Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática
Programa de Pós-Graduação em Ciência da Computação

Efficient Vector-Descriptor Product Exploiting Time-Memory Trade-offs

Ricardo M. Czekster, Paulo Fernandes, Thais Webber

Relatório Técnico N^{o} 057

Porto Alegre, Abril de 2010

Abstract

The description of large state spaces through stochastic structured modeling formalisms like stochastic Petri nets, stochastic automata networks and performance evaluation process algebra is usually made representing the infinitesimal generator of the underlying Markov chain as a Kronecker descriptor instead of a single large sparse matrix. The most known algorithms used to compute iterative solutions of such structured models are: the pure sparse solution approach, an algorithm that can be very time efficient, and almost always memory prohibitive; the Shuffle algorithm which performs the product of a descriptor by a probability vector with a very impressive memory efficiency; and a recent new option that offers a trade-off between time and memory savings, the Split algorithm. This paper presents a full comparison of these algorithms solving some examples of structured Kronecker represented models in order to numerically illustrate the gains achieved considering each model characteristics.

1 Introduction

Performance evaluation of large modern systems usually becomes a challenging problem due to the complexity involved in describing and solving the models for such systems. Several solution techniques are available in the literature but one of the most commonly used technique is the state-based modeling approach and numerical evaluation of transient and stationary distributions.

However, modeling large complex systems with a state-based approach often requires a compact representation. Since complex systems are normally composed by many components, structured formalisms introduce the possibility of describing more than one irreducible component, with interactions among components and individual behavior. Markovian structured formalisms like Stochastic Petri nets (SPN) [1], Stochastic Automata Networks (SAN) [24] and Performance Evaluation Process Algebra (PEPA) [19] offer sophisticated storage and manipulation schemes to handle the infinitesimal generator for the underlying continuous-time Markov chain when calculating the numerical solution.

Among other options, *e.g.*, [8], many formalisms use classical and generalized tensor (Kronecker) algebra [2, 11, 17] as a very effective way to store quite large and complex models that are virtually impossible to be dealt by traditional approaches *e.g.*, sparse matrices [26]. The basic principle of tensor representations is to take advantage of the structural information already used in the model state-based description. In fact, the components behavior can be expressed by individ-

ual transition matrices and tensor operators among them. Such representation of an infinitesimal generator by a sum of tensor products of matrices expressing the behavior of components (subsystems) of a larger system is called *descriptor* [17]. The model storage using a descriptor undeniably reduces the memory requirements [7, 23], but it often presents slower executions to obtain stationary or transient solutions. Nevertheless, the use of descriptors is justifiable, since for many large problems it is the only option of solution. This fact is illustrated by the definition of tensor format for structured formalisms like SPN [13] and PEPA [20], and not only to SAN where the descriptors were originally proposed.

The numerical algorithms known to compute the exact solution of large state spaces in a tensor format are usually iterative and based on the vector-descriptor product (VDP), but there is always a trade-off between memory constraints and CPU processing costs. In fact, there are two classical solutions that can be applied to all types of descriptors:

- to deal with the descriptor as a singular large sparse matrix [1], which has high memory costs, but it has low CPU cost (when the memory costs are not prohibitive); or
- to use the *Shuffle* algorithm [16], which has low memory costs, but demands a higher CPU cost for many practical cases.

Two other options can be applied only to descriptors without functional dependencies among components, *i.e.*, descriptors that use classical, instead of generalized tensor algebra. These options are:

- to use of canonical matrix diagrams [22], which bets on clever data structures to hierarchically represent each component transitions, and may skip entirely the Kronecker representation passing from the the SPN expression to matrix diagrams. Such solution is quite dependent of the choice of components, which are called SPN partitions, but it delivers a very efficient solution to some practical cases given a good choice of components;
- to use the flexible hybrid vector-descriptor algorithm called *Split* [27, 10], which is a rather recent approach currently applied to a subset of the SAN formalism where the interaction between components is limited to synchronized events, *i.e.*, there is no functional rates or probabilities in the model. Once again, like the canonical matrix diagrams approach, the efficiency of the Split algorithm depends on choices of the order of components and a cut-parameter.

While the canonical matrix diagrams approach seems difficult to adapt to deal with functional dependencies, the Split algorithm approach seems more likely to be adapted in the future. However, it is necessary to improve its basic performance with a good choice of order of components and cut-parameter. Therefore, this paper focuses on the analysis of efficiency of the Split algorithm with the proposal of an initial heuristic to achieve a better algorithmically performance presenting slight modifications in its solution core when applied to a complete model. Specifically, we propose and analyze the benefits of different permutation of components to each term of the descriptor.

This paper is organized as follows. Section 2 presents the basic definitions of the options to multiply a vector by a descriptor with special emphasis on the Split algorithm. Section 3 has the main contribution of this paper by proposing an initial heuristic to achieve the better performance of the Split algorithm and the flexible algorithm application by using matrix permutations in each tensor product of the descriptor. Section 4 presents classes of descriptors and the numerical results showing the increase of performance achieve for Split in comparison with other approaches. Finally, the conclusion emphasizes this paper contribution towards an optimized vector-descriptor product approach and draws possible future works to evolve the solution of different structured models including functional dependencies, *i.e.*, to deal with generalized tensor algebra descriptors.

2 Vector-descriptor product

A descriptor for a continuous-time model[1] with N components is a sum of tensor products with N matrices each. The number of tensor product terms in a descriptor is explained slightly differently according to the formalism, but it can be explained in general lines as one single tensor sum describing all transitions that are independent to each component (transitions intra-partition in SPN, or local transitions in SAN and PEPA), plus a pair of tensor product terms to each possible interaction among components (transitions inter-partitions in SPN, synchronizing events in SAN, or cooperations in PEPA).

Hence, assuming a system with N components and E interactions among components, a descriptor is a tensor sum ($\oplus$) term, plus $2E$ tensor products ($\otimes$) terms all off them composed by N matrices (Equation 1).

[1] In the context of this paper only continuous-time model will be considered, since the formulation of discrete-time model is quite different and much more rare. Nevertheless, the reader may find interesting material on discrete-time tensor representation in [5, 25].

$$Q = \left(\bigoplus_{i=1}^{N} Q_{ind}^{(i)} \right) + \sum_{j=1}^{E} \left[\bigotimes_{i=1}^{N} \left(Q_{int_j^+}^{(i)} \right) + \bigotimes_{i=1}^{N} \left(Q_{int_j^-}^{(i)} \right) \right] \quad (1)$$

In this equation:

- $Q_{ind}^{(i)}$ represents the matrix with the rates and diagonal adjustment of the independent transitions of i-th component;

- $Q_{int_j^+}^{(i)}$ represents the matrix with the rates of the j-th interaction among components for the i-th component;

- $Q_{int_j^+}^{(i)}$ represents the matrix with the diagonal adjustment of the j-th interaction among components for the i-th component.

Assuming that the number of states in the i-th component is n_i, the descriptor Q is equivalent to the infinitesimal generator of a Markov chain with $\prod_{i=1}^{N} n_i$ states, which is traditionally represent by a single square matrix of order $\prod_{i=1}^{N} n_i$. However, the order of each matrix in the descriptor (Equation 1) will be equal the number of states of its corresponding component (n_i). Therefore, for large models the use of a descriptor often represents a huge memory saving.

Observing the basic descriptor equation (Eq. 1) it is possible to notice that a descriptor is actually a sum of tensor product terms that can be considered separately, *i.e.*, a tensor product of N matrices[2]. Each vector-descriptor product algorithm can be analyzed as the multiplication of a vector by $N + 2E$ tensor terms as depicted by Eq. 2, where a generic subscript j is used to represent all possible variations of matrices to be considered.

$$vQ = v \sum_{j=1}^{N+2E} \bigotimes_{i=1}^{N} Q_j^{(i)} = \sum_{j=1}^{N+2E} v \bigotimes_{i=1}^{N} Q_j^{(i)} \quad (2)$$

[2]Note that a tensor sum term can be decomposed into a sum of simple tensor terms [7].

2.1 Efficiency of Algorithms

The efficiency of each available option to perform a vector-descriptor product can be analyzed according to the memory usage and CPU demand to perform the multiplication of a vector v by a tensor product term of N matrices resulting in vector w, generically described as:

$$w = v \bigotimes_{i=1}^{N} Q^{(i)} \tag{3}$$

where the subscript j was abandoned to simplify the notation. To characterize the memory and CPU cost to each tensor product, we will define the characteristics of each matrix $Q^{(i)}$ composing the tensor term as being of order n_i and with nz_i nonzero elements.

In order to compute the memory demand of each option we will consider the storage of sparse matrices in Harwell-Boeing format [15] which is composed by three vectors. The first vector (aa) store each of the non-zero elements of the matrix ordered according to their row position. The second vector (ja) has the column position of each nonzero element. The third vector (ia) has the position where each row starts in the first and second vectors. Numerically, in a Harwell-Boeing format it will be necessary to store a Real numbers vector with as many elements as the number of non-zero elements, and two Integer numbers vectors, one with as many elements as the number of non-zeros and another with as many elements as the order of the matrix, *i.e.*, assuming ι Bytes for an Integer and ρ Bytes for a Real[3], the storage of a matrix with nz non-zeros and order n will take $nz \times \rho + (nz + n) \times \iota$ Bytes.

For the CPU demand estimation the number of required floating point multiplications will be used, despite the fact that actual CPU processors efficiency is no longer exactly bound by this number alone. Nevertheless, it is our belief that the number of floating point multiplications still remains the best indication of how much an algorithm will demand, since it usually is the most time demanding operation to be performed.

[3]In all 64-bit architectures the storage of an Integer values is made through a full 64-bit word ($\iota = 8$ Bytes), even though the eventual compiler works with a smaller precision. Analogously, a double precision Real is stored in two full 64-bit words ($\rho = 16$ Bytes).

2.2 Sparse Algorithm Efficiency

This algorithm is the simplest form of vector-descriptor multiplication since it consist in computing all nonzero elements of the matrix equivalent to the tensor product, and then multiply the elements of vector v by these elements.

Assuming a Harwell-Boeing format to store this equivalent matrix, the required amount of memory in Bytes will be:

$$\text{Mem(Sparse)} = (\rho + \iota) \times \prod_{i=1}^{N} nz_i + \iota \times \prod_{i=1}^{N} n_i \tag{4}$$

While the number of floating point multiplications will be:

$$\text{CPU(Sparse)} = \prod_{i=1}^{N} nz_i \tag{5}$$

2.3 Shuffle Algorithm Efficiency

The Shuffle algorithm keeps the matrices as they are, *i.e.*, it stores the N small matrices and then performs a cleaver shuffling of the vector v elements in order to perform several multiplications of sub-vectors v by each matrix $Q^{(i)}$.

Obviously, the memory efficiency of the Shuffle in comparison with the Sparse approach (Eq. 4) is enormous. Assuming Harwell-Boeing format to store the matrices, the amount of memory in Bytes required for the Shuffle algorithm is:

$$\text{Mem(Shuffle)} = (\rho + \iota) \times \sum_{i=1}^{N} nz_i + \iota \times \sum_{i=1}^{N} n_i \tag{6}$$

In contrast, the CPU efficiency of Shuffle is clearly disadvantageous for a general case. However, a simple optimization in the algorithm allows treat differently the matrices whether they are Identity matrices or not. In fact, the number of multiplications required for Shuffle will be composed by the product of all matrices order (product state space size) regardless of the matrices characteristics times the ratio between the number of non-zeros and the order of the non identity matrices, *i.e.*:

$$\text{CPU(Shuffle)} = \prod_{i=1}^{N} n_i \times \sum_{\substack{i=1 \\ \textit{iff } (Q^{(i)} \neq Id)}}^{N} \frac{nz_i}{n_i} \tag{7}$$

2.4 Split Algorithm Efficiency

The Split algorithm is a hybrid approach in between the Sparse and Shuffle algorithms. In fact, its basic principle is to split the tensor product term in two parts, applying a Sparse-like approach to the first (left-hand side) matrices and a Shuffle approach to the remaining (right-hand side) ones. The key to an efficient application of Split algorithm is to correctly choose a cut point splitting the matrices in each part. Still considering a tensor product term with N matrices (Eq. 3), γ denotes this cutting point, where $\gamma = 0$ means to apply a pure Shuffle approach to all matrices, $\gamma = N$ means to apply a pure Sparse approach whereas all other possibilities for γ from 1 to $N-1$ are actual hybrid cases of Split. Figure 1 depicts such choice.

$$\underbrace{(\mathcal{Q}^{(1)} \otimes \ldots \otimes \mathcal{Q}^{(N-2)})}_{\textit{Sparse-like part}} \otimes \underbrace{(\mathcal{Q}^{(N-1)} \otimes \mathcal{Q}^{(N)})}_{\textit{Shuffle part}}$$
$$\uparrow$$
$$\gamma$$

Figure 1: Split as a hybrid application of Sparse and Shuffle.

In order to better understand the efficiency improvement suggested in the next section, it is important to be aware of the basic steps in the Split algorithm. The application of the Split algorithm multiplying a vector v by a tensor product term $\otimes_{i=1}^{N} Q^{(i)}$ must be preceded by the choice of the cutting point γ and the computation of the Sparse-like part matrices. Each of these elements will define an Additive Unitary Normal Factor AUNF($\vec{i}$, $\vec{j}$, sc). In fact, all AUNF's will correspond to the Cartesian product of the matrices in the Sparse-like part. Therefore, there will be $\prod_{i=1}^{\gamma} nz_i$ AUNF's, each of them with:

- a scalar (sc) obtained by the multiplication of one nonzero element of each matrix in the Sparse-like part;
- an input slice of the vector v identified by the row coordinates ($\vec{i}$) of the of the nonzero elements from the Sparse-like matrices multiplied; and
- an output slice of the vector v identified by the column coordinates ($\vec{j}$) of the multiplied nonzero elements.

The actual application of the Split algorithm (Alg. 1), *i.e.*, the multiplication of v by $\otimes_{i=1}^{N} Q^{(i)}$ will correspond to a three step procedure applied to all precomputed

AUNFs. The first step fetch the elements of the input vector v according to the row coordinates expressed by $\vec{i}$ and it multiplies each of these elements by the scalar sc (lines 2, 3 and 4 in Alg. 1). The second step is a simple call of the Shuffle algorithm for the Shuffle part of tensor term (line 5 in Alg. 1). The last step is the accumulation of the Shuffle result in the output vector w according to the column coordinates expressed by $\vec{j}$ (lines 6, 7 and 8 in Alg. 1).

Algorithm 1 Split Algorithm $w = v \times \bigotimes_{i=1}^{N} Q^{(i)}$

1: **for all** AUNF($\vec{i}$, $\vec{j}$, sc) **do**
2: **for** $k = 1$ **to** $\prod_{i=\gamma+1}^{N} n_i$ **do**
3: $v_{in}[k] = sc \times v[\vec{i} + k]$;
4: **end for**
5: **Shuffle multiply** $v_{out} = v_{in} \bigotimes_{i=\gamma+1}^{N} Q^{(i)}$
6: **for** $k = 1$ **to** $\prod_{i=\gamma+1}^{N} n_i$ **do**
7: $w[\vec{j} + k] = w[\vec{j} + k] + v_{out}[k]$;
8: **end for**
9: **end for**

A Harwell-Boeing sparse structure is sufficient to store all AUNF's (row and column indications and scalar), and the same sparse structure can be used to store the small matrices of the Shuffle part. Therefore, the memory requirements for Split application will be denote as the simple requirements for each of the parts, *i.e.*:

$$\text{Mem(Split)} = \left[(\rho + \iota) \times \prod_{i=1}^{\gamma} nz_i + \iota \times \prod_{i=1}^{\gamma} n_i \right] + \left[(\rho + \iota) \times \sum_{i=\gamma+1}^{N} nz_i + \iota \times \sum_{i=\gamma+1}^{N} n_i \right] \quad (8)$$

The CPU demand, expressed as the number of floating point multiplications, corresponds to the application of the Shuffle multiplication for the right-hand side matrices plus the multiplications by scalar e when composing vector v_{in}, all this performed as many times as the number of AUNF's, *i.e.*:

$$\text{CPU(Split)} = \prod_{i=1}^{\gamma} nz_i \times \left(\prod_{i=\gamma+1}^{N} n_i + \left[\prod_{i=\gamma+1}^{N} n_i \times \sum_{\substack{i=1 \\ \textit{iff } (Q^{(i)} \neq Id)}}^{N} \frac{nz_i}{n_i} \right] \right) \quad (9)$$

3 Improving Split Efficiency

Split must balance together the computational cost in terms of multiplications and its memory needs. Consequently, the choice of a cutting point γ is not a trivial task because the number of nonzero elements in the Sparse part can demand a high computational cost in terms of memory for some models. The intrinsic characteristics related to the tensor product matrices (sparsity, identities, *etc.*) themselves can be used as parameters to analyze the adequate γ for each tensor product of a descriptor, considering also the possibility of changing the original lexicographical ordering of some matrices.

Therefore, it is of paramount importance to consider three aspects to improve Split efficiency:

- Each tensor product of the descriptor must be handled individually, *i.e.*, a good cut-parameter and order for a tensor product depends on its matrices characteristics, which is likely to be different for other tensor products in the descriptor;
- Re-ordering tensor products represents a very small computational cost [18], since it corresponding to a simple indirection in access the coordinates of the multiplying vector according to a permutation;
- Shuffle algorithm is absolutely efficient to handle Identity matrices, in fact it just skips their processing, since it is based on a multiplicative decomposition.

The permutation process changes the matrices order in a tensor product term to another configuration, enhancing the AUNF's generation and detecting permuted identity blocks in the underlying transition matrix easily treated by the Shuffle part. Moreover, the skipping of identity matrices in the Shuffle multiplication, clearly reduces the number of multiplications. On the contrary, is not adequate to include Identity matrices in the Sparse part of Split, since the generation of AUNF's will be augmented as well as the additional required memory, rarely with a significant gain in time.

Based on the pseudo-commutativity property [17], Split can rearrange the original lexicographically tensor term order (Eq. 3) as follows:

$$w = v \left[P_\sigma \times \left(\bigotimes_{i=1}^{N} \mathcal{Q}^{\sigma(i)} \right) \times P_\sigma^T \right] \tag{10}$$

where σ is a permutation on the interval $[1..N]$ for each tensor product term and $\sigma(i)$ returns the rank of the matrix $\mathcal{Q}^{(i)}$ in the order identified for the permutation σ. Moreover, P_σ is a permutation matrix and P_σ^T is the transposed matrix equal to P_σ.

The idea of using permutations in the Split method is to optimize the generation of AUNF's and reduce the shuffling part multiplications to a minimum necessary, even to zero if it is possible. In this case, all matrices in the Shuffle part of Split are Identities, and γ will be the index of the last non-Identity matrix being multiplied in the tensor product term.

The proposed heuristic to establish the cutting point γ, considering matrices permutations for each tensor product term, becomes quite straight forward by putting all identity matrices are put at right-hand side and the non-Identity matrices at left-hand side. It results consequently in an optimal reduction of the Split algorithm computational cost, since Identities are skipped in the Shuffle part.

However, if the memory available is restricted, it is possible to define the cutting point γ to include some of the non-Identity matrices in the Shuffle part in order to reduce the number of AUNF's. Obviously, this option only will be required when the number of AUNF's is too big to be stored in the memory, which is a rare case in models generated by structured formalisms. Considering that, depending on the model, we may need many iterations to calculate the final probability vector, the cutting point γ in each tensor term surely should be evaluated and balanced analyzing computational resources.

4 Numerical Analysis

This section presents the optimization achieved for different models. The results dictate the classes of models to consider the hybrid approach, uncovering and explaining the reasons why each model was solved faster than Shuffle and the original application of Split, *i.e.*, without the optimization proposed here.

The Split algorithm optimized is applied with different γ for each tensor product of the descriptor based on a set of characteristics related to them (refer to Section 3). Due to this, matrices permutations are used extensively, rearranging each tensor product to possess only non-Identity matrices in the Sparse part and identities in the structured part as splitting approach. In fact, for all examples, the memory usage due to the number of AUNF's was never a restriction.

The chosen execution platform is a Pentium machine with 3.2 GHz and 4 GB of RAM. The PEPS [6] tool was modified to comprise the Split method im-

plementation for original (stiff) and optimized versions. The PEPS modification included the generation and storage of the AUNF's and auxiliary data structures. The C/C++ compiler was the g++ version $4.0.4$ (GCC – *The GNU Compiler Collection*) with optimization options (-O3) and dynamic linkage. The results were produced running 50 sequential runs of fixed 25 iterations to compute the time per iteration information. These runs were statistically handled to obtain a 95% of confidence interval.

The five classes of examples presented here are the following SAN models in the descriptor format:

(i) Resource Sharing (RS) model [4] – a classical example of resource sharing with different network configurations since P is the number of processes (matrices with two states: *idle* and *occupied*) and R is the number of occupied resources (a matrix with $R+1$ states). The model descriptor presents $(2P)$ synchronizing events, totalizing $(4P)$ tensor products with $P+1$ matrices. The product state space is given by $[2^P \times (R+1)]$ states.

(ii) Dining Philosophers (DP) [12] – a model for the classical problem of K philosophers sitting at a circular table doing one of three things - *taking left fork*, *taking right fork* or *thinking*. The philosopher can reserve the fork on his immediate left or right waiting for eating with two available forks. To avoid deadlock is established an ordering to get the forks in the table, for each philosopher in the model. The model descriptor presents $(2K+2)$ synchronizing events, then $(4K+4)$ tensor products with K matrices. The product state space is formed by $[3^K]$ states.

(iii) Wireless ad hoc Networks (WN) model [14] – the model represents a chain of N mobile nodes in a Wireless Network running over the IEEE 802.11 standard for ad hoc networks where one node is the (*Source*) that generates packets as fast as the standard allows (two states: *idle* and *transmitting*). The packets are forwarded through the chain by the *Relay* nodes (three states: *idle*, *receiving* and *transmitting*), to the last node (*Sink*) which is the destination (two states: *idle* and *receiving*). The model descriptor is formed by a set of $(2N)$ tensor products and a tensor sum containing the local events information. The product state space is given by $[2^2 \times 3^{N-2}]$ states.

(iv) Master Slave (MS) architecture model [3] – a model for an evaluation of the master-slave parallel implementation of the Propagation algorithm considering asynchronous communication. One *Master* of three states (*transmitting*, *receiving* and *idle*), one huge *Buffer* of $K+1$ positions, and S slaves all with three states (*idle*, *processing* and *transmitting*). The model descriptor presents $(3S-3)$ synchronizing events. In this case, it is formed by a set of $(7S-8)$ tensor terms.

The model was extended to run different configurations of S slaves. The product state space is given by $[3^{(S+1)} \times (K+1)]$.

(v) Non-Uniform Memory Access (NUMA) model [9] – a model of processes running in NUMA processors for the Linux operating system. NUMA is a model for capturing the behavior of a single process under a multi-processed point of view, to analytically calculate the chances for a given processor to fail. The model descriptor is formed by a tensor sum and by a set of 64 tensor products of $R+1$ matrices. R is the number of processors. The product state space is given by $[(4^R+1) \times 6^R]$.

Table 1 presents a comparison between the Shuffle algorithm and the original (non-optimized) version of Split considering one iteration of vector descriptor multiplication for each model. It shows the gains obtained applying the same cutting point γ to all tensor terms without permutations. The choice of cutting point was made based on experimentations of all possible cutting points, and the results present the best choice according to time efficiency, *i.e.*, the faster cutting point choice. The columns in this table presents the CPU and memory needs expressed in seconds (s) and KBytes (KB), respectively, for Shuffle and Split. The column "#iter." indicates how many iterations were actually need to solve the model within a 10^{-10} tolerance using the Power method. The column "Time to Solve" indicates how much time will actually be necessary to reach the solution in each algorithm, *i.e.*, the product between the cost of one iteration and the required number of iterations. Finally, the last two columns indicate how much memory is necessary to execute Split algorithm and how much faster it was compared to the Shuffle solution.

The first interesting observation in the results in Table 1 is that Split algorithm is faster than Shuffle, even in its original and non-optimized version. In fact, this was expected, since in the worst case scenario Split performs exactly the same solution as Shuffle. However, we notice that Split original version is considerably demanding on memory resources, as may be seen for example in model (v) - NUMA (R=8) - where a reduction of ≈686.70 seconds per iteration costed ≈382 MB of additional memory.

Nevertheless, the gains in the overall solution fully justify the application of Split algorithm, since the memory usage was never too high. This fact is clearly indicated in the last row of Table 1 were it is indicated that the application of Shuffle to solve all models considered took ≈113.3 days, while the application of the Split original version took ≈9.94 times less (≈11.4 days) with an additional use of memory of less than ≈417 MB in the most demanding example, the model (ii) - DP (K=15).

Analogously as Table 1, Table 2 presents the same comparison between Shuffle algorithm results, but now it is compared to the results of the optimized version of Split proposed in this paper. The choice of cutting point γ was made before starting the iterative method analyzing the tensor term composition and available memory. This procedure have no relevant computational cost considering the gains we can achieve after running many iterations until the convergence for the solution.

One practical result is that the inclusion of ultra-sparse matrices[4] in the Sparse-like part do not result in a significant increasing of additional memory to solve some models, since it sometimes not even represent an increase in the number of AUNF's. This inclusion, or aggregation in the Sparse-like part often represents a really massive gain in time reducing the processing time required for solution from days to seconds.

This is an interesting point when analyzing models with synchronizing events among components because each event generates an ultra-sparse tensor product, *i.e.*, the combinations of nonzero elements of this matrices do not necessarily spent a substantial additional memory such as the examples (ii) and (iii). An aspect to be considered is the fact that we can have a trade-off between memory usage and time spent, *i.e.*, if one have lots of memory and wants performance, the γ could be more easily shifted to use the sparse solution approach, while if memory is limited the choice should allow more weight in the Shuffle part.

Models with interactions between two components are also interesting because one can aggregate at the Sparse-like part these components and be sure that the rest of matrices are identities. The additional memory is manageable and the processing time is faster than using the pure shuffling approach for each matrix as showed in the results for the models (i) and (v).

Models with huge matrices (at least one) and massive interactions (the same synchronizing event in more than one transition in a component) spent more memory since the AUNF's quantities are affected if this matrix is put inside the Sparse-like part as occurred in the model (iv). Moreover, the Shuffle part is also affected because this matrix dimension put for shuffling.

[4]We consider ultra-sparse matrices matrices with very few nonzero elements, *i.e.*, from a number of nonzero elements smaller than the order of the matrix.

Table 1: Comparison of all models total execution times and additional memory spent with Original Split.

Model		PSS	Shuffle		Split		#iter.	Time do Solve		Add. mem.	Time
			time(s)	mem.(KB)	time(s)	mem.(KB)		Shuffle	Split	(KB)	gain
(i)	RS (P=14;R=10)	180,224	0.31	96	0.20	10,239	119	≈0.61 min	≈0.40 min	10,143	1.55×
	RS (P=15;R=15)	524,288	0.68	266	0.34	280	153	≈1.73 min	≈0.87 min	14	1.99×
(ii)	DP (K=14)	4,782,969	5.17	2,342	3.06	141,163	933	≈1.34 h	≈47.58 min	138,821	1.69×
	DP (K=15)	14,348,907	16.64	7,015	9.82	423,491	1,002	≈4.63 h	≈2.73 h	416,476	1.69×
(iii)	WN (N=14)	2,125,764	2.18	1,041	0.50	12,917	78,029	≈1.96 days	≈10.84 h	11,876	4.36×
	WN (N=16)	19,131,876	22.67	9,347	5.07	116,252	93,126	≈24.43 days	≈5.46 days	106,905	4.47×
(iv)	MS (S=8;K=40)	807,003	1.03	398	0.63	41,630	3,094	≈53.07 min	≈32.49 min	41,232	1.63×
	MS (S=10;K=40)	7,263,027	11.34	3,553	6.65	373,100	2,696	≈8.49 h	≈4.98 h	369,547	1.71×
(v)	NUMA (R=6)	1,166,400	6.52	584	0.60	9,040	10,000	≈18.11 h	≈1.67 h	8,456	10.87×
	NUMA (R=8)	55,427,328	730.19	27,103	43.49	409,406	10,000	≈84.51 days	≈5.03 days	382,303	16.79×
							Total	**≈113.3 days**	**≈11.4 days**	max: **417 MB**	

Table 2: Comparison of all models total execution times and additional memory spent with Optimized Split.

Model		PSS	Shuffle		Split		#iter.	Time do Solve		Add. mem.	Time
			time(s)	mem.(KB)	time(s)	mem.(KB)		Shuffle	Split (opt.)	(KB)	gain
(i)	RS (P=14;R=10)	180,224	0.31	96	0.11	104	119	≈0.62 min	≈0.21 min	8	2.81×
	RS (P=15;R=15)	524,288	0.68	266	0.34	280	153	≈1.73 min	≈0.88 min	14	1.99×
(ii)	DP (K=14)	4,782,969	5.17	2,342	1.99	2,343	933	≈1.34 h	≈30.94 min	≈1	2.60×
	DP (K=15)	14,348,907	16.64	7,015	6.25	7,016	1,002	≈4.63 h	≈1.74 h	≈1	2.66×
(iii)	WN (N=14)	2,125,764	2.18	1,041	0.39	1,041	78,029	≈1.96 days	≈8.54 h	≈0	5.59×
	WN (N=16)	19,131,876	22.67	9,347	4.02	9,347	93,126	≈24.43 days	≈4.33 days	≈0	5.64×
(iv)	MS (S=8;K=40)	807,003	1.03	398	0.39	9,154	3,094	≈53.07 min	≈20.18 min	8,756	2.64×
	MS (S=10;K=40)	7,263,027	11.34	3,553	4.18	80,527	2,696	≈8.49 h	≈3.13 h	76,974	2.71×
(v)	NUMA (R=6)	1,166,400	6.52	584	0.38	589	10,000	≈18.11 h	≈1.05 h	5	17.16×
	NUMA (R=8)	55,427,328	730.19	27,103	25.70	27,112	10,000	≈84.51 days	≈2.97 days	9	28.41×
							Total	**≈112.3 days**	**≈8.1 days**	max:**77 MB**	

The Split algorithm in its optimized version provides impressive gains in comparison to Shuffle, mainly for models with a large number of synchronizing events such as model (v) or models with one interaction occurring among two or more components such as model (iii) which generates few AUNF's to be multiplied and lots of identities to be skipped. This contribution is related to classical descriptors being possible to extend to generalized tensor (Kronecker) products as well.

5 Conclusion

This paper presented the analysis of classical tensor products permutations using Split, *i.e.*, intrinsic matrices details such as type, total number of nonzero elements and computational cost in operations. These parameters opens the possibility of a thorough analysis of the related theoretical cost of the Split method and descriptor restructuring to balance memory and execution time. Moreover, the numerical solution could be enhanced with considerations about the impact of functional elements (with their particular dependencies) in the descriptor, as it is a new advance starting to emerge also for other formalisms [20, 21]. A similar work about these functional dependencies changed completely the performance of the Shuffle algorithm [17] when the terms took advantage of generalized tensor algebra properties. It is only natural to estimate that similar gains with functional dependencies analysis and possible matrices permutations could bring benefits to the Split algorithm as well.

The solution of classical descriptors, *i.e.*, having only constant rates, can be numerically interesting when matrices permutations are applied. The bottleneck for performing VDP is still bounded by the probability vector memory requirements. However, Split can explore, in extended versions, the application of more sophisticated manners to enhance the overall solution of complex Markovian models. A clear example is the use of sparse vector implementations, which accesses only reachable vector positions. Another approach to consider are data structures such as multivalued decision diagrams (MDD), which deals only with the reachable state space.

Additionally, to explore the lack of tensor term dependability introduced by Split, parallel and even distributed versions of the algorithm could be implemented in a near future. In this case, the major concern is the needed synchronization of the result probability vector at each iteration of VDP. For the sequential version, memory and time efficiency are dealt as a single demand, but, parallel implementations should consider other metrics such as the amount of memory needed,

the volume of exchanged data and other processing demands to evenly distribute tasks among machines. Obviously, these improvements require further inspections, with much deeper and complex analysis since neither the number of floating point multiplications, nor any other known index for that matter, seems to be a good estimation of processing time (at least not at the present moment).

Despite all these interesting future work, it is crystal clear that the gains achieved by the optimized version of Split are massive. Observing the time needed to solve all examples presented in this paper in Shuffle algorithm we verify that nearly three months are required (112.9 days exactly), while the application of the Split optimized version dropped it to a little more than a week (8.2 days actually) requiring only 77 MBytes of additional memory for the more demanding example. These results are even more impressive when considering the gains achieved in comparison with the original version of Split that resulted in significant reductions both in memory and CPU needs. In fact, this new version of Split upgrades the use of vector-descriptor product, and therefore, Kronecker-based solution of structured stochastic models, to a higher level of efficiency.

References

[1] M. Ajmone-Marsan, G. Conte, and G. Balbo. A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems. *ACM Transactions on Computer Systems*, 2(2):93–122, 1984.

[2] V. Amoia, G. D. Micheli, and M. Santomauro. Computer-Oriented Formulation of Transition-Rate Matrices via Kronecker Algebra. *IEEE Transactions on Reliability*, R-30(2):123–132, 1981.

[3] L. Baldo, L. Brenner, L. G. Fernandes, P. Fernandes, and A. Sales. Performance Models for Master/Slave Parallel Programs. *Electronic Notes In Theoretical Computer Science (ENTCS)*, 128(4):101–121, April 2005.

[4] A. Benoit, P. Fernandes, B. Plateau, and W. J. Stewart. On the benefits of using functional transitions and Kronecker algebra. *Performance Evaluation*, 58(4):367–390, 2004.

[5] L. Brenner. *Réseaux d'Automates Stochastiques: Analyse transitoire en temps continu et Algèbre tensorielle pour une sémantique en temps discret*. PhD thesis, Institut Polytechnique de Grenoble, France, 2009.

[6] L. Brenner, P. Fernandes, B. Plateau, and I. Sbeity. PEPS2007 - Stochastic Automata Networks Software Tool. In *Fourth International Conference on the Quantitative Evaluation of Systems (QEST'07)*, pages 163–164, Edinburgh, UK, 2007. IEEE Computer Society Press.

[7] L. Brenner, P. Fernandes, and A. Sales. The Need for and the Advantages of Generalized Tensor Algebra for Kronecker Structured Representations. *International Journal of Simulation: Systems, Science & Technology*, 6(3-4):52–60, February 2005.

[8] 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*, 13(3):203–222, 2000.

[9] R. Chanin, M. Corrêa, P. Fernandes, A. Sales, R. Scheer, and A. F. Zorzo. Analytical Modeling for Operating System Schedulers on NUMA Systems. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 151(3):131–149, 2006.

[10] R. M. Czekster, P. Fernandes, J. M. Vincent, and T. Webber. Split: a flexible and efficient algorithm to vector-descriptor product. In *Proceedings of the* 2^{nd} *International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS'07)*, page 83, Nantes, France, 2007. ACM.

[11] M. Davio. Kronecker Products and Shuffle Algebra. *IEEE Transactions on Computers*, C-30(2):116–125, 1981.

[12] E. Dijkstra. Hierarchical Ordering of Sequential Processes. *Acta Informatica*, 1:115–138, 1971.

[13] S. Donatelli. Superposed generalized stochastic Petri nets: definition and efficient solution. In R. Valette, editor, *Proceedings of the* 15^{th} *International Conference on Applications and Theory of Petri Nets*, pages 258–277. Springer-Verlag Heidelberg, 1994.

[14] F. L. Dotti, P. Fernandes, A. Sales, and O. M. Santos. Modular Analytical Performance Models for Ad Hoc Wireless Networks. In 3^{rd} *International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOpt'05)*, pages 164–173, Trentino, Italy, April 2005. IEEE Computer Society.

[15] I. Duff, R. Grimes, J. Lewis, and B. Poole. Sparse matrix test problems. *ACM Transactions on Mathematical Software (TOMS)*, 15(1):1–14, 1989.

[16] P. Fernandes. *Méthodes numériques pour la solution de systèmes Markoviens à grand espace d'états*. PhD thesis, Institut National Polytechnique de Grenoble, France, 1998.

[17] P. Fernandes, B. Plateau, and W. J. Stewart. Efficient descriptor-vector multiplication in Stochastic Automata Networks. *Journal of the ACM*, 45(3):381–414, 1998.

[18] P. Fernandes, B. Plateau, and W. J. Stewart. Optimizing tensor product computations in stochastic automata networks. *RAIRO, Operations Research*, 3:325–351, 1998.

[19] J. Hillston. *A compositional approach to performance modelling*. Cambridge University Press, New York, USA, 1996.

[20] J. Hillston and L. Kloul. An Efficient Kronecker Representation for PEPA models. In L. de Alfaro and S. Gilmore, editors, *Proceedings of the first joint PAPM-PROBMIV Workshop)*, pages 120–135, Aachen, Germany, September 2001. Springer-Verlag Heidelberg.

[21] J. Hillston and L. Kloul. Formal techniques for performance analysis: blending SAN and PEPA. *Formal Aspects of Computing*, 19(1):3–33, March 2007.

[22] A. S. Miner. Efficient solution of GSPNs using Canonical Matrix Diagrams. In 9^{th} *International Workshop on Petri Nets and Performance Models (PNPM'01)*, pages 101–110, Aachen, Germany, September 2001. IEEE Computer Society Press.

[23] A. S. Miner and G. Ciardo. Efficient Reachability Set Generation and Storage Using Decision Diagrams. In *Proceedings of the* 20^{th} *International Conference on Applications and Theory of Petri Nets (ICATPN)*, volume 1639 of *Lecture Notes in Computer Science (LNCS)*, pages 6–25, Williamsburg, VA, USA, June 1999. Springer-Verlag Heidelberg.

[24] B. Plateau and K. Atif. Stochastic Automata Networks for modelling parallel systems. *IEEE Transactions on Software Engineering*, 17(10):1093–1108, 1991.

[25] A. Sales. *Réseaux d'Automates Stochastiques: Génération de l'espace d'états atteignables et Multiplication vecteur-descripteur pour une sémantique en temps discret*. PhD thesis, Institut Polytechnique de Grenoble, France, 2009.

[26] W. J. Stewart. *Probability, Markov Chains, Queues, and Simulation*. Princeton University Press, USA, 2009.

[27] T. Webber. *Reducing the Impact of State Space Explosion in Stochastic Automata Networks*. PhD thesis, Pontifícia Universidade Católica do Rio Grande do Sul, Brazil, 2009.