An Alternative Algorithm to Multiply a Vector by a Kronecker Represented Descriptor

Paulo Fernandes*, Ricardo Presotto†, Afonso Sales‡, Thais Webber§

Abstract

The key operation to obtain stationary and transient solutions of models described by Kronecker structured formalisms using iterative methods is the Vector-Descriptor product. This operation is usually performed with the Shuffle algorithm, which was proposed to Stochastic Automata Networks, but it is also currently used by Stochastic Petri Nets and Performance Evaluation Process Algebra solvers. This paper presents an alternative algorithm to perform the Vector-Descriptor product, called Slice algorithm. The numerical advantages of this new algorithm over the previous one is shown to SAN models, in which the computational costs are compared. Finally, we discuss some possible optimizations of the Slice algorithm and implementation issues regarding parallel versions of both algorithms.

1 Introduction

All formalisms used to model complex systems are based on a structured description. This is particularly the case of Markovian performance and reliability evaluation models. A myriad of formalisms is available in the research community, e.g., Stochastic Activity Networks [13], Queueing Networks [9], Stochastic Petri Nets (SPN) [1], Performance Evaluation Process Algebra (PEPA) [11], and Stochastic Automata Networks (SAN) [12]. Among such formalisms we are specially interested in those which use Tensor (or Kronecker) Algebra to represent the infinitesimal generator of the underlying Markov chain [7, 8]. Such tensor formula representation is referred in the literature as descriptor.

The key operation to perform iterative solutions, both stationary and transient, of models described as a descriptor is the multiplication by a probability vector [14]. Such operation is performed using the Shuffle algorithm [8] which has a quite efficient way to handle tensor structures and takes advantage of several techniques developed to SAN [4], to PEPA [10], and to SPN [6].

The main purpose of this paper is to propose an alternative algorithm to perform the vector-descriptor product, which we call Slice. The main advantage of the Slice algorithm is the possible reduction of computational cost (number of multiplications)

---

* PUCRS, paulof@inf.pucrs.br (corresponding author). P. Fernandes is partially funded by CNPq/Brazil.
† PUCRS, rpresotto@inf.pucrs.br
‡ PUCRS, asales@inf.pucrs.br
§ PUCRS, twebber@inf.pucrs.br, T. Webber is funded by CAPES/Brazil.
for very sparse tensor components. Such reduction is achieved keeping the compact
tensor format of the descriptor. In some way, the Slice algorithm can be considered
as a trade-off between the sparse matrix approach used for straightforward Markov
chains, and the fully tensor approach used by the Shuffle algorithm.

Nevertheless, this paper does not exploit the Slice algorithm possibilities to its lim-
its, since very few considerations are made concerning possible optimizations. In par-
ticular, we do not analyse the possible benefits of automata reordering according to
functional dependencies, which was deeply studied for the Shuffle algorithm. Also a
possible hybrid approach using Shuffle and Slice algorithms are not discussed in de-
tail. Actually, we focus our contribution in the presentation of an original way to handle
the vector-descriptor product and we present encouraging measures to develop further
studies based on this new approach.

This paper is organized with a brief introduction to the descriptor structure (Section
2). Section 3 presents the basic operation of the vector-descriptor product followed
by sections describing the Shuffle algorithm principle (Section 3.1) and the proposed
Slice algorithm (Section 3.2). In Section 4, we show some comparative measures of
both Shuffle and Slice algorithms applied to two distinct structured models. Finally the
conclusion points out some future works necessary to raise the Slice algorithm to a
similar level of optimization as the level already obtained by the Shuffle algorithm.

2 Tensor Represented Descriptor

Regardless of the structured formalism adopted, e.g., SAN, SPN, PEPA, the basic
principle consists in the representation of a whole system by a collection of subsystems
with an independent behavior (local behavior) and occasional interdependencies (syn-
chronized behavior). According to the formalism, the primitives to describe local and
synchronized behaviors may change their denomination, the reader interested in the
formalisms definitions can found information in [12, 4] for SAN, in [1, 6] for SPN,
and [11, 10] for PEPA.

For the purpose of this paper it is only important to consider that, unlike the non-
structured approaches, e.g., straightforward Markov chains, a structured model is not
described by a single sparse matrix, but instead, by a descriptor. For a structured model
with \(N\) subsystems and \(E\) synchronizing primitives, the descriptor (\(Q\)) is an algebraic
formula containing \(N + 2NE\) matrices:

\[
Q = \bigoplus_{i=1}^{N} Q^{(i)}_{l} + \sum_{j=1}^{E} \left( \bigotimes_{g}^{N} Q^{(i)}_{e_{j}} + \bigotimes_{i=1}^{N} Q^{(i)}_{e_{j}} \right)
\]

where:

- \(Q^{(i)}_{l}\) represents \(N\) matrices describing each local behaviors of the \(i^{th}\) subsystem;
- \(Q^{(i)}_{e_{j}}\) represents \(NE\) matrices describing the occurrence of synchronizing primitive \(e\) in the \(i^{th}\) subsystem;
- and \(Q^{(i)}_{e_{j}}\) represents \(NE\) analogous matrices describing the diagonal adjustment of
  synchronizing primitive \(e\) in the \(i^{th}\) subsystem.
Table 1 details descriptor $Q$, which is composed of two separated parts: a tensor sum corresponding to the local events; a sum of tensor products corresponding to the synchronizing events [8]. The tensor sum operation of the local part can be decomposed into the ordinary sum of $N$ normal factors, i.e., a sum of tensor products where all matrices but one are identity matrices$^1$. Therefore, in this first part, only the non-identity matrices ($Q_{I}^{(i)}$) need to be stored.

\[ \sum N \]

\[ e^+ \]

\[ e^- \]

Table 1: SAN descriptor

### 3 Vector-Descriptor Product

The vector-descriptor product operation corresponds to the product of a vector $v$, as big as the product state space ($\prod_{i=1}^{N} n_i$), by descriptor $Q$. Since the descriptor is the ordinary sum of $N + 2E$ tensor products, the basic operation of the vector-descriptor product is the multiplication of vector $v$ by a tensor product of $N$ matrices:

\[ \sum_{j=1}^{N+2E} \left( v \times \left[ \prod_{i=1}^{N} Q_{I}^{(i)} \right] \right) \]

where $Q_{I}^{(i)}$ corresponds to $I_{n_i}$, $Q_{e^+}^{(i)}$, $Q_{e^-}^{(i)}$, or $Q_{e^-}^{(i)}$ according to the tensor product term where it appears.

For simplicity in this section, we describe the Shuffle and Slice algorithms for the basic operation $v \times \text{tensor product term}$ omitting index $j$ from equation 2, i.e.:

\[ v \times \left[ \prod_{i=1}^{N} Q_{I}^{(i)} \right] \]

$^1I_{n_i}$ is an identity matrix of order $n_i$. 

59
3.1 Shuffle Algorithm

The Shuffle algorithm is described in this section without any considerations about optimizations for the evaluation of functional elements. A thorough study about matrices reordering and generalized tensor algebra properties with this objective can be found in [8]. All those optimizations aim to reduce the overhead of evaluate functional elements, but they do not change the number of multiplications needed by the Shuffle algorithm. Therefore, we ignore the functional elements in the context of this paper, and the basic operation (equation 3) is simplified to consider classical (⊗) and not generalized (⊙) tensor products.

The basic principle of the Shuffle algorithm concerns the application of the decomposition of a tensor product in the ordinary product of normal factors property:

\[
Q^{(1)} \otimes \ldots \otimes Q^{(N)} = (Q^{(1)} \otimes I_{n_2} \otimes \ldots \otimes I_{n_{N-1}} \otimes I_{n_N}) \times (I_{n_1} \otimes Q^{(2)} \otimes \ldots \otimes I_{n_{N-1}} \otimes I_{n_N}) \times \ldots \times (I_{n_1} \otimes I_{n_2} \otimes \ldots \otimes Q^{(N-1)} \otimes I_{n_N}) \times (I_{n_1} \otimes I_{n_2} \otimes \ldots \otimes I_{n_{N-1}} \otimes Q^{(N)})
\]

Rewritten the basic operation (equation 3) according to this property:

\[
v \times \left[ \prod_{i=1}^{N} I_{n_{left_i}} \otimes Q^{(i)} \otimes I_{n_{right_i}} \right]
\]

where \( n_{left_i} \) corresponds to the product of the order of all matrices before the \( i^{th} \) matrix of the tensor product term, \( n_{right_i} \) corresponds to the product of the order of all matrices after the \( i^{th} \) matrix of the tensor product term, \( n_{left_1} \) (particular case: \( n_{left_1} = 1 \)) and \( n_{right_N} \) (particular case: \( n_{right_N} = 1 \)).

Hence, the Shuffle algorithm consists in multiplying successively a vector by each normal factor. More precisely, vector \( v \) is multiplied by the first normal factor, then the resulting vector is multiplied by the next normal factor and so on until the last factor. In fact, the multiplication of a vector \( v \) by the \( i^{th} \) normal factor corresponds to shuffle the elements of \( v \) in order to assemble \( n_{left_i} \times n_{right_i} \) vectors of size \( n_i \) and multiply them by matrix \( Q^{(i)} \). Therefore, assuming that matrix \( Q^{(i)} \) is stored as a sparse matrix, the number of multiplications needed to multiply a vector by the \( i^{th} \) normal factor is:

\[
n_{left_i} \times n_{right_i} \times n_{left_i}
\]

where \( n_{z_i} \) corresponds to the number of nonzero elements of the \( i^{th} \) matrix of the tensor product term (\( Q^{(i)} \)). Considering the number of multiplications to all normal factors of a tensor product term, we obtain [8]:

\[
\prod_{i=1}^{N} n_i \times \sum_{i=1}^{N} \frac{n_{z_i}}{n_i}
\]

3.2 Slice Algorithm

Slice is an alternative algorithm to perform the vector-descriptor product not based only on the decomposition of a tensor product in the ordinary product of normal factors.
property (equation 4), but also applies a very basic property, the \textit{Additive Decomposition} \cite{8}. This property simply states that a tensor product term can be described by a sum of unitary matrices\(^2\):
\[
Q^{(1)} \otimes \ldots \otimes Q^{(N)} = \sum_{i_1=1}^{n_1} \ldots \sum_{i_N=1}^{n_N} \sum_{j_1=1}^{n_1} \ldots \sum_{j_N=1}^{n_N} \left( \hat{q}^{(1)}_{(i_1,j_1)} \otimes \ldots \otimes \hat{q}^{(N)}_{(i_N,j_N)} \right) \tag{8}
\]

where \(\hat{q}^{(k)}_{(i,j)}\) is an unitary matrix of order \(n_k\) in which the element in row \(i\) and column \(j\) is equal to element \((i, j)\) of the matrix \(Q^{(k)}\).

Obviously, the application of such property over a tensor product with fully dense matrices results in a catastrophic number of \(\prod_{i=1}^{N} (n_i)^2\) unitary matrix terms, but the number of terms is considerably reduced for sparse matrices. In fact, there is one unitary matrix to each possible combination of one nonzero element from each matrix. We may define \(\theta(1 \ldots N)\) as the set of all possible combinations of nonzero elements of the matrices from \(Q^{(1)}\) to \(Q^{(N)}\). Therefore, the cardinality of \(\theta(1 \ldots N)\), and consequently the number of unitary matrices to decompose a tensor product term, is given by \(\prod_{i=1}^{N} n_i\).

Generically evaluating the unitary matrices from equation 8, the sole nonzero element appears in the tensor coordinates \((i_1, j_1)\) for the outermost block, coordinates \((i_2, j_2)\) for the next inner block, and so on until the coordinates \((i_N, j_N)\) for the innermost block. By the own definition of the tensor product, the value of an element is \(\prod_{k=1}^{N} \hat{q}^{(k)}_{(i_k,j_k)}\), where \(\hat{q}^{(k)}_{(i_k,j_k)}\) is the element in row \(i\) and column \(j\) of matrix \(Q^{(k)}\). For such unitary matrices, we use the following notation:
\[
\hat{q}^{(1 \ldots N)}_{i_1 \ldots i_N, j_1 \ldots j_N} = q^{(1)}_{(i_1,j_1)} \otimes \ldots \otimes q^{(N)}_{(i_N,j_N)} \tag{9}
\]

The pure application of the Additive Decomposition property corresponds to generate a single equivalent sparse matrix to the tensor product term. For many cases, it may result in a too large number of elements. It is precisely to cope with this problem that the Shuffle algorithm was proposed. However, the manipulation of considerably sparse tensor product terms like this is somewhat awkward, since a decomposition in \(N\) normal factors may be a too large effort to multiply very few resulting elements.

The basic principle of the Slice algorithm is to handle the tensor product term in two distinct parts. The Additive Decomposition property is applied to all first \(N - 1\) matrices, generating \(\prod_{i=1}^{N-1} n_i\) very sparse terms which are multiplied (tensor product) by the last matrix, \(i.e.:\)
\[
Q^{(1)} \otimes \ldots \otimes Q^{(N)} = \sum_{\forall i_1, \ldots, i_{N-1}, j_{N-1}}^{\in \theta(1 \ldots N-1)} \hat{Q}^{(1 \ldots N-1)}_{i_1, \ldots, i_{N-1}, j_{N-1}} \otimes Q^{(N)} \tag{10}
\]

Therefore, the Slice algorithm consists in dealing with \(N - 1\) matrices as a very sparse structure, and dealing with the last matrix as the Shuffle approach did. The multiplication of a vector \(v\) by the tensor product term (equation 3) using the Slice algorithm can be rewritten as:
\[
 v \times \left[ \sum_{\forall i_1, \ldots, i_{N-1}, j_{N-1}}^{\in \theta(1 \ldots N-1)} \hat{Q}^{(1 \ldots N-1)}_{i_1, \ldots, i_{N-1}, j_{N-1}} \otimes Q^{(N)} \right] \tag{11}
\]

\(^2\)A unitary matrix is a matrix in which there is only one nonzero element.
Applying the distributive property, equation 11 can be rewritten as:

\[
\sum_{\forall i_1,\ldots,i_{N-1},j_1,\ldots,j_{N-1} \in \theta(1\ldots N-1)} v \times \left( \mathcal{Q}^{(1\ldots N-1)}_{i_1,\ldots,i_{N-1},j_1,\ldots,j_{N-1}} \otimes \mathcal{Q}^{(N)} \right)
\]  

(12)

We call each term of the previous equation as Additive Unitary Normal Factor, since it is composed of an unitary matrix times a standart normal factor. The decomposition in normal factors applied to each additive unitary normal factor of equation 12 results in:

\[
v \times \left[ \left( \mathcal{Q}^{(1\ldots N-1)}_{i_1,\ldots,i_{N-1},j_1,\ldots,j_{N-1}} \otimes \mathcal{I}_{n_N} \right) \times \left( \mathcal{I}_{\text{left}_N} \otimes \mathcal{Q}^{(N)} \right) \right]
\]  

(13)

It is important to notice that the first multiplication takes only \(n_N\) elements of vector \(v\) and it corresponds to the product of this sliced vector (called \(v_s\)) by the single scalar which is the nonzero element of matrix \(\mathcal{Q}^{(1\ldots N-1)}_{i_1,\ldots,i_{N-1},j_1,\ldots,j_{N-1}}\). The resulting vector, called \(v'_s\), must then be multiplied only once by matrix \(\mathcal{Q}^{(N)}\), since all other positions of the intermediate vector (except those in \(v'_s\)) are zero.

The application of the Slice algorithm must generate the nonzero element (\(c\)) of matrix \(\mathcal{Q}^{(1\ldots N-1)}_{i_1,\ldots,i_{N-1},j_1,\ldots,j_{N-1}}\). Hence, it must pick a slice of vector \(v\) (called \(v_s\)) according to the row position of element \(c\), and multiply all elements of \(v_s\) by \(c\). In fact, this multiplication by a scalar corresponds to the first multiplication by a normal factor of equation 13. The resulting vector, called \(v'_s\), must be multiplied by the matrix \(\mathcal{Q}^{(N)}\) (second multiplication in equation 13), accumulating the result \(r_s\) in the positions of the resulting vector \(r\) corresponding to the column position of element \(c\).

The Slice algorithm (Algorithm 1) can be summarize for all Additive Unitary Normal Factors in the operation:

\[
r = v \times \left[ \left( \mathcal{Q}^{(1\ldots N-1)}_{i_1,\ldots,i_{N-1},j_1,\ldots,j_{N-1}} \otimes \mathcal{I}_{n_N} \right) \times \left( \mathcal{I}_{\text{left}_N} \otimes \mathcal{Q}^{(N)} \right) \right]
\]

Algorithm 1 Slice Algorithm

1. for all \(i_1,\ldots,i_{N-1},j_1,\ldots,j_{N-1} \in \theta(1\ldots N-1)\) do
2. \(c \leftarrow \prod_{k=1}^{N-1} q_{(i_k,j_k)}\)
3. slice \(v_s\) from \(v\) according to \(i_1,\ldots,i_{N-1}\)
4. \(v'_s \leftarrow c \times v_s\)
5. \(r_s \leftarrow v'_s \times \mathcal{Q}^{(N)}\)
6. add \(r_s\) to \(r\) according to \(j_1,\ldots,j_{N-1}\)
7. end for

The computational cost (number of needed multiplications) of the slice algorithm considers: the number of unitary matrices \(\prod_{i=1}^{N-1} n_{z_i}\); the cost to generate the nonzero element of each unitary matrix \((N-2)\); the cost to multiply it by each element of sliced vector \(v_s\) \((n_N)\); and the cost to multiply \(v_s\) by the last matrix \(\mathcal{Q}^{(N)}\) \((nz_N)\), i.e.:

\[
\prod_{i=1}^{N-1} n_{z_i} \times \left( (N-2) + n_N + nz_N \right)
\]  

(14)
4 Numerical Analysis

In order to analyze the performance of both Shuffle and Slice algorithms, two different sets of models were considered. The first set of models describes a two-classes mixed finite capacity queueing network model (Figure 1).

![Figure 1: Mixed Queueing Network model](image)

For this model, customers of the first class will act as an open system visiting all queues, and the customers of the second class will act as a closed system visiting only the first three queues. In this model, all queues have only one server available and the customers of class 1 have priority over the customers of class 2. Such model was modeled using the SPN formalism and it was split in 8 subnets ($N = 8$) with 9 synchronized transitions ($E = 9$).

The second set of models describes a parallel implementation of a master/slave algorithm developed by SAN formalism. This model was introduced in [2] and, considering an implementation with $S$ slave nodes, it has $S + 2$ automata ($N = S + 2$) and $3 + 2 \times S$ synchronizing events ($E = 3 + 2 \times S$).

The numerical results for this section were obtained on a 2.8 GHz Pentium IV Xeon under Linux operating system with 2 GBytes of memory. The actual PEPS 2003 implementation [4] was used to obtain the Shuffle algorithm results and a prototype implementation was used to obtain the Slice algorithm results.

4.1 Shuffle and Slice Comparison

The first set of experiments is conducted for the two examples using both algorithms (columns Shuffle and Slice). For each option we compute the number of multiplications performed (computational cost - c.c.), and the time to execute a complete multiplication in seconds (time). For the Mixed Queue Network model (Mixed QN), we consider all queues with the same capacity ($K$) assuming the values $K = 3..7$. For the Parallel Implementation model (Parallel) described in [2], we assign the number of slaves ($S$) with the values $S = 3..7$. For all models, we also measured the memory needs to store the descriptor in KBytes, which is indicated in column mem$^3$ (see Table 2).

The number of multiplications needed for the Slice algorithm (equation 14) is less significant than the number needed in the Shuffle algorithm (equation 7). Even though the time spent in the Slice algorithm is still better than Shuffle one, the gains are slightly less significant than the computational cost gain. This happens probably due to a more optimized treatment of the function evaluations in the Shuffle algorithm.

---

$^3$Obviously, the memory needs for the Shuffle and Slice approach are equal, since the choice of algorithm does not interfere with the descriptor structure.
### Table 2: Shuffle and Slice algorithms comparison

<table>
<thead>
<tr>
<th></th>
<th>Shuffle</th>
<th></th>
<th></th>
<th>Slice</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>c.c.</td>
<td>time</td>
<td>mem.</td>
<td>c.c.</td>
<td>time</td>
<td>mem.</td>
</tr>
<tr>
<td>$K = 3$</td>
<td>$9.14 \times 10^6$</td>
<td>0.16</td>
<td>4</td>
<td>$2.59 \times 10^6$</td>
<td>0.05</td>
<td>4</td>
</tr>
<tr>
<td>$K = 4$</td>
<td>$5.53 \times 10^7$</td>
<td>0.87</td>
<td>5</td>
<td>$1.58 \times 10^7$</td>
<td>0.28</td>
<td>5</td>
</tr>
<tr>
<td>$K = 5$</td>
<td>$2.40 \times 10^8$</td>
<td>3.77</td>
<td>6</td>
<td>$6.86 \times 10^7$</td>
<td>1.19</td>
<td>6</td>
</tr>
<tr>
<td>$K = 6$</td>
<td>$8.30 \times 10^8$</td>
<td>12.86</td>
<td>6</td>
<td>$2.36 \times 10^8$</td>
<td>4.06</td>
<td>6</td>
</tr>
<tr>
<td>$K = 7$</td>
<td>$2.43 \times 10^9$</td>
<td>47.31</td>
<td>7</td>
<td>$6.85 \times 10^8$</td>
<td>14.82</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Shuffle</th>
<th></th>
<th></th>
<th>Slice</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>c.c.</td>
<td>time</td>
<td>mem.</td>
<td>c.c.</td>
<td>time</td>
<td>mem.</td>
</tr>
<tr>
<td>$S = 3$</td>
<td>$2.43 \times 10^6$</td>
<td>&lt; 0.01</td>
<td>9</td>
<td>$6.59 \times 10^4$</td>
<td>&lt; 0.01</td>
<td>9</td>
</tr>
<tr>
<td>$S = 4$</td>
<td>$1.10 \times 10^6$</td>
<td>0.02</td>
<td>12</td>
<td>$2.61 \times 10^4$</td>
<td>&lt; 0.01</td>
<td>12</td>
</tr>
<tr>
<td>$S = 5$</td>
<td>$4.67 \times 10^6$</td>
<td>0.09</td>
<td>15</td>
<td>$9.97 \times 10^4$</td>
<td>0.02</td>
<td>15</td>
</tr>
<tr>
<td>$S = 6$</td>
<td>$1.88 \times 10^6$</td>
<td>0.33</td>
<td>18</td>
<td>$3.71 \times 10^4$</td>
<td>0.07</td>
<td>18</td>
</tr>
<tr>
<td>$S = 7$</td>
<td>$7.31 \times 10^6$</td>
<td>1.14</td>
<td>21</td>
<td>$1.35 \times 10^4$</td>
<td>0.23</td>
<td>21</td>
</tr>
</tbody>
</table>

#### 4.2 Slice Algorithm Optimizations

The second set of experiments is conducted over the Mixed Queue Network example assuming all queues, but the last one, with the same capacity ($K = 4$). The capacity of the last queue ($K_5$) is tested with values 3, 4, 5, 6, and 7. Figure 2 shows a table with the numeric results obtained for these experiments and a plot of the time spent in both approaches.

![Figure 2: Experiments on Slice Optimization for Mixed Queue Network model](image-url)
Observing equations 7 and 14, it is possible to notice that, unlike the cost of the Shuffle algorithm, the cost of the Slice algorithm is less dependent on the order of the last matrix. This can be verified by the results in Figure 2, since both Slice and Shuffle curves have clearly different behaviors.

![Figure 2: Comparison of Slice and Shuffle algorithms](image)

<table>
<thead>
<tr>
<th>S</th>
<th>Non Ordered</th>
<th>Reordered</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>c.c.</td>
<td>time</td>
</tr>
<tr>
<td>3</td>
<td>$6.59 \times 10^3$</td>
<td>$&lt; 0.01$</td>
</tr>
<tr>
<td>4</td>
<td>$2.61 \times 10^4$</td>
<td>$&lt; 0.01$</td>
</tr>
<tr>
<td>5</td>
<td>$9.97 \times 10^5$</td>
<td>$0.02$</td>
</tr>
<tr>
<td>6</td>
<td>$3.71 \times 10^6$</td>
<td>$0.07$</td>
</tr>
<tr>
<td>7</td>
<td>$1.35 \times 10^7$</td>
<td>$0.23$</td>
</tr>
</tbody>
</table>

Figure 3: Experiments on Slice Optimization for Parallel Implementation model

The last set of experiments (Figure 3) shows the effect of automata reordering for the Parallel Implementation model. This model has one very large automaton (40 states) and all other automata with only 3 states. For these experiments, only the results of the Slice algorithm are indicated. The left hand side columns (Non Ordered) indicate the results obtained for the example with the larger automaton appearing at the beginning. The right hand side columns (Reordered) indicate the results obtained putting the largest automaton as the last one. The results show clearly the improvements in the number of multiplications as well as in the time spent. Such encouraging result suggests that many other optimizations could still be found to the Slice algorithm.

It is important to notice that an analysis of the functional evaluations for the Slice algorithm may reveal further optimizations, but as said in the introduction such analysis is out of the scope of this paper.

5 Conclusion

This paper proposes a different way to perform vector-descriptor product. The new Slice algorithm has shown a better overall performance than the traditional Shuffle algorithm for all examples tested. In fact, the Shuffle algorithm would only be more efficient for quite particular cases in which the descriptor matrices would be nearly full. Even though we could imagine such tensor products (with only nearly full matrices), we were not able to generate a real model with such characteristics. It seems that real case models have naturally sparse matrices. The local part of a descriptor is naturally very sparse due to the tensor sum structure. The synchronizing primitives are mostly
used to describe exceptional behaviors, therefore it lets the synchronizing part of the descriptor also quite sparse.

As a matter of fact, the Slice algorithm seems to offer a good trade-off between the unique sparse matrix approach used for straightforward Markov chains and the pure tensor approach of the Shuffle algorithm. It is much more memory efficient than the unique sparse matrix approach, and it would only be slower than the Shuffle algorithm in hypothetical models with nearly full matrices. However, even for those hypothetical models, the Slice approach may be used for some terms of the descriptor. Such hybrid approach could analyze which algorithm should be used to each one of the tensor product terms of the descriptor.

Besides the immediate future works to develop further experiments with the Slice algorithm already mentioned in the previous section, we may also foresee studies concerning parallel implementations. The prototyped parallel implementation of the Shuffle algorithm [3] has already shown consistent gains to solve particularly slow SAN models. Nevertheless, the Shuffle algorithm parallelization suffers an important limitation that consists in the passing of a whole tensor product term to each parallel node. This is a problem since all nodes must compute multiplications of the whole vector \( v \) by a tensor product term that usually has nonzero elements in many positions.

The Slice algorithm can offer a more effective parallelization since its Additive Unitary Normal Factors only affect few positions of vector \( v \). A parallel node could receive only similar terms and, therefore, not handle the whole vector \( v \). This can be specially interesting for parallel machines with nodes with few memory resources.

Concentrating back in the sequential implementation, our first results with the Slice algorithm prototype were very encouraging, but we expect to have many improvements to do before integrate this new algorithm in a new version of the PEPS software tool [4]. As we said before, this paper is just a first step for this new approach and much numerical studies have to be done. However, the current version of the Slice algorithm already shows better results than Shuffle.

References


