} + v \times \bigotimes_{i=1}^{N}{}_g \, Q_{e^-}^{(i)}$$

which means that we must carry out the synchronizing transitions of event e. By a similar reasoning as that expressed for the local part, when we have completely treated an event, the vector obtained w must be in RSS. However, during the computation of w, we are lead to perform multiplications by normal factors (the decomposition studied in the case of the shuffle algorithm). When we multiply v by the i^{th} normal factor it is possible to

leave the RSS, because the synchronizing event is performed only on automaton i (and not on the other automata which are synchronized with automaton i). However, during the final multiplication, we must return to RSS because we will by then have performed the synchronizing transition on all the effected automata.

Summary

During the multiplication by a normal factor, we may obtain a vector that is not in RSS. This can occur during any multiplication in the synchronizing part other than the multiplication by the last normal factor. In all other cases, and therefore before each addition, the vectors computed are in RSS. This property may be exploited in order to accelerate algorithms for multiplication by normal factors.

3.2 Multiplication with Reduced Vector Data Structures

In general $\#RSS \ll \#PSS$. However, in Algorithm 2.3 the computation of *all* the elements of the vector is performed as can be seen from line 16 where the elements of the vector are actually stored. Therefore, in the case of a model with many unreachable states, we carry out many needless computations since most of the elements of the vector will be zero. Indeed, we perform $\#PSS$ multiplications of a vector slice by a column of the matrix. We now introduce a new algorithm for the multiplication of normal factors that takes advantage of the fact that the vector obtained by the vector–description multiplication remains in RSS. The new probability vectors are of size $\#RSS$ and only elements corresponding to reachable states are stored. This allows us to reduce the number of multiplications from $\#PSS$ to $\#RSS$, by computing only the elements of the result that correspond to reachable states. However, when the intermediate vector is not in RSS, we are obliged to perform all the computations to find those elements of the resulting vector that are nonzero. The techniques used to handle intermediate vectors which are not is RSS will be described in detail later. For comparison purposes, the probability vectors used in the previous algorithms were all of size $\#PSS$.

Data Structures Used

The fact that we shall store only the values of the vector that correspond to reachable states means that somewhere we must keep track of the positions of these elements in the corresponding vector in PSS. We shall use two arrays of size $\#RSS$: the array *vec* contains the values of the reachable states and the array *positions* contains the positions of the reachable states in the corresponding vector in PSS. Figure 5 illustrates the structures used.

Initialization

When the vector is initialized, we need to know the set of reachable states RSS in order to be able to fill the arrays *vec* and *positions* corresponding to the initial vector. This may be

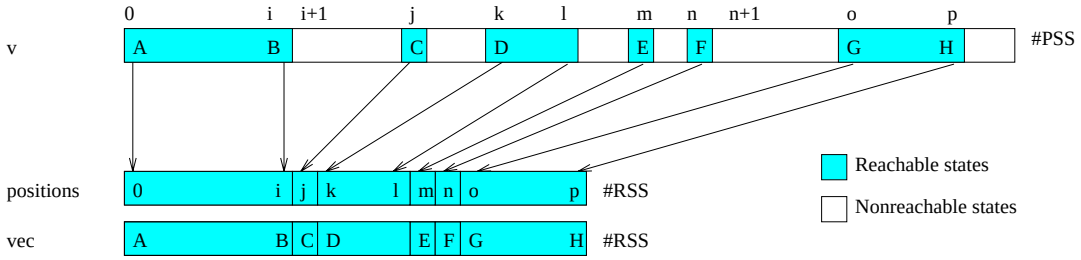


Figure 5: Illustration of data structures that take advantage of sparsity

done either by applying an algorithm that explores all reachable states ([11]), or by asking the user to provide an expression which represents the set of reachable states of the system[4]. According to the choice of an initial vector, (equiprobable vector, vector provided by the user, and others) we may complete the arrays with the necessary information.

Division into l-slices

We would like to apply the shuffle algorithm previously described. The idea of an l-slice remains the same because the format of a normal factor does not change: we wish to compute $v \times I_{nleft_i} \otimes Q^{(i)} \otimes I_{nright_i}$. The division of v into l-slices p_l may be carried out by inspecting the array *positions*, and by taking the elements that correspond to the l-slice. An l-slice is a set of consecutive elements and a single sweep over the vector allows us to determine the limits of l-slices. Figure 6 shows this division. We then work by looping over the l-slices, just the same as before.

Extracting the z_{in}

The principal difficulty in applying the old algorithm with the new reduced vector data structure lies in extracting the slices of the vector z_{in} from an l-slice of *vec* and *positions*. Indeed, the previous algorithms 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 somewhat similar method can however, be adopted by traversing the vector for each extraction of a z_{in}, during which the nonzero elements that belong to this z_{in} are found. At each loop on $nright_i$ (line 6 of Algorithm 2.3), we traverse the l-slice of the vector and store the elements corresponding to the z_{in}. In order to avoid the numerous vector traversals implicit in this approach (1 traversal per z_{in} must be extracted, that is $nright_i$ traversals

[4]This is what actually occurs in the current version of PEPS [14].

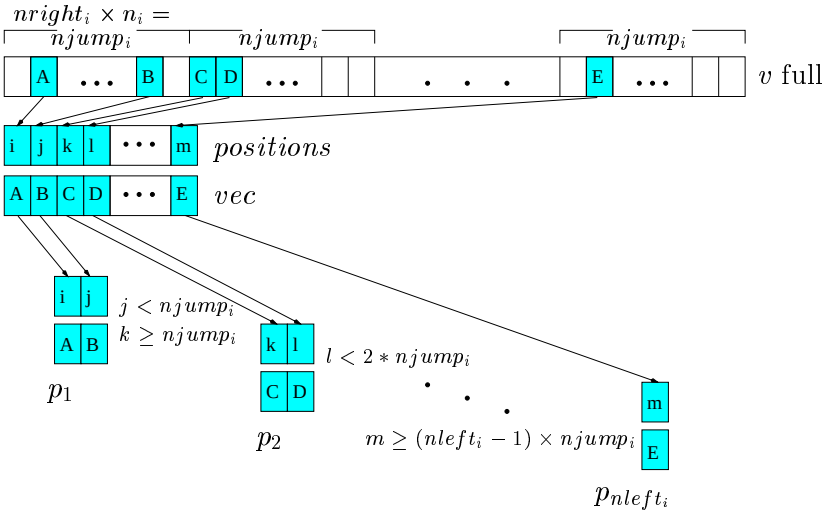


Figure 6: Separation of a reduced vector into l-slices, p_l

per l-slice), we may recuperate, in a single sweep, all the z_{in} for the l-slice being treated. The cost of extracting the z_{in} is therefore divided by $nright_i$.

We stock all the z_{in} corresponding to an l-slice p_l in an array. The z_{in}'s are similar to those used in the *extended* shuffle algorithm **E**; they are stored in a vector of length at most $\#PSS$. Thus, for each l-slice p_l (stored in reduced format) we build an extended vector which contains the sequence of z_{in}, in an appropriate order for this normal factor. Thus, each element of p_l belongs to a given z_{in}, and it is in a specific location in this z_{in}. The number of the z_{in} indicates the z_{in} to which an element belongs. The numbering goes from 0 to $nright_i - 1$, and corresponds to the numbering introduced in Section 2 (Algorithm **E**). 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. Therefore, $z_{in}[r]$ represents z_{in} n° r; it is an array of size n_i containing all the elements of z_{in} n° r. Thus $z_{in}[r][k]$ represents the element in position k in z_{in} n° r. z_{in} is an array of size $nright_i \times n_i$.

To extract the z_{in}, we perform an integer division on the values in the array *positions*, which gives, for j from 0 to $\#RSS$, the number of the z_{in} to which belongs the corresponding element ($positions[j]$ mod $nright_i$), as well as its place in z_{in} ($positions[j]$ 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. Figure 7 shows this technique for extracting the z_{in}.

Once we have the z_{in}, it remains to perform the multiplications $z_{in} \times q_{*,k}$ (recall the presentation of the shuffle algorithm of Figure 4) and then to store the results in the correct

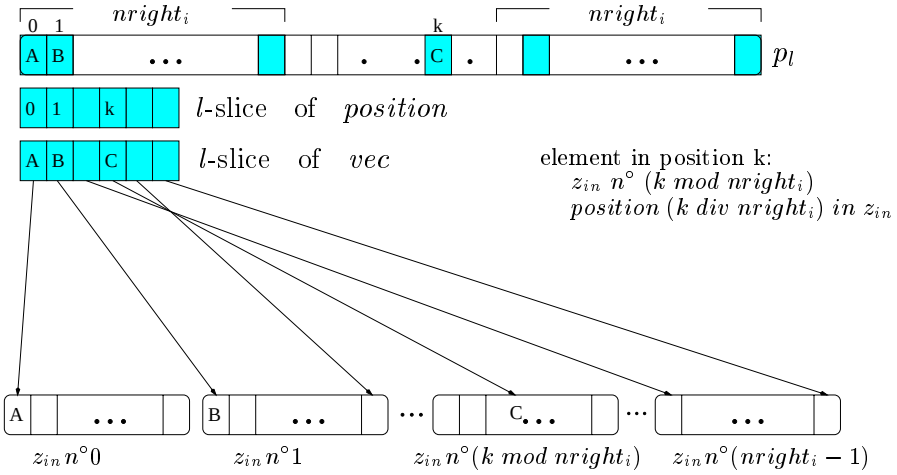


Figure 7: Extracting all the z_{in} for an l-slice of a reduced vector

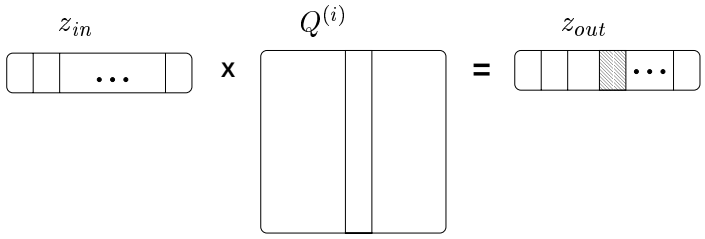
place in the vector w (reduced format: $w.vec$ and $w.positions$). It is also necessary to evaluate the matrices in cases when they are functional. We shall first consider the case in which we remain in RSS (local events only) and then we shall consider the modifications that must be made if we exit the RSS.

Case I: Remain in RSS

We now assume that we have all the z_{in} (for an l-slice). Therefore, we can directly access a given z_{in}. The fact that we remain in RSS lets us know which elements will be nonzero in the solution vector obtained: these are the elements corresponding to reachable states. Therefore we shall compute only the value of elements corresponding to these states. To do this, we traverse the resulting vector (whose values in the array *positions* correspond to reachable states) and we compute the probability for each reachable state, as described below. We saw previously, during the presentation of the shuffle algorithm (Figure 4) that the computation of an element of the resulting vector is performed by multiplying a given z_{in} by a column of the matrix $Q^{(i)}$.

To obtain the number of the z_{in} and the column of the matrix allowing the multiplication for a given element of the resulting vector, the schema is similar to that corresponding to the extraction of the z_{in}. Indeed, the elements corresponding to the multiplication by the same z_{in} are spaced $nright_i$ apart in the extended vector. If we wish to compute the value of the element in position k, we begin by doing an integer division of k by $nright_i$. The remainder

of the division gives us the z_{in} to use for the multiplication, and the quotient gives us the column of the matrix. The multiplication is carried out as illustrated in Figure 8.



The grey position corresponds to a reachable state in position k.
Its value is obtained by multiplying z_{in} $n°$ $(k \ mod \ nright_i)$
and the column of the matrix $(k \ div \ nright_i)$

Figure 8: Computation of the values in RSS of an l-slice of the resulting vector

Algorithm 3.1 performs the multiplication $w = v \times (I_{nleft_i} \otimes Q^{(i)} \otimes I_{nright_i})$. We shall refer to this algorithm as **PR**, for *partially reduced*, because the vectors v and w are stored in reduced format but the algorithm uses intermediate structures in extended format (the z_{in}). The algorithm corresponds to the case without functions in which it remains in RSS.

The **while** loops (lines 8 and 14 of Algorithm 3.1) serve to limit our study by l-slice. The l^{th} l-slice corresponds to elements of the vector that are found between positions $l \times njump_i$ and $((l+1) \times njump_i) - 1$. The variables j and $j2$ mark progress within v and within the resulting vector w respectively.

It should be noted that we do not perform the division of the position of an element by $nright_i$, but that we begin by subtracting $l \times njump_i$ from the position. Indeed, $l \times njump_i$ corresponds to the position of the beginning of the l-slice l. To obtain a quotient between 0 and $n_i - 1$, this subtraction is needed. Otherwise, the quotient would lie between $l \times n_i$ and $((l+1) \times n_i) - 1$. In theory, the subtraction is only needed for the computation of the quotient, but in practice, both computations are performed at the same time. This is why we also perform the subtraction for the computation of the remainder.

A problem arises in evaluating functional elements, because we can no longer proceed by traversing *positions* in a linear fashion. If we were to proceed like this it would be necessary to re-evaluate the functional matrices for each product of a column of $Q^{(i)}$ by a z_{in}. But the evaluation of functional elements is very expensive. We would prefer to proceed as we did in Algorithm 2.3 (**E** *with functions*), since this implements a less expensive solution. Indeed, in this algorithm, states are changed (as functional arguments) progressively by

Algorithm 3.1

```
1   j = 0; j2 = 0;
2   for l = 0, 1, ... , nleft_i − 1          // loop over the l-slices
3   do  for r = 0, 1, ... , nright_i − 1          // initialization of z_in
4       do  for k = 0, 1, ... , n_i − 1
5           do  z_in[r][k] = 0;
6           end do
7       end do
        // Fill in the z_in
8       while   (j < v.dimension) && (v.positions[j] < (l + 1) × njump_i)
9       do  re = (v.positions[j] − l × njump_i) mod nright_i;
10          qo = (v.positions[j] − l × njump_i) quo nright_i;
11          z_in[re][qo] = v.vec[j];
12          j = j + 1;
13      end do
        // Compute the result
14      while   (j2 < w.dimension) && (w.positions[j2] < (l + 1) × njump_i)
15      do  re = (w.positions[j2] − l × njump_i) mod nright_i;
16          qo = (w.positions[j2] − l × njump_i) quo nright_i;
17          w.vec[j2] = z_in[re] × Q^(i)[qo];   // Q^(i)[qo] represents column qo of matrix Q^(i)
18          j2 = j2 + 1;
19      end do
20  end do
```

Algorithm 3.1: **PR** *without functions* - Case I: Remain in RSS

increments. The change of state occurs when z_{in} is changed (the loop containing lines 6 to 20 of Algorithm 2.3). For a given z_{in}, only a single function evaluation is needed.

In the case of reduced probability vectors, it is necessary to proceed in a different way to that of Algorithm 3.1, which performs a linear traversal. We must now treat the z_{in} in the same order as in Algorithm 2.3. Algorithm 3.2 performs the multiplication with function evaluations in the innermost loop. We now provide details on the different phases involved.

The initialization of the z_{in} and their extraction occur in the same way as for the algorithm without function evaluation (Algorithm 3.1, lines 1 to 12). For the initialization functions we must add the subvectors with the local states of automata, and the same for the initialization of several arrays considered below. These arrays are used to store information concerning the computations that must be performed when a linear traversal is no longer possible. To decrease the amount of computation, we keep elements that are known to be nonzero in an array and perform the computations only for these elements. An algorithm that does not use arrays must perform $nright_i$ traversals of the l-slice. A single traversal of the l-slice of w provides all the elements that will be nonzero, but these elements will not be in the correct order. In the case of Algorithm 3.1, a single traversal provides us with the elements to be computed, one after the other. and there is therefore no neeed of arrays since elements are handled in a linear fashion. The traversal is performed by lines 13 through 20 of Algorithm 3.2. Intermediate information is stored in the following arrays:

- the array **used** specifies z_{out} which contain at least one reachable state. The z_{out} are numbered in the same way as the z_{in}: $z_{out}[r] = z_{in}[r] \times Q^{(i)}(\dots)$ with $r \in [0 \dots nright_i - 1]$. The array **used** is an array of booleans of size $nright_i$. Element $used[r]$ has the value *true* if and only if z_{out} n° r contains at least one element that corresponds to a reachable state. With each change of l-slice, all the values of this array are initialized to *false*.

- the array **index** is a counter to keep track of how many reachable states are present in each z_{out}. Thus, the z_{out} n° r contain $index[r]$ reachable states.

- 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} n° r. Element $useful[r][k]$ corresponds to the position in z_{out} n° 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} n° 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 $nright_i$ (line 22), and we perform the evaluation of the matrix only if this is necessary (line 24). 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*.

Algorithm 3.2

```
1   j = 0; j2 = 0;
2   initialize   l_1 = 1, l_2 = 1, ..., l_{i-1} = 1;
3   for l = 0, 1, ..., nleft_i - 1          // loop over the l-slices
4   do  initialization of z_in to 0
5       initialization of index to 0
6       initialization of used to false
        // Fill in the z_in
7       while   (j < v.dimension) && (v.positions[j] < (l + 1) × njump_i)
8       do  re = (v.positions[j] - l × njump_i) mod nright_i;
9           qo = (v.positions[j] - l × njump_i) quo nright_i;
10          z_in[re][qo] = v.vec[j];
11          j = j + 1;
12      end do
        // Intermediate information
13      while   (j2 < w.dimension) && (w.positions[j2] < (l + 1) × njump_i)
14      do  re = (w.positions[j2] - l × njump_i) mod nright_i;
15          qo = (w.positions[j2] - l × njump_i) quo nright_i;
16          used[re] = true;          // Which z_out must be computed
17          useful[re][index[re]] = qo;          // For a given z_out, the places used
18          place[re][index[re]] = j2;          // For a given z_out, the place in the vector
                                                // into which to store the result
19          index[re] = index[re] + 1; j2 = j2 + 1;
20      end do
21      initialize   r_{i+1} = 1, r_{i+2} = 1, ..., r_N = 1;
22      for r = 0, 1, ..., nright_i - 1          // Compute the result
23      do  if   (used[r])
24              evaluate   Q^(i)(a_{l_1}^(1), ..., a_{r_N}^(N));          // Evaluate only if necessary
25              for k = 0, 1, ..., index[r] - 1
                // Multiplication by the matrix column
26              do  w.vec[place[r][k]] = z_in[r] × Q^(i)[useful[r][k]];
27              end do
28          end if
29          next   r_{i+1}, r_{i+2}, ..., r_N;
30      end do
31      next   l_1, l_2, ..., l_{i-1};
32  end do
```

Algorithm 3.2: **PR** *with functions* - Case I: Remain in RSS

Finally, we compute each element corresponding to a reachable state of this z_{out} n° r, by carrying out a loop from 0 to $index[r]-1$ (line 25). 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. The computation (line 26) is similar to that performed by line 17 of Algorithm 3.1.

Case II: RSS is exceeded

During the multiplication of a vector v by a tensor product $\bigotimes_{g\ i=1}^{N} Q_e^{(i)}$ corresponding to the treatment of a synchronizing event e, the intermediate vectors computed are not necessarily in RSS. 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 must form the products $z_{in} \times Q_e^{(i)}$, and only keep the nonzero elements on exit. We must therefore update the array *positions* of the resulting vector when we fill the vector with nonzero elements.

In most cases, the number of states with a nonzero probability on exit is less than or equal to $\#RSS$. It is therefore possible to use the same size of vector. However, it sometimes happens (but rarely in our experience) that the number of states increases. In this case, it becomes necessary to dynamically reallocate memory in which to store the vector (the arrays *vec* and *positions* are not sufficiently large)[5].

The previous algorithms (3.1 and 3.2) read the *positions* of v (vector on entry) and of w (resulting vector) separately. Indeed, when we stay within RSS, the values of the *positions* of w are already initialized to the *positions* of the states in RSS. On the other hand, it is possible that the *positions* of v are different, notable if we have just handled a case which leaves the RSS. When we know that we leave the RSS, we compute the *positions* of w during the vector–matrix multiplications. We no longer use the known *positions* as in the previous case.

The principal difference with Algorithm 3.2 is that we do not perform the search for which elements to compute (lines 13 through 20). Thus, we no longer perform the multiplication of a z_{in} by the column of a matrix, but 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*.

Algorithm 3.3 performs the multiplication in the case when we leave the RSS and there are functions to be evaluated. Here we handle the z_{in} in the same order as in Algorithm 2.3; the evaluations are performed in the same manner. The multiplication of line 14 stores the nonzero elements in z_{out} during the computation $z_{in} \times Q^{(i)}$. After sorting these elements into increasing order (line 17), we can store the results in the solution vector (lines 18 to 22). We do not necessarily use all of array *vec*, which means that it is useful to include a

[5]We have not yet implemented dynamic reallocation in PEPS; instead we simply alert the user that the expanded method should be used. This happens very rarely and is the reason why we have not had need of dynamic reallocation for the moment.

position equal to -1 once all the nonzero elements have been stored. In this way we know what part of the array has been used.

Algorithm 3.3

```
1   j = 0; j2 = 0;
2   initialize   l_1 = 1, l_2 = 1, ... , l_{i-1} = 1;
3   for l = 0, 1, ... , nleft_i - 1          // loop over the l-slices
4   do  initialize the z_in to 0 and remove z_out;
        // Fill in the z_in
5       while   (j < v.dimension) && (v.positions[j] < (l + 1) × njump_i)
6       do  re = (v.positions[j] - l × njump_i) mod nright_i;
7           qo = (v.positions[j] - l × njump_i) quo nright_i;
8           z_in[re][qo] = v.vec[j];
9           j = j + 1;
10      end do
11      initialize   r_{i+1} = 1, r_{i+2} = 1, ... , r_N = 1;
12      for r = 1, 2, ... , nright_i          // Computation of the result
13      do  evaluate   Q^(i)(a^(1)_{l_1}, ... , a^(N)_{r_N});
14          multiply   z_out = z_in × Q^(i);
15          next   r_{i+1}, r_{i+2}, ... , r_N;
16      end do
17      Sort z_out by increasing position order;
18      while   ((pos, val) ∈ z_out)          // Store the result of the l-slice
19      do  w.positions[j2] = pos + l × njump_i;
20          w.vec[j2] = val;
21          j2 = j2 + 1;
22      end do
23      if   (j2 ≥ dimension)
24          Warn the user of array overflow, or else reallocate dynamically.
25      end if
26      next   l_1, l_2, ... , l_{i-1};
27  end do
    // If needed, place a marker to keep the last position used in the array
28  if   (j2 < dimension)
29      w.positions[j2] = -1
30  end if
```

Algorithm 3.3: **PR** *with functions* - Case II: RSS is exceeded

Complexity

We end this section with a study of the complexity of the algorithms **PR**, and by a theoretical comparison with algorithm **E**. The complexity of the product of a vector by a classic tensor product term is obtained by observing the number of multiplications of a vector slice by a column of a matrix. In the case of algorithm **E**, we perform $\#PSS$ products, while we only perform $\#RSS$ for algorithm **PR** if we remain in RSS. When we leave RSS, we cannot exploit the properties that permitted us to reduce the number of computations. The improvement with respect to algorithm **E** is therefore zero. This algorithm is therefore only interesting from the point of view of amount of computation if we remain in RSS.

In the case of functional matrices, the complexity remains of the same order (the number of computations does not change) but the evaluation of the matrix represents an important cost. We reduce the number of evaluations with respect to algorithm **E** if certain vector–matrix multiplications yield a zero result. If the model has a high percentage of nonreachable states, this decrease in the number of evaluations is significant.

As far as the amount of memory occupied by **PR** is concerned, we note that the vectors are stored in a reduced format, which allows an improvement in memory with respect to the shuffle algorithm. The vectors are of size $2 \times \#RSS$ (two arrays of size $\#RSS$) in reduced format while they were of size $\#PSS$ in extended format. However, the intermediate structures which are used for the shuffle (**PR**) are stored in extended format. We use arrays of size $nright_i \times n_i$ in which to store the z_{in} and other intermediate information. By definition, $nright_i \times n_i \times nleft_i = \#PSS$. Thus we use an array of size $\#PSS$. If it is impossible to hold this array in memory, the algorithm cannot be used. The classic shuffle algorithm (**E**) stores the vector in a structure of size $\#PSS$. The improvement in memory of the new algorithm is therefore zero.

We now wish to improve this algorithm from the perspective of memory usage, so that we will have no need of any data structure of size $\#PSS$.

4 Improvements in Memory Requirements

The algorithms of the previous section (**PR**) show improvements from the point of view of execution time: when it is known in advance, only nonzero elements of the resulting vector are computed. However, the amount of computation remains high once the vector leaves the RSS, and the gain in memory when compared to algorithm **E** is zero, because we still use data structures of size $\#PSS$. In this section we present new algorithms which use no data structure of size $\#PSS$, but which lose a little in computation time in cases when we remain in RSS.

Memory Structures

For this new algorithm, we used the Standard Template Library (STL). This library has *containers* in which to store and organize a set of objects, *iterators* with which to sweep across

these containers, and *generic algorithms* which act on the containers (sorting algorithms, algorithms for searching, and so on). We will use containers containing triplets (number, place, index). A triplet is stored per element of an l-slice.

- **number** represents the number of z_{in} or of z_{out} corresponding to the element.
- **place** represents the place of the element in the z_{in} or in the z_{out}.
- **index** represents the index of the element in the reduced vector. This allows us to find the value of the element as $vec[index]$.

Initialization and division into l-slices

The initialization phase is similar to that presented in the previous section. Indeed, the structures used are the same. Also, we need to know the elements of the RSS. The division into l-slices is handled in the same way. (Figure 6).

Collecting information

We handle the l-slices in a sequential manner, just like the algorithms of section 3 (**PR**). Thus, for each l-slice, we begin by collecting the necessary information for an efficient execution of the remainder of the algorithm. In the case in which we remain in RSS, the array *positions* of the output vector w is initialized with the positions of the states of RSS. We then perform:

- a traversal of the l-slice of the *positions* of the incoming vector v, to construct a container *infozin*. This container indicates for each element of the vector (reachable state), the z_{in} to which it belongs, its place in this z_{in}, and the index of this element in the reduced vector.
- a traversal of the l-slice of the *positions* of the output vector w, to complete a container *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 has been obtained by a case that exits the RSS).

In the case in which we exit RSS, only the traversal of the l-slice of the *positions* of the incoming vector v is necessary. We do not know which elements are likely to be nonzero in the resulting vector. The traversal is similar to that performed in the case in which we remain in RSS; we fill the array *infozin* in the same manner.

Extraction of the z_{in}

In order to extract the z_{in}, we just need to perform a sort[6] on the containers so that all the elements of the same z_{in} end up in adjacent positions. We then perform a loop on the

[6] The STL sort used is *introsort*, a variant of *quicksort* which offers a complexity of $O(N \log N)$ in the worst case.

z_{in} (on $nright_i$). We use an iterator, *izin* to traverse *infozin*, and, if we remain in RSS, a second iterator, *izout* to traverse *infozout*. 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. Figure 9 shows this extraction of the z_{in}.

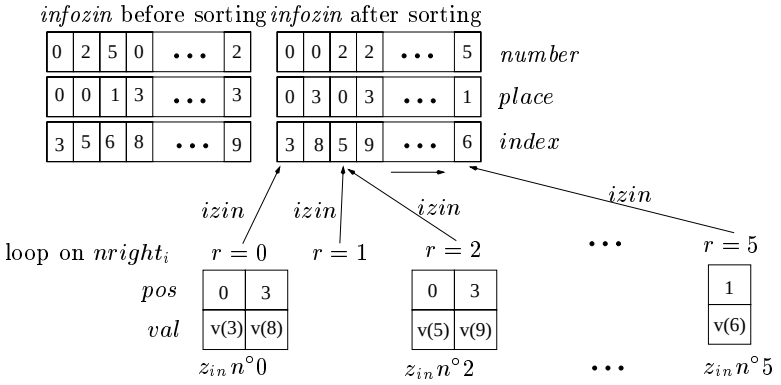


Figure 9: Extracting the z_{in} from *infozin*

The sequence of operations depends on the following cases:

- If we remain in RSS, we check if there are elements of the current z_{out} to be treated. The iterator *izout* lets us know which is the next z_{out} that contains the elements of RSS; we therefore know if there are elements likely to be nonzero in the current z_{out}. If there are no elements in this z_{out}, we increment the subvector with the local states of automata on the right of the matrix. If not, we extract all the elements of the current z_{in} (situated after *izin*). We look to see which elements need to be multiplied (elements of the current z_{out} determined from *izout*), and we can finally perform the multiplication of the z_{in} by the column of the matrix. We have all the information we need available to us.
- If RSS is exceeded, the treatment is similar, but *infozout* no longer exists. We extract the current z_{in} from the *izin*, then we perform the sparse multiplication of z_{in} by the entire matrix. The treatment for recuperating the nonzero elements and storing them in the resulting vector is then identical to that of Algorithm 3.3.

Algorithms 4.1 and 4.2 correspond respectively to the case in which we remain in RSS and the case in which we exit the RSS, with the evaluation of functions in the innermost loop.

Algorithm 4.1

```
1   j = 0; j2 = 0;
2   initialize   l_1 = 1, l_2 = 1, ... , l_{i-1} = 1;
3   for l = 0, 1, ... , nleft_i - 1          // loop over the l-slices
4   do  delete infozin and infozout;
5       initialize izin and izout;
        // Fill in infozin
6       while   (j < v.dimension) && (v.positions[j] < (l + 1) × njump_i)
7       do  re = (v.positions[j] − l × njump_i) mod nright_i;
8           qo = (v.positions[j] − l × njump_i) div nright_i;
9           append (re,qo,j) to infozin;
10          j = j + 1;
11      end do
        // Fill in infozout
12      while   (j2 < w.dimension) && (w.positions[j2] < (l + 1) × njump_i)
13      do  re = (w.positions[j2] − l × njump_i) mod nright_i;
14          qo = (w.positions[j2] − l × njump_i) div nright_i;
15          append (re,qo,j2) to infozout;
16          j2 = j2 + 1;
17      end do
18      sort infozin and infozout according to their numbers;
19      initialize   r_{i+1} = 1, r_{i+2} = 1, ... , r_N = 1;
20      for r = 0, 1, ... , nright_i - 1          // detail of an l-slice: loop over the z_in
21      do  (numo, poso, indo) = izout;
22          if   (numo == r)
23              delete z_in;
24              (numi, posi, indi) = izin;
25              while   (numi == r)      // extract z_in
26              do  append (posi, v.vec[indi]) to z_in
27                  increment izin;
28                  (numi, posi, indi) = izin;
29              end do
30              evaluate   Q^(i)(a_{l_1}^(1), ... , a_{r_N}^(N));
31              while   (numo == r)      // Compute result
32              do  w.vec[indo] = z_in × Q^(i)[poso];      // Multiplication by a column
33                  increment izout;
34                  (numo, poso, indo) = izout;
35              end do
36          end if
37          next   r_{i+1}, r_{i+2}, ... , r_N;
38      end do
39      next   l_1, l_2, ... , l_{i-1};
40  end do
```

Algorithm 4.1: **FR** *with functions* - Case I: Remain in RSS

Algorithm 4.2

```
1   j = 0; j2 = 0;
2   initialize   l_1 = 1, l_2 = 1, ... , l_{i-1} = 1;
3   for l = 0, 1, ... , nleft_i - 1          // loop over the l-slices
4   do  delete infozin and infozout;
5       initialize izin and izout;
        // Fill in infozin
6       while   (j < v.dimension) && (v.positions[j] < (l + 1) × njump_i)
7       do  re = (v.positions[j] - l × njump_i) mod nright_i;
8           qo = (v.positions[j] - l × njump_i) div nright_i;
9           append (re,qo,j) to infozin;
10          j = j + 1;
11      end do
12      sort infozin according to number;
13      initialize   r_{i+1} = 1, r_{i+2} = 1, ... , r_N = 1;
14      for r = 0, 1, ... , nright_i - 1          // detail of the l-slice: loop over the z_in
15      do  (num, pos, ind) = izin;
16          if   (num == r)
17              delete z_in;
18              while   (num == r)      // extract z_in
19              do  append (pos, v.vec[ind]) to z_in
20                  increment izin;
21                  (num, pos, ind) = izin;
22              end do
23              evaluate   Q^(i)(a_{l_1}^(1), ... , a_{r_N}^(N));
24              multiply   z_out = z_in × Q^(i);
25          end if
26          next   r_{i+1}, r_{i+2}, ... , r_N;
27      end do
28      sort z_out according to increasing position number;
29      while   ((pos, val) ∈ z_out)          // Store the result of the l-slice
30      do  w.positions[j2] = pos + l × njump_i;
31          w.vec[j2] = val;
32          j2 = j2 + 1;
33      end do
34      if   (j2 ≥ dimension)
35          Warn the user of array overflow, or else reallocate dynamically.
36      end if
37      next   l_1, l_2, ... , l_{i-1};
38  end do
    // If needed, place a marker to keep the last position used in the array
39  if   (j2 < dimension)
40      w.positions[j2] = -1
41  end if
```

Algorithm 4.2: **FR** *with functions* - Case II: RSS is exceeded

Complexity of the algorithm

The number of multiplications of a vector slice by a column for the algorithm **FR** remains identical to that of algorithm **PR** ($\#RSS$). We also need to do a sort per l-slice, but the complexity of the sort is less that the complexity generated by all the other computations. We sort a structure of maximum size $\#RSS$, while the traversals performed are of the order of $\#PSS$ ($nright_i$ traversals). When the percentage of nonreachable states is high, the improvement is significant ($\#PSS \gg \#RSS$).

The gain in memory is equally important for we no longer use arrays of size $nright_i$. All the arrays are of size $\#RSS$. We have therefore fulfilled our objective of not using any structure of size $\#PSS$ while maintaining an efficient algorithm.

Now that we have presented shuffle algorithms that take advantage of reduced probability vector structures, we present an algorithm for reordering automata that allows us to improve the shuffle algorithm in certain cases, by reducing the number of function evaluations that must be performed, for all three categories of algorithm **E**, **PR** and **FR**.

5 Reordering Automata to Improve Performance

Several methods for evaluating functions during the the vector–descriptor multiplication of the shuffle algorithm are possible ([7, 15]). The first, which we call the **no permutation method**, performs the function evaluation in the innermost loop of the algorithm. The algorithms detailed previously used this method. We may also use permutations to reduce the number of function evaluations (which we shall call **method with permutation**). The reordering techniques that are presented in [7, 8], allow us to perform, in certain cases, these evaluations outside the innermost loop. We will not present the theory behind these reordering techniques here, but will only show their effect on the algorithms that we have developed in this paper.

To implement these techniques, we must first perform a vector permutation. When we use reduced vectors, this must obviously be modified; this is why we give details of this algorithm in both the expanded and reduced vector cases. We shall then describe the modifications that must be made to the vector–descriptor multiplication algorithms that have been presented.

5.1 The Permutation Algorithm

The vector permutation algorithm must change if the structure of the vector changes. We shall first present the principles of the permutation, and explain how it works. Then we shall provide details of the algorithm in the extended vector case, and finally its adaption to the reduced vector case.

Permutation Principle

We assume that we possess a reordering of the automata that allows us to reduce the number of function evaluations. When the order of the automata change, the elements of the vector must change also, because their lexicographical ordering has changed. The table below shows the states of a vector when the automata are ordered in two different ways. The elements of the vector are sorted according to a lexicographical ordering expressed by the list of the automata in some order. On the left, we use the order $\mathcal{A}^{(1)}, \mathcal{A}^{(2)}, \mathcal{A}^{(3)}$. The automata have size 2, 3, 2 respectively. On the right, we have reordered the automata as $\mathcal{A}^{(3)}, \mathcal{A}^{(1)}, \mathcal{A}^{(2)}$.

$\mathcal{A}^{(1)}, \mathcal{A}^{(2)}, \mathcal{A}^{(3)}$				$\mathcal{A}^{(3)}, \mathcal{A}^{(1)}, \mathcal{A}^{(2)}$			
position	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$	position	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$
0	0	0	0	0	0	0	0
1	0	0	1	1	0	1	0
2	0	1	0	2	0	2	0
3	0	1	1	3	1	0	0
4	0	2	0	4	1	1	0
5	0	2	1	5	1	2	0
6	1	0	0	6	0	0	1
7	1	0	1	7	0	1	1
8	1	1	0	8	0	2	1
9	1	1	1	9	1	0	1
10	1	2	0	10	1	1	1
11	1	2	1	11	1	2	1

No matter which order is used, it is always easy to increment the state of the vector by +1. Incrementing a vector corresponds to incrementing a local state, with a possible carry-over, as explained previously in the case of the shuffle algorithm (Section 2). We provide details of this algorithm to allow for a better understanding of the permutation algorithm, which is founded on these increments.

To perform the increment, we consider the automata from the last to the first, and we try to increment the individual states of the automata. An incrementation is invalid when the individual state of the automaton to be incremented is its last state. When an incrementation is invalid, the individual state of the automaton is restored to its first state (*reset*), and the next state is considered (which is the automaton preceding it in the automata ordering). The procedure stops as soon as a valid incrementation can take place. In the previous example, and for the order $\mathcal{A}^{(1)}, \mathcal{A}^{(2)}, \mathcal{A}^{(3)}$, we look, for example, to increment the state $(0, 1, 1)$ by +1. The automaton $\mathcal{A}^{(3)}$ is the first that we try to increment. But this automaton is in its last state, and so its value is set to 0. We then try to increment $\mathcal{A}^{(2)}$, and this incrementation is valid. Thus the state $(0, 2, 0)$ is obtained.

When we perform, for a permutation, an increment according to some lexicographical ordering, we need to be able to compute the position of the state in the permuted vector, i.e., according to a different lexicographical ordering. We compute this new position from the current position in the permuted vector. In our example, the position of the state $(0, 1, 1)$

in the new lexicographical ordering $\mathcal{A}^{(3)}, \mathcal{A}^{(1)}, \mathcal{A}^{(2)}$ is 7. The positions of the state after the incrementation (state $(0, 2, 0)$) is 2. To compute the new position, we proceed by increments and decrements corresponding to the increments and *resets* performed on the individual states of the automata. The values of these increments depends on the new lexicographical order. A valid increment of $+1$ on an individual state corresponds to an addition of $nright_i$ on the order, and a *reset* of the state of an automaton corresponds to a subtraction of $nright_i \times n_i$ on the position. In our example we perform a *reset* on the state of automaton $\mathcal{A}^{(3)}$, and in the new lexicographical order $nright_3 = 2 \times 3 = 6$. We therefore obtain the position $7 - 6 = 1$. We then increment the value of the state of automaton $\mathcal{A}^{(2)}$, for which $nright_2 = 1$ in the new lexicographical order. We therefore increment the order 1, to obtain 2.

We wish to perform the permutation of the vector in traversing this vector sequentially. We know the position of the initial state according to the new lexicographical order. We perform an increment of $+1$ on the state of the vector to be permuted and we compute the position of the next state according to the new automata order, as explained above. The permutation algorithm for extended vectors is trivial and is presented in the next paragraph. We shall then show how this can be adapted for the case of reduced vectors.

Permutations for extended vectors

The permutation of a vector in extended format is immediate. We take advantage of the change in the lexicographical order. The algorithm creates a new vector structure (an array of size $\#PSS$). We traverse sequentially the vector to be permuted, v, and we see which is the new place for each element in the permuted vector w (position idx). In order to know this position, we increment the current state (*state*) by $+1$, and we compute the position (idx) of this state in the permuted vector. Figure 10 and Algorithm 5.1 present the premutation of a vector stored in extended format.

Algorithm 5.1

```
1  create a new array w of size #PSS
2  initialize state ;          // initial state in v
3  idx = 0;                    // index in w (position of state in the permuted vector)
4  for i = 0, 1, ..., #PSS     // sequential traversal of v
5  do  w[idx] = v[i];          // element of the vector in position i: now at position idx
6      increment the state state by +1
7      update the position idx according to the new lexicographical order
8  end do
9  v = w;
```

Algorithm 5.1: Permutation of a vector v in extended format

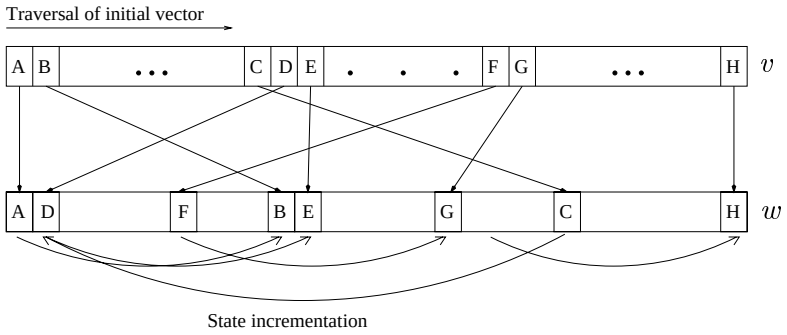


Figure 10: Permutation of a vector in extended format

Permutation of a Reduced Vector

In Algorithm 5.1, we perform accesses to w to store the element corresponding to position idx in this vector (line 5). When the vector is stored in reduced format, this operation is no longer possible because we only store the reachable states and we do not have a direct correspondance between idx and the index of the state in the extended vector. This last index depends on the number of reachable states that precede the state in position idx according to the new lexicographical order in the resulting vector.

We choose nevertheless to proceed in a similar manner to that of the extended algorithm. Thus we perform a loop on $\#PSS$. We proceed by increments of +1 on the state of the vector (*state*). We can, as before, compute the position of this state in the permuted vector (idx). When the state of the vector v is reachable (value present in the vector v which is stored in reduced format), we store the pair (new position, value) in an STL vector called w. Once this loop has terminated, we possess all the values to store in the resulting vector, as well as their positions. However, these positions are not necessarily in increasing order. Thus we must sort this vector in increasing order and then store the result in a reduced vector (arrays *newpos* and *newvec*). The iterator iw allows us to traverse the STL vector w. Algorithm 5.2 performs this permutation.

5.2 Vector–Descriptor Multiplication Algorithms

Once the permutation algorithm has been taken care of, we can apply the reordering technique and hence perform the function evaluations outside the innermost loop. Only one matrix evaluation per l-slice needs to be performed. The modifications that must be performed on the algorithms presented in the previous sections are now considered.

Algorithm 5.2

```
1  create new arrays newpos and newvec of size #RSS
2  initialize state ;          // initial state in v
3  idx = 0;          // index in w (order of state in the permuted vector)
4  k = 0;          // pointer to the current element of the reduced vector
5  for i = 0, 1, ... , #PSS
6  do  if   (i == positions[k])          // got a reachable state
7           append (idx, vec[k]) to w
8           k = k + 1;
9      end if
10     increment state by +1
11     update the position idx according to the new lexicographical order
12 end do
13 Sort w and initialize iw at the beginning of w
14 k = 0;
15 while   (iw is not at the end of w) // store the elements of w in newpos and newvec
16 do  (pos, val) = iw;
17     newpos[k] = pos;
18     newvec[k] = val;
19     k = k + 1; increment iw;
20 end do
21 positions = newpos; vec = newvec;
```

Algorithm 5.2: Permutation of a reduced vector v

Algorithm E (Section 2)

In the case of the expanded shuffle algorithm, it is sufficient to simply change the place of the line that corresponds to the evaluation of the matrix in Algorithm 2.3, to move it out of the innermost loop. Line 12 is therefore moved to a new position between lines 5 and 6.

Algorithm PR (Section 3)

In the case in which we leave the RSS, algorithm **PR** (Algorithm 3.3) is modified by simply moving line 13 and putting it between lines 11 and 12 (this moves the matrix evaluation outside the innermost loop).

On the other hand, a simple modification of Algorithm 3.2 is not the most appropriate solution in the case in which we remain within the RSS. Algorithm 3.1 without function evaluations is much more simple than Algorithm 3.2, but we must rewrite it in order to recreate the innermost loop in which the function evaluations should be handled. We are therefore obliged to treat the z_{in} in a sequential order, and, especially, to compute all the resulting elements obtained by the multiplication of the same z_{in} by a column of the matrix, one after the other, to avoid costly re-evaluations.

When we only perform a single matrix evaluation per l-slice, the order in which we handle the z_{in} is no longer important. This means that we can take Algorithm 3.1, which performs the computation of the elements of the result sequentially, without worrying about the order of the z_{in}. It is therefore sufficient to add the vector initializations, the increments of the left and right subvectors and the evaluation of the matrix between lines 7 and 8, to this algorithm.

Algorithm FR (Section 4)

In the algorithms presented in this section, the structure must be maintained. We shall therefore be content to move the evaluation out of the innermost loop by moving the corresponding line. For the case in which we remain in RSS (Algorithm 4.1), we move line 30 so that it lies between lines 19 and 20. When RSS is exceeded (Algorithm 4.2), we move line 23 so that it lies between lines 14 and 15. In this way, we only perform a single matrix evaluation per l-slice.

6 Comparison Tests

Now that the principal 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. We present results obtained from three classic models chosen from the literature ([7, 8, 15]). 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; and the third, **queue**, represents a queueing network model with blocking and priority service.

6.1 Test Conditions

All numerical experiments were performed using the software package, PEPS ([14]), into which we implanted the three 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 and running Linux Mandrake 2.2.14. 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.

The changes that we made to PEPS only concerned the vector descriptor multiplication. As a result, the improvement or deterioration in performance of the new algorithms are identical for all the iterative methods (Power, Arnoldi, GMRES) and for all the different types of preconditioning. Because of this we restrict our numerical experiments to a single iterative method, *the unpreconditioned power method.*

Presentation of Results

The results are presented in two tables for each model. The first gives the results in terms of execution time, and the second in terms of memory needs. The two tables have the same structure.

- The initial columns give the parameters of the model (values of N and P for the mutex examples), as well as the number of states of the system, (column $\#PSS$), the number of reachable states, (column $\#RSS$), and their ratio, (column %), given as $(\#RSS/\#PSS) \times 100$.
- Column E (extended) presents the results obtained when the algorithm of section 2 is used, i.e., the extended shuffle algorithm **E**.
- Column PR (partially reduced) presents the results obtained by the algorithms of Section 3, **PR**, which improve execution time, but which use intermediate data structures that are of size $\#PSS$.
- The results of column FR (fully reduced) are obtained using the algorithms presented in Section 4 (**FR**), which reduce memory requirements.

The computation times are actually given in two subcolumns, the first provides the compilation time (comp) while the second subcolumn presents the execution time (exec).

The memory requirements are also given in two subcolumns, the first provides the size of the descriptors (des) while the values given in the second (exec) are taken from the system during execution. They represent 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 (vectors, intermediate array structures, and so on).

6.2 Mutex1

In the resource sharing model, called **mutex1**, N distinct clients share P identical units of a common resource. Each client is modeled by an automaton having two states: *sleeping* and *active*. In state *sleeping*, the client does not use any resource while in state *active*, it uses one unit of the common resource.

Notice that if $P = 1$, this model represents the classical mutual exclusion model whith n clients. Similarly if $P \geq N$, this model represents the case in which the clients are completely independent.

An automaton $\mathcal{A}^{(i)}$ in *sleeping* mode can move to state *active* according to a rate λ_i multiplied by a function f defined by[7]:

$$f(\tilde{x}) = \delta\left(\sum_{i=1}^{N} \delta(x^{(i)} = active) < P\right)$$

This function represents 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*. A unit of resource is freed at a different rate for each automaton $\mathcal{A}^{(i)}$ (μ_i). Freeing up resource, as opposed to acquiring resource, occurs in an independent manner. Figure 11 illustrates this model.

This model has no synchronizing transitions; it has local transitions only. We use this model to test the algorithms in the case when we remain in RSS, with functional matrices. For the tests, we use the values $\lambda_i = 6$ and $\mu_i = 9$ for all automata. The values of N and P are varied according to the experiment.

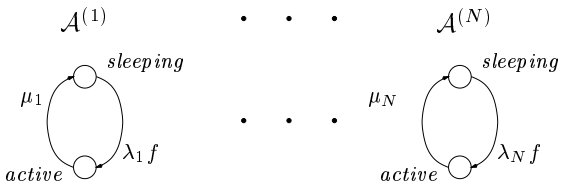


Figure 11: The resource sharing model, **mutex1**

Method without permutation

The first tests were performed with the method without permutation on a mutex1 model with 16 clients, and a number of units of resources that varied between 1 and 16.

[7] Recall that $\delta(b)$ is a function that equals 1 if the expression b is true, otherwise this function is equal to zero.

Execution time (sec) - mutex1										
Model					E		PR		FR	
N	P	$\#PSS$	$\#RSS$	%	comp	exec	comp	exec	comp	exec
16	1	65536	17	0.03%	2.4	27.2	0.02	1	0.02	3.5
16	4	65536	2517	3.8%	2.4	112	0.14	11.5	0.14	22.1
16	6	65536	14893	22.7%	2.4	178.2	0.68	70	0.68	93.1
16	8	65536	39203	59.8%	2.4	235.4	1.6	206.6	1.6	267.4
16	10	65536	58651	89.5%	2.7	292.7	2.3	348.6	2.3	478.9
16	12	65536	64839	98.9%	2.6	332	2.5	419.6	2.5	612.9
16	16	65536	65536	100%	3.7	27	3.7	84.9	3.7	377.8

This example shows the performance of the algorithms when the percentage of nonreachable states is varied. The compilation is identical for algorithms PR and FR, and faster than the compilation of algorithm E. In fact, algorithm E needs to generate all the diagonal elements while PR and FR store only the diagonal elements corresponding to reachable states, and therefore only compute these elements during the compilation.

As concerns the new algorithms PR and FR presented in this paper, we notice, as expected, that algorithm PR is faster than algorithm FR. These algorithms, which take advantage of a large percentage of nonreachable states, are faster than algorithm E so long as the percentage of reachable states remains reasonable (less than 60% for PR, less than 50% for FR).

Memory used (KB) - mutex1										
Model					E		PR		FR	
N	P	$\#PSS$	$\#RSS$	%	des	exec	des	exec	des	exec
16	1	65536	17	0.03%	522	4208	10	4476	10	2124
16	4	65536	2517	3.8%	522	4208	30	4584	30	2304
16	6	65536	14893	22.7%	522	4208	126	5116	126	3128
16	8	65536	39203	59.8%	522	4208	316	6176	316	4748
16	10	65536	58651	89.5%	522	4212	468	7016	468	6040
16	12	65536	64839	98.9%	522	4212	516	7280	516	6448
16	16	65536	65536	100%	522	4212	522	7832	522	6516

As for memory needs, we note first of all, that the size of the descriptor is fixed in the case of algorithm E, while it depends on the number of reachable states for the other algorithms. This is linked to the fact that in the case of PR and FR we only store the elements of the diagonal corresponding to reachable states, while E stores all the elements of the diagonal.

During execution, we notice that algorithm FR permits a significant reduction in memory needs with respect to algorithm E as long as the the percentage of nonreachable states is sufficiently high. Indeed, when we are working in reduced vector format, we use structures of the size of $\#RSS$, 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 $\#RSS$, which is greater than $\#PSS$. The intermediate structures used in PR make its memory use greater in all cases than that of algorithm E.

We performed additional experiments on larger models to determine the limits and the possibilities of the algorithms developed. The indication (-) in the tables means that the algorithm did not terminate, due to a need for excessive memory. For $N = 24$, only the column FR appears in the table, because the two other algorithms did not terminate.

Execution time (sec) - mutex1										
Model					E		PR		FR	
N	P	$\#PSS$	$\#RSS$	%	comp	exec	comp	exec	comp	exec
20	1	1048576	21	0.002%	51.8	601.7	0.2	21.2	0.2	64.4
20	4	1048576	6196	0.6%	56.4	2266.7	0.6	105	0.6	263.6
20	6	1048576	60460	5.8%	59.5	3285.1	3.9	440.9	3.9	698
20	8	1048576	263950	25.2%	62.9	4675.1	15.8	1952.7	15.8	2604.2
20	10	1048576	616666	58.8%	58.6	6154.7	34.1	5160.1	34.1	7047.5
20	20	1048576	1048576	100%	92.9	586.6	88	2159.5	88	12585.3

Model					FR	
N	P	$\#PSS$	$\#RSS$	%	comp	exec
24	1	16777216	25	0.0001%	2.6	1100.9
24	4	16777216	12952	0.08%	3.7	3986.6
24	6	16777216	190051	1.1%	18.7	7018.6
24	8	16777216	1271626	7.6%	102.3	19671.4
24	10	16777216	4540386	27%	331.6	101348.6
24	12	16777216	9740686	58%	672.6	-

Memory used (KB) - mutex1										
Model					E		PR		FR	
N	P	$\#PSS$	$\#RSS$	%	des	exec	des	exec	des	exec
20	1	1048576	21	0.002%	8207	35164	15	39776	15	2380
20	4	1048576	6196	0.6%	8207	35164	63	40040	63	2816
20	6	1048576	60460	5.8%	8207	35164	487	42400	487	6428
20	8	1048576	263950	25.2%	8207	35164	2077	51128	2077	19932
20	10	1048576	616666	58.8%	8207	35164	4833	66292	4833	43360
20	20	1048576	1048576	100%	8207	35164	8207	84860	8207	72052

Model					FR	
N	P	$\#PSS$	$\#RSS$	%	des	exec
24	1	16777216	25	0.0001%	21	6252
24	4	16777216	12952	0.08%	122	7116
24	6	16777216	190051	1.1%	1506	18896
24	8	16777216	1271626	7.6%	9956	90712
24	10	16777216	4540386	27%	35493	307768
24	12	16777216	9740686	58%	76120	424808

The results obtained with $N = 20$ confirms our previous observations ($N = 16$).

Algorithm PR however, uses a great deal of intermediate memory structures which limits its application on large models. Its usefulness lies in the acceleration that it brings to models with a large percentage of nonreachable states and which are not too large (it is therefore better than FR). So, for $N = 20$, PR is better than FR in terms of execution time. As for algorithm E, it is most appropriate for models with few nonreachable states. It is the best algorithm when $P \geq 10$.

When $N = 24$, only algorithm FR is successful. The limits of algorithm FR are reached when the percentage of nonreachable states is high. When $P \geq 12$, the memory needs of algorithm FR become excessive and the solution cannot be computed. Here we see that algorithm FR takes advantage of the large number of nonreachable states; if the model has few unreachable states, FR becomes inefficient and unusable.

Method with permutation

We performed another series of test on this model, but this time using the algorithms with permutations. We restricted ourselves to the case when $P = 16$ to observe the behavior of the algorithms, knowing that this method performs less well than those without permutation, since all the evaluations must take place. This means that we perform permutations without eliminating function evaluations. The data relative to the compilation and the size of the descriptor also remain identical. We present the results concerning the execution time and memory needs in the same table.

Execution time (sec) and Memory used (KB) - mutex1										
Model					E		PR		FR	
N	P	$\#PSS$	$\#RSS$	%	sec	KB	sec	KB	sec	KB
16	1	65536	17	0.03%	32.2	4208	29.5	4496	29.3	2144
16	4	65536	2517	3.8%	132.4	4208	124	4636	123.1	2292
16	6	65536	14893	22.7%	209.8	4208	220	5312	217.7	2972
16	8	65536	39203	59.8%	277.9	4208	367	6656	368.1	4304
16	10	65536	58651	89.5%	345.8	4212	545.7	7724	542.7	5372
16	12	65536	64839	98.9%	392	4212	659.8	8060	656.6	5708
16	16	65536	65536	100%	26.8	4212	99	7344	379.5	6536

We can see from these results that algorithm PR and FR yield a reduction in execution time that is less than that obtained in the method with permutation. This is due to the fact that the permutation of reduced vectors results in an additional cost with respect to the permutation of extended vectors. We can also note that algorithm FR performs better than PR in these examples, both in terms of computation time and in memory needs. In the future, when we use the methods with permutation in models without synchronizing events, we shall use algorithm FR in place of PR.

6.3 Mutex2

It is possible to represent the resource sharing model by means of a stochastic automata network without the use of functions: **mutex2**. This representation produces a more complex model and makes execution times longer, but it is nevertheless interesting to test the algorithms without function evaluation and when we leave the RSS (from handling synchronizing events). To model the resource sharing, we continue to use one automaton per client, with the same two states *sleeping* and *active*. An additional automaton of size $P+1$ is also used, where P is the number of units of resource available.

An automaton $\mathcal{A}^{(i)}$ in state *sleeping* can move to the state *active* by a synchronizing event, p_i which removes a unit of resource from the resource pool. Similarly, when an automaton finishes using the resource, it uses a synchronizing event, r_i, to return the resource to the pool. Figure 12 illustrates this concept.

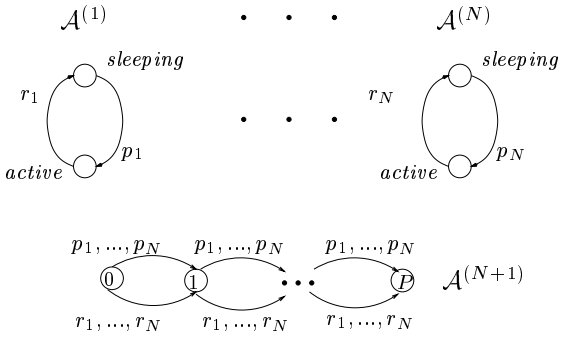


Figure 12: The resource sharing model, **mutex2**

There are no functions in this example. The method without permutation is therefore identical to the method with permutation.

Execution time (sec) - mutex2										
Model					E		PR		FR	
N	P	$\#PSS$	$\#RSS$	%	comp	exec	comp	exec	comp	exec
16	1	131072	17	0.01%	33.6	13.6	0.07	6.1	0.07	2.2
16	4	327680	2517	0.77%	84.3	85.4	0.7	40.1	0.7	11.1
16	6	458752	14893	3.25%	116.1	174.9	3.9	101.3	3.9	47.9
16	8	589824	39203	6.65%	149.1	296.5	9.9	219.8	9.9	150.3
16	10	720896	58651	8.14%	182.6	563.1	15.3	449.8	15.3	350.9
16	12	851968	64839	7.61%	215.6	1002.5	16.8	774.6	16.8	603.5
16	16	1114112	65536	5.88%	283.3	1861.5	17	1350	17	918.2

Memory used (KB) - mutex2										
Model					E		PR		FR	
N	P	$\#PSS$	$\#RSS$	%	des	exec	des	exec	des	exec
16	1	131072	17	0.01%	1210	6480	187	7928	187	2368
16	4	327680	2517	0.77%	2747	12672	207	21784	207	2628
16	6	458752	14893	3.25%	3772	16800	304	37440	304	3628
16	8	589824	39203	6.65%	4796	20928	494	59860	494	5564
16	10	720896	58651	8.14%	5821	25060	647	84604	647	7116
16	12	851968	64839	7.61%	6845	29188	696	112564	696	7628
16	16	1114112	65536	5.88%	8894	37452	702	179872	702	7780

Perhaps the first thing to note in these tests is the real inefficiency of algorithm PR which is worse than algorithm FR both from the point of view of memory needs and from execution time. The difference in memory needs can be explained by the fact that algorithm PR uses intermediate data structures that are of size $\#PSS$, while FR only uses reduced data structures. As for execution time, we have already seen that algorithm PR is not efficient when we leave RSS, because we do not know which elements of the result do not need to be computed, and hence we perform $\#PSS$ multiplications of vector slices by a column of the matrix. Algorithm FR 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 z_{in} by the matrix. These tests allow for a decrease in the number of multiplications, which in turn occasions an improvement in time when compared to algorithm PR which does not perform these tests. In what follows, we will no longer use algorithm PR when we leave RSS. We shall use algorithm FR in its place.

The use of synchronizing transitions creates a model with relatively few reachable states. We are, in effect, obliged to increase the state space, when compared to the model **mutex1**, but the reachable state space remains the same size. So, here we only test models in which algorithm FR is better that algorithm E.

Algorithm FR 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.

6.4 Queuing Network Model

The third model is that of an open queueing network with blocking and priority service, (**queue**). It consists of N finite capacity queues and $N-1$ different classes of customers. For $i = 1 \ldots N-1$, customers of class i arrive from the exterior to queue i according to an exponential distribution of rate λ_i. Customers are lost if the queue is full. The server at queue i serves customers according to an exponential distribution with rate μ_i.

Customers who have been served by queues $1 \ldots N-1$ proceed to queue N. If this queue is full, the customers are blocked and the servers of the other queues must stop. Hence, these servers cannot serve a new customer while queue N is full. When a place becomes available in this queue, the customers that have been blocked at the exit of the other queues are transferred towards queue N.

Queue N serves customers of class i ($i = 1 \ldots N-1$) according to an exponential distribution with rate μ_N^i. It is the only queue which serves all $N-1$ classes of customer. In this queue, for $1 \leq i < j \leq N-1$, customers of class i have priority over those of class j. Thus, a customer cannot be served in queue N if there is another customer with a higher priority in this queue. After receiving service in queue N, customers leave the system.

We shall let C_i denote the capacity of queue i (for $i = 1 \ldots N$). Figure 13 illustrates this model.

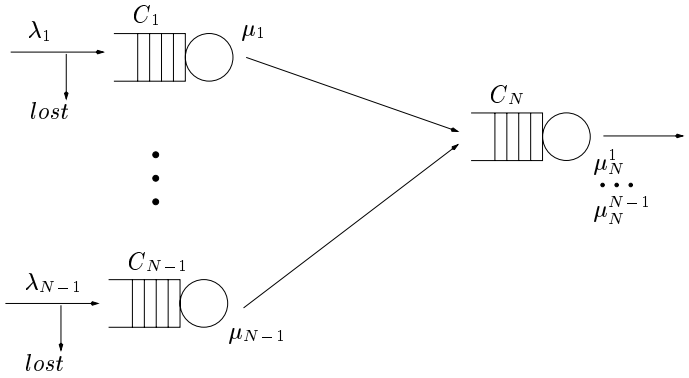


Figure 13: Open queuing network model with N queues with blocking and priority service

Queues i, for $i = 1 \ldots N-1$, are each represented by a single automaton, $\mathcal{A}^i$, with $C_i + 1$ states. When there are k customers in one of the queues, the automaton is in state k. The queue N needs $N-1$ automata for its representation; $\mathcal{A}^{N,i}$ corresponds to the number of class i customers present in the queue, for $i = 1 \ldots N-1$.

This SAN has $N-1$ synchronizing events: s_i $(i = 1 \ldots N-1)$ corresponds to a transfer of a customer of class i from queue i to queue N. In addition to these synchronizing events, this SAN uses functions. The first, f, is a function that restricts arrivals into queue N. This function has the value 0 when queue N is full, and 1 in all other cases.

$$f(\tilde{x}) = \delta\left(\left[\sum_{k=1}^{N-1} x^{(N,k)}\right] < C_N\right)$$

where $x^{(N,k)}$ represent the number of customers of class k $(k = 1 \ldots N-1)$ in queue N; $x^{N,k}$ corresponds to the state of automaton $\mathcal{A}^{N,k}$.

To describe the priority of different classes of customers, we use $N-2$ functions $g^{(k)}$, for $k = 2 \ldots N-1$.

$$g^{(k)}(\tilde{x}) = \delta\left(\left[\sum_{m=1}^{k-1} x^{(N,m)}\right] = 0\right)$$

The function $g^{(k)}$ expresses the priority of customers of class m, with $m < k$, over those of class k. This function has the value 0 when a customer of class m is present in queue N, and prevents, in this case, the service of any client of class k. Its value is 1 in all other cases. Figure 14 presents this stochastic automata network for numerical values $N = 3$, $C_1 = C_2 = 1$ and $C_3 = 2$.

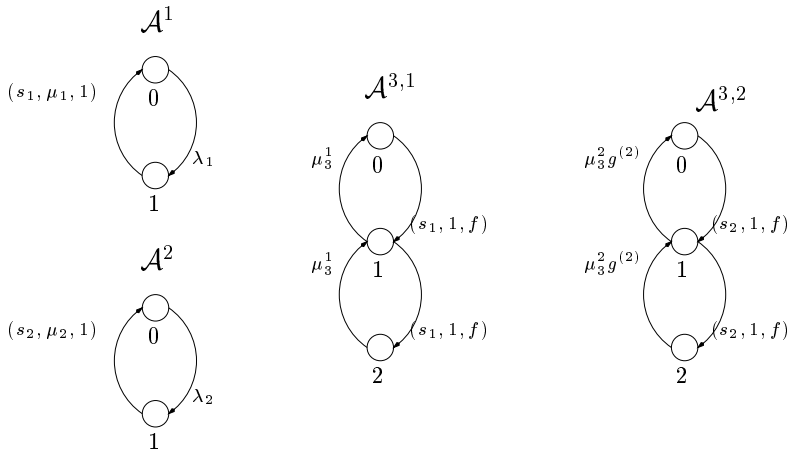


Figure 14: SAN model for the queueing network model with blocking and priority service

The set of reachable states (RSS) has size

$$\#RSS = (\prod_{i=1}^{N-1}(C_i + 1)) \times (C_N + 1) \times (C_N + 2) \times \ldots \times (C_N + (N-1))/(N-1)!$$

while the product state space (PSS) has size

$$\#PSS = (\prod_{i=1}^{N-1} (C_i + 1)) \times (C_N + 1)^{N-1}$$

Thus, for a fixed number of queues, N, the larger that C_N becomes, the more nonreachable states the model contains. For C_N tending towards infinity, $\#PSS = (N-1)! \times \#RSS$.

Results

The tests were performed using the method without permutation on a queuing network model, in which the number of queues N and the capacity of the queue N are varied. Thus, the size of the model and the percentage of reachable to nonreachable states changed. In all the examples, for $i = 1 \ldots N-1$, $C_i = 1$, $\lambda_i = 6$, $\mu_i = 9$ and $\mu_N^i = 9$.

Moreover, for these tests, we no longer use algorithm PR when RSS is exceeded, because we have already shown that this algorithm is inefficient in this case (results of the mutex2 model). We therefore use algorithm FR in this case. The column PR/FR is obtained by using algorithm PR when we remain in RSS, and algorithm FR when RSS is exceeded.

Execution time (sec) - queue										
Model					E		PR/FR		FR	
N	C_N	$\#PSS$	$\#RSS$	%	comp	exec	comp	exec	comp	exec
8	2	279936	4608	1.6%	18.9	5733.6	0.4	957.9	0.4	1496.3
8	3	2097152	15360	0.7%	147.2	68048.9	1.4	10583.5	1.4	15343.2

Memory used (KB) - queue										
Model					E		PR/FR		FR	
N	C_N	$\#PSS$	$\#RSS$	%	des	exec	des	exec	des	exec
8	2	279936	4608	1.6%	2222	10984	71	13608	71	2600
8	3	2097152	15360	0.7%	16419	68236	155	111240	155	3912

For this kind of model, algorithm FR is best in terms of memory used. In fact, the percentage of nonreachable states is high because of the synchronized transitions. The use of an algorithm combining PR and FR allow us to obtain an execution time that is superior to that obtained with FR alone. Synchronizations do not reduce the performances of algorithm PR/FR whereas it does reduce the performance of PR. However, the memory used by this algorithm is similar to the memory used by PR and is therefore a limitation to the application of this new algorithm.

When $N = 12$ and $C_N = 2$, we have $\#PSS = 362797056$ and $\#RSS = 159744$. This model with many states can be solved only with algorithm FR.

7 Conclusions

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

- **Algorithm E, 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** takes advantage when the percentage of nonreachable states is high to improve the performance of the vector–descriptor multiplication when we remain in the RSS. However, it uses intermediate structures that are of size $\#PSS$, which limits its applicability on very large models. Furthermore, when we leave the RSS the performances are worse than algorithm FR, and we do not recommend its use in this case. The gain in time with respect to the classic algorithm E is substantial when the models are not too large and have many nonreachable states, so long as we remain in RSS. The principal use of PR therefore resides in models with a moderate percentage of nonreachable states which are moderately large, which remain in RSS.
- **Algorithm FR** keeps all data structures to be of reduced size. This permits us to handle very large models, models which none of the other algorithms can handle. We notice however, a loss in computation time with respect to algorithm PR when we remain in RSS. This is the price we pay in order to be able to handle the largest models.

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. For this comparison, we have integrated into PEPS such an algorithm. The generation of the global matrix is unfortunately long with this software. However, when we have 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 limitated. Therefore, the new algorithms allow us to solve models that can't be solved with the HBF algorithm.

We have therefore fulfilled our objective: algorithm FR allows us to handle models which cannot be handled by any other algorithm. However, all the algorithms that were tested are better for a given type of model. To exploit the particularities of all these algorithms, we believe that is is appropriate to program a heuristic which automatically choses the most appropriate algorithm. Thus, in PEPS, the user may either choose the algorithm, or else leave the choice of the most appropriate algorithm for the particular model (taking into account all the parameters such as percentage of reachable states, size of model, etc.) to the program itself.

Finally, the new algorithms were 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 notable algorithms that use Petri nets, [2, 1, 12], were not included. Such a comparison will be performed in future work.

References

[1] P. Buchholz. Hierarchical Structuring of Superposed GSPNs. In *Proc. 7th Int. Workshop Petri Nets and Performance Models (PNPM' 97), St-Malo (France), pages 81–90.*. IEEE CS Press, 1997.

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

[3] G. Ciardo and A. Miner. Efficient reachability set generation and storage using decision diagram. In *proc. 20th int. Conf. Application and Theory of Petri Nets, LNCS 1 639, Springer*, 1999.

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

[5] S. Donatelli. Superposed Stochastic Automata: A Class of Stochastic Petri Nets with Parallel Solution and Distributed State Space. *Performance Evaluation*, Vol. 18, pp. 21–36, 1993.

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

[7] P. Fernandes. *Méthodes Numériques pour la Solution de Systèmes Markoviens à Grand Espace d'États.* Ph.D Thesis, University of Grenoble, 1998.

[8] P. Fernandes, B. Plateau and W.J. Stewart. Efficient Descriptor-Vector Multiplication in Stochastic Automata Networks. *JACM*, 1998, vol=45, nb=3, pp=381-414.

[9] P. Fernandes, B. Plateau and W.J. Stewart. Optimizing tensor product computations in Stochastic Automata Networks. *RAIRO, Operations Research*, 3, 1998, pp=325-351.

[10] H. Hermanns, J. Meyer-Kayser and 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*, 2000.

[11] P. Kemper. Reachability Analysis Based on Structured Representations. In J. Billington, W. Reisig (eds.) *Application and Theory of Petri Nets 1996*, Springer LNCS 1091 269-288 (1996).

[12] P. Kemper. Numerical Analysis of Superposed GSPNs. *IEEE Transactions on software engineering, vol. 22, no. 9*, September 1996.

[13] B. Plateau. On the Stochastic Structure of Parallelism and Synchronization Models for Distributed Algorithms. In *Proc. ACM Sigmetrics Conference on Measurement and Modelling of Computer Systems*, Austin, Texas, August 1985.

[14] B. Plateau, J.M. Fourneau and K.H. Lee. PEPS: A Package for Solving Complex Markov Models of Parallel Systems. In R. Puigjaner, D. Potier, Eds., *Modelling Techniques and Tools for Computer Performance Evaluation*, Spain, September 1988.

[15] B. Plateau, W.J. Stewart and P. Fernandes. On the Benefits of Using Functional Transitions in Kronecker Modelling. *Submitted to Performance Evaluation*. 2001.

[16] W.J. Stewart. *An Introduction to the Numerical Solution of Markov Chains*, Princeton University Press, New Jersey, 1994.

[17] W.J. Stewart. *MARCA : Markov Chain Analyser*. A Software Package for Markov Modelling. Version 3.0, 1996.

Contents

1 Introduction **3**

2 The Shuffle Algorithm **6**
2.1 The Non-functional case 7
2.2 Handling Functional Dependencies 13
2.2.1 Establishing the order of normal factors 14
2.2.2 Evaluation of functional elements 15

3 Improvements in Computation Time **19**
3.1 Multiplication details 19
3.2 Multiplication with Reduced Vector Data Structures 21

4 Improvements in Memory Requirements **31**

5 Reordering Automata to Improve Performance **36**
5.1 The Permutation Algorithm 36
5.2 Vector–Descriptor Multiplication Algorithms 39

6 Comparison Tests **41**
6.1 Test Conditions 42
6.2 Mutex1 43
6.3 Mutex2 47
6.4 Queuing Network Model 49

7 Conclusions **52**

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399