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

PPGCC

Applying tensor term permutations to improve the numerical solution of structured Markovian models

Ricardo M. Czekster, Paulo Fernandes, Thais Webber

Relatório Técnico N^o 058

Porto Alegre, Abril de 2010

Abstract

Stochastic Automata Networks is a formalism that derives a Kronecker descriptor suitable for performance indices extraction. These descriptions use modeling primitives among their components or automata by capturing its operational semantics and allowing analysis to extract quantitative performance indices when subjected to numerical solution. The solution mechanism equipped in SAN uses Tensor Algebra properties (both classical and generalized) to multiply tensor product terms (*i.e.*, *a Markovian descriptor*) by a probability vector, studying stationary or transient regime. This operation is called Vector-Descriptor Product (VDP) and can be performed by three ways: sparsely (memory inefficient, time efficient under certain conditions), using the *Shuffle* algorithm (memory efficient, time inefficient depending on the model) or using the a hybrid approach, *i.e.*, the *Split* algorithm. The main contribution of Split was the proposition of an alternative method where reasonable memory increments are devised to accelerate the calculations per iteration in the core of VDP solution. Nevertheless, the main challenge of *Split* is directed to uncover the details on how to restructure classical and generalized tensor products to reduce computational costs per iteration. The present work focuses on these issues, observing the different manners to restructure the original (lexicographical) matrices orders. Also, it discusses the execution of the algorithm *Split* with matrices containing constant or functional rates, presenting results for a variety of models. For the observed cases, the best gain is verified when the matrices of the tensor products are reordered, treating the identity ones in the structured part and evaluating functional elements just once in the sparse part. When evaluating the functions only once in all VDP process, a conversion of generalized to classical descriptors takes place in execution time, with considerable gains in time for some classes of models. It was also observed that synchronization or communication activities between each module or automaton, in conjunction with the total number of functional dependencies, plays a crucial role in the performance. The present work is finalized through the identification of the most suitable models for the utilization of the Split algorithm, proposing restructurings in the Markovian descriptor that privilege sparsity and identity matrices to balance memory costs and execution time.

1 Introduction

The choice of a modeling formalism to depict a system is often based on the set of primitives available for use. The mapping of the system functionalities to an analytical model help modelers comprehend its particularities and relations through reasoning and also allowing the extraction of performance indices. A classic example of an unstructured formalism are traditional Markov Chains (MC) [17, 23], with a restrict yet very representative set of primitives (*i.e.*, states and transitions) to describe a model and numerically compute its stationary or transient measures. In MC, every state are viewed as a property of a system and each transition maps not only how the systems is interconnected but also at which frequency or rate it changes among one another. In a sense, the MC describe a very simple mechanism to convey communication among states and the amount of information that enables the transitions.

Despite the simplicity of MC to model numerous realities and calculate measures of interest, even small sized problems can derive an elevated amount of states to deal with, storing the transitions as a huge flat matrix [23]. For that matter, structured formalisms were built, intended to decompose monolithic systems in more manageable parts. The modular view of system added representation power to modelers and extended the general comprehension of the system to cope with more intricate relations among its entities or modules [3]. The adoption of such formalisms allowed more complex modeling primitives to be derived as well, mapping more complex interactions and communications. One important modeling primitive present particular to SAN are functional transitions, *i.e.*, transitions that are somehow dependent upon other entities states to be properly fired.

A non-comprehensive list of examples of such structured formalisms are Stochastic Automata Networks (SAN) [19, 20], Petri Nets (PN) [18, 1], Markovian Process Algebras (MPA) [12] and Performance Evaluation Process Algebra (PEPA) [13], to name a few. These modeling formalisms are constructed with different auxiliary inner structures to deal with the transition matrix. In PEPA, the matrix is stored and then solved using standard mathematical tools equipped with a Gaussian elimination procedures or other linear algebra mechanism to discover the permanence probabilities of each and every state. However, more sophisticated means to store the equivalent Markovian matrix that represents a system are available to the community particularly those that uses Kronecker algebra to access matrices elements. A whole area of research is devoted to the formalization of the properties and novel mechanisms to address Kronecker (or tensor structures) for model descriptions and compact storage [11]. It is classified as Classical

Tensor Algebra (CTA) when only constant rates are present in the model and as Generalized Tensor Algebra (GTA) when functional rates are in play.

Moreover, recent years witnessed the efforts for some structured formalisms to define its own Kronecker format [10, 14], respecting the set of rules firstly dictated by SAN, mainly for storage purposes [16]. A Kronecker descriptor is a representation of a system behavior through tensor product decomposition, informing how it operates locally or with synchronizations among other entities. The solution is numerically computed using a Vector-Descriptor Product (VDP) procedure that basically multiplies a probability vector by a non trivial structure, *i.e.*, a descriptor. The complex calculation of indices to access the inner tensor structures uses different sets of algorithms, specially designed to work with tensor representations. There are three major ways to perform VDP: using a standard *Sparse* [23] approach, using the traditional *Shuffle* [9, 11] or through a combination of both previous techniques, called *Split* [8].

The Split mechanism was initially developed to demonstrate the possibility to combine both sparse and Shuffle approach altogether defining flexible division points or *cut-parameters* for any set of tensor products composing a descriptor, balancing memory and numerical complexity. However, the previous work [8] did not take into account important implications such as the use of functional rates and how to combine it with permutations when considering Split as the core of the VDP method. This paper addresses these issues, analyzing where to best divide tensor products when GTA is present and the time-memory trade-offs to consider to accelerate each iteration.

This paper is organized as follows: Section 2 presents the background in Kronecker descriptors, Section 3 addresses available solution mechanisms mainly the Split algorithm definitions and computational cost. Section 4 discusses strategies to restructure classical and generalized tensor products when applying the Split approach with matrices permutations to accelerate iterations. The numerical results provided by classes of selected models are presented in Section 5.1 presenting a time-memory analysis to emphasize the gains observed. Conclusions and future works towards to a thorough analysis of possible method extensions are discussed in Section 6.

2 Kronecker Descriptors

The main advantage to use a Kronecker representation of a system is due to the memory savings. A Kronecker descriptor uses tensor algebra to access the elements of a potentially huge transition matrix that represents the Infinitesimal Generator (a more precise definition is given below) of a system and return valid performance indices. This representation is specially useful when the amounts of states of a system are massive and intractable and need to be decomposed in subsystems or independent entities. Besides SAN, equipped with Kronecker since its definition [19], other structured formalisms also provided such representations in its modeling primitives such as Superposed Generalized Stochastic Petri Nets (SGSPN) [10] and more recently, Performance Evaluation Process Algebra (PEPA) [14].

A Kronecker descriptor holds information about how the system operates locally and how it synchronizes activities. To model a system, one should decompose it in subcomponents or compartments having states, interconnected by transitions. Each transition can be defined by a set of events that marks if it will operate independently (local event) in conjunction with other parts or components (a synchronizing event). For the local events, tensor sums shall be used since they do not affect the firing of other elements whereas tensor products will be considered when more than one element of a system changes its local state. The choice of whether to model it with constant or functional rates in the transitions will impact the way to take in account the Kronecker descriptor. If only constant rates are defined in the transitions, Classical Tensor Algebra (CTA) will be used and for the case where functional rates are present, *i.e.*, rates with functions that inspect the state of other elements to be fired, Generalized Tensor Algebra (GTA) will be adopted. The reader interested in a broader discussion about descriptor translations, there are proofs of interchangeability between both algebras [5].

The entities independent behavior is represented by a tensor sum ($\bigoplus_g$) of local transition matrices containing the rates of local events. The synchronizations are represented by a set of tensor products ($\bigotimes_g$) of matrices containing the occurrence rates for them. Equation (1) demonstrates the algebraic expression for Kronecker descriptors $\tilde{\mathcal{Q}}$, where N indicates the number of entities (or components) of a larger system and E is the number of synchronizing events. For simplification purposes we suppressed the letter *g* present in the tensor sum and product operators, however, we are considering the transition rates containing constant or functional values, *i.e.*, a generalized descriptor.

$$\tilde{\mathcal{Q}} = \bigoplus_{i=1}^{N} \mathcal{Q}_l^{(i)} + \sum_{e=1}^{E} \left(\bigotimes_{i=1}^{N} \mathcal{Q}_{e^+}^{(i)} + \bigotimes_{i=1}^{N} \mathcal{Q}_{e^-}^{(i)} \right) \tag{1}$$

The equation shows how to construct the descriptor $\tilde{\mathcal{Q}}$ where $\mathcal{Q}_l$ are the matrices that represent the occurrence and the diagonal adjust for the local events rates, and $\mathcal{Q}_{e^+}$ and $\mathcal{Q}_{e^-}$ are, respectively, the matrices for the occurrence and diagonal adjust for synchronizing events rates within a system. Since tensor sums are decomposed in tensor products with identities called *normal factors*, the whole system can be considered as a sum of tensor products [11]. Equation (2) presents that the set $\mathcal{L}$ holds j tensor products of N matrices of types $\{l, e^+, e^-\}$ with maximum cardinality given by $|\mathcal{L}| = N + 2E$.

$$\tilde{\mathcal{Q}} = \sum_{j}^{|\mathcal{L}|} \bigotimes_{i=1}^{N} \mathcal{Q}_j^{(i)} \tag{2}$$

Table 1 illustrates a general representation of Kronecker Descriptors. The table is divided in two parts, divided by the local behavior showing the tensor products with identities and the synchronizing behavior, which is divided by the occurrence and the diagonal adjust for each event.

Table 1: General representation of Kronecker Descriptors.

$\sum$	N		$Q_l^{(1)} \otimes I_{n_2} \otimes \cdots \otimes I_{n_{N-1}} \otimes I_{n_N}$
			$I_{n_1} \otimes Q_l^{(2)} \otimes \cdots \otimes I_{n_{N-1}} \otimes I_{n_N}$
			$\vdots$
			$I_{n_1} \otimes I_{n_2} \otimes \cdots \otimes Q_l^{(N-1)} \otimes I_{n_N}$
			$I_{n_1} \otimes I_{n_2} \otimes \cdots \otimes I_{n_{N-1}} \otimes Q_l^{(N)}$
	$2E$	e^+	$Q_{e_1^+}^{(1)} \otimes Q_{e_1^+}^{(2)} \otimes \cdots \otimes Q_{e_1^+}^{(N-1)} \otimes Q_{e_1^+}^{(N)}$
			$\vdots$
			$Q_{e_E^+}^{(1)} \otimes Q_{e_E^+}^{(2)} \otimes \cdots \otimes Q_{e_E^+}^{(N-1)} \otimes Q_{e_E^+}^{(N)}$
		e^-	$Q_{e_1^-}^{(1)} \otimes Q_{e_1^-}^{(2)} \otimes \cdots \otimes Q_{e_1^-}^{(N-1)} \otimes Q_{e_1^-}^{(N)}$
			$\vdots$
			$Q_{e_E^-}^{(1)} \otimes Q_{e_E^-}^{(2)} \otimes \cdots \otimes Q_{e_E^-}^{(N-1)} \otimes Q_{e_E^-}^{(N)}$

Using tensor algebra enables the definition of the underlying transition matrix of a system in a compact manner. However, the calculation of the numerical solution is dependable of the definition of Vector-Descriptor Product (VDP) operations inside iterative methods. Next section will discuss how to perform VDP mentioning the known basic approaches, *i.e.* a Sparse based technique [6, 23], the Shuffle [11] and a special interest in the Split approach [8] flexibility, which is the focus of this paper.

3 Solution Mechanisms

Markov Chains are the base class of models which are in fact the inspiration for all other structured Markovian representations. Simply stating, a Markov Chain represents a system having only states, transitions and frequencies or rates of change among states. The solution of such representations can be done by multiplying a vector by a usually huge particular matrix, which is equivalent to the Infinitesimal Generator of a system [23]. Such matrix encapsulates the transition rates present in the model from all participating states of the system, ensuring that the main diagonal cells contains the negative sum of each row, except itself. This procedure will force each row to be zero summed, the main characteristic of an infinitesimal generator in practice. Then, this Vector-Matrix Product (VMP) runs an iterative solution method [21, 23] in order to find the convergent regime, also called stationarity. This regime can be reached by verifying if the residue computed on every two runs of the iterative method is smaller than an acceptable amount or tolerance accepted.

On the contrast, VDP methods multiply a probability vector $\pi^{(t)}$ by a not trivial structure, *i.e.*, a Kronecker product. Equation (3) defines the multiplication of $|\mathcal{L}|$ tensor products, meaning the entire descriptor $\tilde{\mathcal{Q}}$, in the iterative process until an acceptable vector $\pi^{(t+1)}$ containing the model solution. Note that the objective of solving a Kronecker descriptor is to discover how it operates or if stationary regime was achieved (or not) for further inspection and analysis.

$$\sum_{j=1}^{|\mathcal{L}|} \left(\pi^{(t)} \times \left[\bigotimes_{i=1}^{N} \mathcal{Q}_j^{(i)} \right] \right) = \pi^{(t+1)} \tag{3}$$

This section will emphasize in the solution mechanisms available to solve Kronecker descriptors produced by any formalism equipped with a valid tensor format for performance purposes, *i.e.*, defining an implicit infinitesimal generator. There are basically three different approaches to consider when performing VDP operations: the pure Sparse based approach that is commonly memory impeditive, the traditional memory efficient Shuffle algorithm, or through a combination of both aforementioned techniques, which is known as Split algorithm. This work will briefly describe the first two approaches and will focus on the latter approach, mainly when functional rates and permutations are in play. For further explanations on Sparse based techniques or the Shuffle method refer to the literature [9, 19, 22, 11, 3, 23].

The Sparse based algorithm is the most intuitive VDP method, however, it requires extremely large amounts of memory to compute the stationary probability vector since it stores the full flat transition matrix. This is the main cause why a more memory efficient manner was imperative to be devised, since modeling solution became impractical and for some cases, impossible. The alternative was to use the tensor algebra properties to jump within the tensor structure finding the needed elements for the multiplication, and that proved to use lots of indices calculation with minor effects to the storage needs. This technique was called Shuffle [9] because its main idea is to shuffle and access specific vector positions according to the tensor product matrices being multiplied. The algorithm was later extended to work with functional rates, cyclic function dependencies and matrices permutations, to perform fewer functional evaluations [11] achieving significant numerical results.

However, as the Shuffle algorithm proved to be useful when storing the flat matrix in a very efficient manner, the solution of large models were still a major research challenge. The process of finding the elements in tensor products leads the computation of indices to a point that nearly impairs the method as a whole depending on the descriptor. For this particular reason, the discovery of different VDP algorithms was a focus of research which eventually resulted in the proposition of the Split algorithm [8].

3.1 Split algorithm

The main idea of Split algorithm expressed in the Equation (4) is to divide each tensor product in two distinct but equally important parts (or sides) through a cut-parameter γ: a *sparse part*, where the Sparse algorithm operates, and a *structured part* where Shuffle algorithm is performed. In this paper we refer the division point as γ instead of the previous σ definition of the original Split [8].

$$\underbrace{\mathcal{Q}_j^{(1)} \otimes \ldots \otimes Q_j^{(N-2)}}_{\textit{sparse part}} \overset{\overset{\gamma}{\downarrow}}{\otimes} \underbrace{\mathcal{Q}_j^{(N-1)} \otimes Q_j^{(N)}}_{\textit{structured part}} \tag{4}$$

The initial inspiration for Split [8] was to somehow take advantage of the known *Additive Decomposition* property [9] combined with classical tensor products within the normal factors. For the sparse part, each non-zeroed element will be combined together to form a list of *Additive Unitary Normal Factors* (AUNFs), containing three basic information for each position (Algorithm 1): the scalar (s), the indices of the input vector ($base_{in}$) and the indices for the output vector ($base_{out}$). These factors can be viewed as precomputed elements that will help the procedure to accelerate each iteration, since the required indices are no longer needed to be computed if they remain in the sparse part. Fully sparse approaches can deal with an optimized generation of scalars [6], however, iterative procedures will repeat this computation if the scalars can not be stored in memory.

Algorithm 1 consists in computing the scalar element s of each AUNF in $\theta(1..\gamma)$ multiplying each non-zeroed element from each matrix of the first set of matrices (the sparse part) from $\mathcal{Q}_j^{(1)}$ until $\mathcal{Q}_j^{(\gamma)}$ (lines 4 to 8). According to the line elements used to generate s, a contiguous slice of the input vector υ called υ_{in} will be used to build a data structure to hold the information. The vector υ_{in} of size $nright_\gamma = \prod_{i=\gamma+1}^{N} n_i$, corresponding to the product of the dimensions of the matrices after the division point γ of the tensor product, is multiplied by the precomputed scalar s. Lines 9 and 11 multiply s to every position in υ, finishing the sparse part of the algorithm. The resulting vector υ_{in} is also used as the input for the structured part, *i.e.*, where Shuffle will be performed (lines 12 to 31) on the tensor product of the matrices belonging to the second part (from $\mathcal{Q}_j^{(\gamma+1)}$ to $\mathcal{Q}_j^{(N)}$). At the end of the structured part, the resulting υ vector is accumulated to π (lines 32 to 34). Line 16 shows a directive called **evaluate**, used for the evaluation of functional elements. It is worth noticing that Split was modified to work with GTA, *i.e.*, functional rates in transitions, the major distinction in comparison with the previous definition [8]. Along with the use of cleverly ordered permutations, that is the main algorithmic change and the fully expansion to work with the full set of primitives available to model and derive Kronecker representations.

The computational cost in terms of floating-point multiplications is presented on Equation (5), which shows the number of times needed to compute the AUNF list $(\gamma - 1)$, plus the number of multiplications of each scalar by each vector position υ_{in}. The equation is complete when the cost of multiplying the values from the input vector υ_{in} by the tensor product of the matrices belonging to the structured part is concatenated, as follows:

$$\left(\prod_{i=1}^{\gamma} nz_i\right) \times \left[(\gamma - 1) + \left(\prod_{i=\gamma+1}^{N} n_i\right) + \left(\prod_{i=\gamma+1}^{N} n_i \times \sum_{\substack{i=\gamma+1,\\ Q_j^{(i)} \neq Id}}^{N} \frac{nz_i}{n_i}\right)\right] \tag{5}$$

Algorithm 1 Split method: $\pi = \upsilon \times \otimes_{g\, i=1}^{N} \mathcal{Q}_j^{(i)}$

```
1: for all $i_1, \ldots, i_\gamma, j_1, \ldots, j_\gamma \in \theta(1..\gamma)$ do
2:     $s = 1$
3:     $base_{in} = base_{out} = 0$
4:     for all $k = 1, 2, \ldots, \gamma$ do
5:         $s = s \times q^{(k)}_{(i_k, j_k)}$
6:         $base_{in} = base_{in} + ((i_k - 1) \times nright_k)$
7:         $base_{out} = base_{out} + ((j_k - 1) \times nright_k)$
8:     end for
9:     for all $l = 0, 1, 2, \ldots, nright_\gamma - 1$ do
10:        $\upsilon_{in}[l] = \upsilon[base_{in} + l] \times s$
11:    end for
12:    for all $i = \gamma + 1, \ldots, N$ do
13:        $base = 0$
14:        for all $m = 0, 1, 2, \ldots, \frac{nleft_i}{nleft_\gamma} - 1$ do
15:            for all $j = 0, 1, 2, \ldots, nright_i$ do
16:                evaluate $Q^{(i)}(a^{(1)}_{m_1}, \ldots, a^{(i-1)}_{m_{i-1}})$
17:                $index = base + j$
18:                for all $l = 0, 1, 2, \ldots, n_i - 1$ do
19:                    $z_{in}[l] = \upsilon_{in}[index]$
20:                    $index = index + nright_i$
21:                end for
22:                $multiply\ \ z_{out} = z_{in} \times \mathcal{Q}_j^{(i)}$
23:                $index = base + j$
24:                for all $l = 0, 1, 2, \ldots, n_i - 1$ do
25:                    $\upsilon_{in}[index] = z_{out}[l]$
26:                    $index = index + nright_i$
27:                end for
28:            end for
29:            $base = base + (nright_i \times n_i)$
30:        end for
31:    end for
32:    for all $l = 0, 1, 2, \ldots, nright_\gamma - 1$ do
33:        $\pi[base_{out} + l] = \pi[base_{out} + l] + \upsilon_{in}[l]$
34:    end for
35: end for
```

There are a few optimizations to implement in VDP algorithms, responsible for the alteration of the theoretical computational cost presented in Equation (5). Ideally, generating normal factors in the structured parts with only identities is preferable, since the whole procedure will be skipped by the Shuffle algorithm (this is the main existing optimization for Shuffle, already discussed in other works [11, 6]). However, in this case, the memory should take special attention since it can use huge amounts, depending on the case, to save all necessary scalars and precomputed indices within the AUNF list per event. This improvement can change the Shuffle part contained at the Split algorithm in the matrices $\mathcal{Q}_j^{(\gamma+1)}$ to $\mathcal{Q}_j^{(N)}$. Case the matrix indexed by i in the algorithm ($\mathcal{Q}_j^{(i)}$) is not an identity, the cost of $\frac{nz_i}{n_i}$ multiplications is added.

Another valuable optimization concerns the precomputation of the non-zeroed elements through the utilization of an auxiliary data structure that will be continuously used in all iterations but calculated only once, at the beginning of the VDP procedure. These optimizations cause a significant reduction in the computational cost of Split. If this optimization is to be used, the number of floating-point multiplications is no longer defined by Equation (5) but by Equation (6):

$$\left(\prod_{i=1}^{\gamma} nz_i\right) \times \left[\left(\prod_{i=\gamma+1}^{N} n_i\right) + \left(\prod_{i=\gamma+1}^{N} n_i \times \sum_{\substack{i=\gamma+1,\\ Q_j^{(i)} \neq Id}}^{N} \frac{nz_i}{n_i}\right)\right] \tag{6}$$

One could also argue that the Split algorithm has two special cases (if $\gamma = N$, Sparse will be considered and if $\gamma = 0$, Shuffle will be performed). This is valid and it extrapolates the flexibility attained by the Split algorithm when dealing with Kronecker descriptors. The algorithm proved to be flexible enough to analyze available memory and tensor products formations while simply combining both approaches. Note that the intersection of both techniques in the VDP plays an important role when dealing with different descriptors characteristics (number of non-zeroed elements in each sparse matrix, number of identity matrices, number of tensor products, and so on). However, it introduces new challenges for solution, related to three major factors: (i) firstly, where exactly the division point for each tensor product should be set, (ii) secondly, how to take advantage of permutations or reordering in the original order of matrices to accelerate each iteration thus providing quicker results to modelers, and (iii) finally, how to organize the tensor product when functional rates are being used and how it would impact the memory-time trade-offs in the method. Next section will debate how to restructure tensor products for both CTA and GTA Kronecker descriptors taking advantage of the flexibility allowed by Split exploiting these details.

4 Tensor Product Restructuring

Kronecker descriptors are used to describe systems with simple and useful primitives enabling the composition of the system and consider its partitions or permissible divisions. The descriptor itself comprise a division, in this case, by the design of each event that constitutes a model, addressing different tensor algebra properties for each type, *i.e.*, tensor sums for local events and tensor products for synchronizations. However, the way on which the tensor product is originally formed can be changed simply altering the position of its matrices. This section discusses how to restructure generalized tensor products altering the original orders with permutations and the adequate strategies when solving tensor descriptors using the Split method.

The prior knowledge developed for the Shuffle algorithm [11] about the transformation of tensor products using permutations provides an indication on where to divide each tensor product after a restructuring took place. Due to the nature of Kronecker descriptors, in terms of matrices composition, permutations are mandatory to optimize operations in the Split algorithm. The permutation process changes the natural matrices order in a tensor term (initially the order of matrices is taken from the lexicographical order) to another configuration, enhancing the AUNFs (scalars) creation, reducing functions evaluations in generalized descriptors, and detecting permuted identity blocks in the underlying transition matrix.

The permutation procedure [11] uses a permutation matrix P_σ multiplied over the target tensor product, *i.e.*, the permutation matrix must be multiplied at left to redefine the position of matrices $\mathcal{Q}^{(i)}$ $(i = 1..N)$. To obtain the original ordering, the transposed (and inverse) matrix (correspondent to the permutation matrix), it must be multiplied at right. Specifically, the multiplication of the P_σ matrix at left of a tensor product, means the reordering of the matrix lines. Analogously, the multiplication of P_σ^T at right of the tensor product, means the reordering of the matrix columns. This is formally known as the pseudo-commutativity property as follows in the Equation (7):

$$\bigotimes_{i=1}^{N} \mathcal{Q}_j^{(i)} = P_{\sigma_j} \times \left[\bigotimes_{i=1}^{N} \mathcal{Q}_j^{(\sigma_j(i))} \right] \times P_{\sigma_j}^T \tag{7}$$

where σ_j is a permutation on the interval $[1..N]$. P_{σ_j} is the permutation matrix, *i.e.*, it is a matrix that has one nonzero element in each row and column, also equal to 1.

The parameter σ_j in this context corresponds to the establishment of a new order (or permutation) for a set of N matrices for the j-th tensor product term. Considering that $\sigma_j(i)$ returns the rank of the matrix $\mathcal{Q}_j^{(i)}$ in the order identified for the permutation σ_j, one denote σ_j^k as the matrix index placed in the rank k of order defined for the permutation σ_j (if $\sigma_j^k = i$, $\sigma_j(i) = k$). Rewriting the Equation (3), the iterative process including the VDP method with permutations is now given by Equation (8):

$$\sum_{j=1}^{|\mathcal{L}|} \left[\pi^{(t)} \times \left(P_{\sigma_j} \times \left[\bigotimes_{i=1}^{N} \mathcal{Q}_j^{(\sigma_j(i))} \right] \times P_{\sigma_j}^T \right) \right] = \pi^{(t+1)} \tag{8}$$

The Split algorithm when dealing with a classical descriptor has to define the division point γ to separate two groups of matrices. The idea of using permutations is to optimize the generation of AUNFs and reduce the shuffling part multiplications to a minimum necessary, even to zero if it is possible. It is a strategy of handling the classical descriptor in a nearly-full sparse-like approach without the storage of a huge sparse matrix but storing one aggregated matrix composed of the combination of sparse matrices put at left of γ. Hence, the number of floating point multiplications for the Split method with matrices permutations σ_j for j-th tensor product term is defined by the Equation (9):

$$\left(\prod_{i=1}^{\gamma} nz_{\sigma_j(i)} \right) \times \left[\left(\prod_{i=\gamma+1}^{N} n_{\sigma_j(i)} \right) + \left(\prod_{i=\gamma+1}^{N} n_{\sigma_j(i)} \times \sum_{\substack{i=\gamma+1, \\ Q_j^{(\sigma_j(i))} \neq Id}}^{N} \frac{nz_{\sigma_j(i)}}{n_{\sigma_j(i)}} \right) \right] \tag{9}$$

Intuitively, we can take advantage of the fact that the shuffle part skips identity matrices to reduce the computational cost in multiplications and consequently, accelerate the iteration. Another circumstance that can produce low extra memory costs is the existence of ultra sparse matrices at left of γ, specially those having only one scalar. The division point γ can just include non-identity matrices in the shuffle side if the memory needed to store AUNFs is restricted. At the end, with a low number of matrices elements combinations, a sparse-like approach is fully applied. It is important to mention that ultra sparse matrices are recurrent in Kronecker descriptors obtained from general SAN models.

4.1 Strategies to Generalized Tensor Products

Another relevant aspect concerns the production of generalized descriptors, *i.e.*, having functional elements within its transition system. The use of classical or generalized tensor products affects the way matrices are divided using Split, since the characteristics and constraints are different for each case. For instance, using GTA, each function have a list of parameters (matrices) that should be available for the VDP method every step of the iteration method, defining at runtime the rates to be applied to each transition. The matrices that are parameters of functions should be placed at left of the matrix containing the functional element. This locality constraint related to function parameters, establishes that generalized descriptors are more complex to deal with Split, which can produce undesirable amounts of memory or functions evaluations, depending on the possible matrices reordering and γ.

Following we present a representative example of tensor product to better explain what can be done within Split. Table 2 demonstrates a visual depiction of the possible restructurings or permutations for a given tensor product to exemplify the available choices for reordering on the Split method application. We omitted the generalized tensor product operator ($\otimes_g$) among all matrices for simplicity reasons. The table shows a given tensor product of five matrices ($\mathcal{A}^{(1)}$, $\mathcal{A}^{(2)}$, $\mathcal{A}^{(3)}$, $\mathcal{A}^{(4)}$ and $\mathcal{A}^{(5)}$) representing an event occurring among components within the Kronecker descriptor. The division points possibilities can vary from γ_0 to γ_5. For this example, because the tensor product is based on GTA, the function f is defined in a transition on the matrix $\mathcal{A}^{(3)}$. The function could test, for example, whether component $\mathcal{A}^{(1)}$ is in state 1 and (&) component $\mathcal{A}^{(5)}$ is in state 0, the component $\mathcal{A}^{(3)}$ will be able to fire a transition changing its state with a rate r, if both boolean functions ($=$) are evaluated to *true*: $[f \leftarrow (st\ \mathcal{A}^{(1)} = 1\ \&\ st\ \mathcal{A}^{(5)} = 0) \times r]$.

Due to all the different manners on which the generalized tensor product can be reorganized, we will explain the choices to rearrange it, explaining its impact on memory, *i.e.*, the list of AUNFs[1] for the event, as follows:

• **Case (a):** It presents the original order for the tensor product before the application of matrices permutations. It is representing the occurrence of an event in a system having five components or matrices named $\mathcal{A}_1$ to $\mathcal{A}_5$, with different dimensions and sparsity. For instance, the tensor product in question have matrices with dimensions $\{2, 3, 2, 3, 4\}$ and sparsities $\{2, 3, 1, 1, 4\}$.

[1]The number of AUNFs for each tensor product is the product of non-zeroed elements until γ, *i.e.*, $|AUNFs| = \prod_{i=1}^{\gamma} nz_i$.

The matrices types are also variable, which are respectively $\{$*constant, identity, functional, constant, identity*$\}$, and the types *constant* and *functional* differ basically in the absence or presence of functional elements. In this case, it is possible to divide the tensor product anywhere between the matrices, from γ_0 to γ_5. The choice of γ reflects the memory to be spent to perform the VDP procedure. However, it is also possible to rearrange the matrices to maybe spend less memory or organize the tensor product by matrix type, as next case shows;

• **Case (b):** At this point, it is crucial to decide *where* (in the sparse or structured part) to evaluate the functional element for the tensor product. This case shows an example where the functions are evaluated in the structured part, as the table is showing. Because f needs information of matrices $\mathcal{A}^{(1)}$ and $\mathcal{A}^{(5)}$, all three matrices (and $\mathcal{A}^{(3)}$ as well) should belong to the same side, so the parameters are available for the procedure. The chosen division point for this case would be set at γ_2 and the number of AUNFs or scalars for the sparse part would be $3 \times 1 = 3$. It is worth of noticing that the matrix containing the functional element ($\mathcal{A}^{(3)}$) should be put at the rightmost position, assuring that each one of its parameters will be at its left side, validating the evaluation;

• **Case (c):** This case shows an example where the functions are evaluated in the sparse part. The same constraint for the functional element applies, *i.e.*, the parameter matrices should be attached together at the side where the function lies. For this case, the division point will be set at γ_3 and $|AUNFs| = 2 \times 4 \times 1 = 8$;

• **Case (d):** From the previous case, the matrix $\mathcal{A}^{(4)}$ is of type *constant*, moreover it is ultra sparse and it has only one non-zeroed element. If this matrix was to be shifted to the sparse part, it would not affect the memory, since the number of scalars would remain unchanged (set at 8 as mentioned previously). Another important consideration is that if only matrices of type *identity* are left in the structured part, the method would skip it, due to the optimization also presented in Equation (9). However, it is not possible to shift matrix $\mathcal{A}^{(5)}$ (also an identity) to the structured part and leave the structured side with only identities because this matrix is a parameter for function f. In this case, the division point is γ_4 and the tensor product will be treated almost as purely sparse, because the identity matrix is skipped from a Shuffle algorithm point of view, removing the computational costs associated to the method;

• **Case (e):** The unique condition for a functional matrix to be correctly evaluated is to assure that it is located in the rightmost position and that its parameters are available on its left. So, one can reorder the tensor term to evaluate functions in the structured part, decide to put all identities matrices in the same side taking advantage of the optimization discussed in Case (d), and working with only matrices of type constant (mainly those classified as ultra sparse matrices) in the sparse part. On that case, the number of AUNFs is given by $2 \times 1 = 2$ and the cut is set on γ_2. Nevertheless, the function will be evaluated each time in the iteration whereas if the function is located at the sparse part, it would be evaluated just once and simply used, rather than recalculated.

• **Case (f):** In the absence of functional dependencies, for example replacing the functional matrix $\mathcal{A}^{(3)}$ by an identity, one can maintain the original order for the classical tensor product or perform permutations placing the more sparse matrices at left of the division point and the identities at right as demonstrated. On that case, the number of AUNFs is also given by $2 \times 1 = 2$ and the cut is set on γ_2. As discussed in the Case (d) it is better to put matrices of type *identity* in the structured part, because the method would skip it. In the sparse part, identities would only degrade the generation of AUNFs. Moreover, permutations of constant matrices to the structured part can be also done spending less additional memory if needed, storing just a controlled number of AUNFs.

The reordering presented in the Table 2 showed how to restructure generalized and classical Kronecker products and how to deal with functional matrices and their dependencies. It is possible to infer important conclusions about the solution performance from the reordering cases presented. It is well known that the computational cost of using permutations is very low because it fundamentally deals with precalculated rearrangements for the necessary jumps within the tensor structure. The example also showed that different matrices organizations and custom division points should be considered for each event in a descriptor, due to the inner tensor product characteristics such as matrices type, sparsity, and so on. Note that it is possible to quantify the level of interaction among components of a system through an event, which will dictate the level of sparsity within the tensor products. If an event affects, interferes or involves few other entities, its inner matrices will possess high sparsity properties mainly because it suffices to encode the information about where the synchronization takes place for every event. This fact is crucial to Split, directly influencing the additional memory to be spent and the position for the division point γ.

However, one can rely on the flexibility strength of the Split algorithm to treat every tensor product differently and discover the adequate matrices ordering for every case. The only trade-off to observe is whether spend or not more (or less) memory for each choice of division point that is derived. Depending on the case

Table 2: Restructuring to apply Split method in generalized tensor products.

(a)	$\gamma_0 \underbrace{\overset{\mathcal{A}^{(1)}}{\begin{pmatrix}0&1\\2&0\end{pmatrix}} \gamma_1 \overset{\mathcal{A}^{(2)}}{\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}} \gamma_2 \overset{\mathcal{A}^{(3)}}{\begin{pmatrix}0&0\\f&0\end{pmatrix}} \gamma_3 \overset{\mathcal{A}^{(4)}}{\begin{pmatrix}0&0&0\\0&0&0\\0&5&0\end{pmatrix}} \gamma_4 \overset{\mathcal{A}^{(5)}}{\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix}}}_{\textit{original automata order (without transformations) for a given generalized tensor product and available division points: } \gamma_0 \textit{ to } \gamma_5} \gamma_5$
(b)	$\gamma_0 \underbrace{\overset{\mathcal{A}^{(2)}}{\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}} \gamma_1 \overset{\mathcal{A}^{(4)}}{\begin{pmatrix}0&0&0\\0&0&0\\0&5&0\end{pmatrix}}}_{\textit{sparse part}} \overset{\downarrow}{\boldsymbol{\gamma_2}} \underbrace{\overset{\mathcal{A}^{(1)}}{\begin{pmatrix}0&1\\2&0\end{pmatrix}} \gamma_3 \overset{\mathcal{A}^{(5)}}{\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix}} \gamma_4 \overset{\mathcal{A}^{(3)}}{\begin{pmatrix}0&0\\f&0\end{pmatrix}}}_{\textit{structured part}} \gamma_5$
(c)	$\gamma_0 \underbrace{\overset{\mathcal{A}^{(1)}}{\begin{pmatrix}0&1\\2&0\end{pmatrix}} \gamma_1 \overset{\mathcal{A}^{(5)}}{\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix}} \gamma_2 \overset{\mathcal{A}^{(3)}}{\begin{pmatrix}0&0\\f&0\end{pmatrix}}}_{\textit{sparse part}} \overset{\downarrow}{\boldsymbol{\gamma_3}} \underbrace{\overset{\mathcal{A}^{(2)}}{\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}} \gamma_4 \overset{\mathcal{A}^{(4)}}{\begin{pmatrix}0&0&0\\0&0&0\\0&5&0\end{pmatrix}}}_{\textit{structured part}} \gamma_5$
(d)	$\gamma_0 \underbrace{\overset{\mathcal{A}^{(1)}}{\begin{pmatrix}0&1\\2&0\end{pmatrix}} \gamma_1 \overset{\mathcal{A}^{(5)}}{\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix}} \gamma_2 \overset{\mathcal{A}^{(3)}}{\begin{pmatrix}0&0\\f&0\end{pmatrix}} \gamma_3 \overset{\mathcal{A}^{(4)}}{\begin{pmatrix}0&0&0\\0&0&0\\0&5&0\end{pmatrix}}}_{\textit{sparse part}} \overset{\downarrow}{\boldsymbol{\gamma_4}} \underbrace{\overset{\mathcal{A}^{(2)}}{\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}}}_{\textit{structured part}} \gamma_5$
(e)	$\gamma_0 \underbrace{\overset{\mathcal{A}^{(1)}}{\begin{pmatrix}0&1\\2&0\end{pmatrix}} \gamma_1 \overset{\mathcal{A}^{(4)}}{\begin{pmatrix}0&0&0\\0&0&0\\0&5&0\end{pmatrix}}}_{\textit{sparse part}} \overset{\downarrow}{\boldsymbol{\gamma_2}} \underbrace{\overset{\mathcal{A}^{(2)}}{\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}} \gamma_3 \overset{\mathcal{A}^{(5)}}{\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix}} \gamma_4 \overset{\mathcal{A}^{(3)}}{\begin{pmatrix}0&0\\f&0\end{pmatrix}}}_{\textit{structured part}} \gamma_5$
(f)	$\gamma_0 \underbrace{\overset{\mathcal{A}^{(1)}}{\begin{pmatrix}0&1\\2&0\end{pmatrix}} \gamma_1 \overset{\mathcal{A}^{(4)}}{\begin{pmatrix}0&0&0\\0&0&0\\0&5&0\end{pmatrix}}}_{\textit{sparse part}} \overset{\downarrow}{\boldsymbol{\gamma_2}} \underbrace{\overset{\mathcal{A}^{(2)}}{\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}} \gamma_3 \overset{\mathcal{A}^{(3)}}{\begin{pmatrix}1&0\\0&1\end{pmatrix}} \gamma_4 \overset{\mathcal{A}^{(5)}}{\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix}}}_{\textit{structured part}} \gamma_5$

and the number of other elements that an event affects, the necessary memory will vary accordingly, showing the importance to choose the suitable reorder and the appropriated matrices to be shifted between the sparse and the structured part. Next section will present a discussion about these possibilities and point out when Split performs better than previous VDP approaches.

5 Numerical Analysis

There are a few clear advantages on where to divide matrices of the tensor products without impairing the memory and decrease the computational complexity at each iteration. There are a few considerations to discuss, mostly involving the existing time-memory trade-offs when executing the Split algorithm. Saving the list of AUNFs in advance means to precompute the scalars for later use. When this auxiliary list is stored, it implicates in a substantial reduction of index calculations that are necessary to the traditional Shuffle algorithm.

Depending on the model structure, *i.e.*, if a given event affects few elements, it will have hyper or ultra sparse [6] matrices and the tensor product will have a large number of identities matrices. As mentioned before, if lots of ultra sparse matrices are presented with identities, it corresponds to a better execution for the Split procedure. Each tensor product should be uniquely studied to determine the adequate division point γ to be applied, minimizing memory spent in auxiliary structures, thus providing quicker iterations.

It is a fact that identity matrices are best dealt in the structured part because not only it is skipped by the Shuffle algorithm but also it retains no information at all for the stored scalar in the list of AUNFs. The position of γ dictates the block where the scalars will be extracted from the input vector and then multiplied in the output vector, of size $nright_\gamma$ (refer to Section (3)). This value should be carefully chosen because for γ values that are closer to the beginning of the tensor product it will be necessary two times $nright_\gamma$ multiplications (lines 10 and 33 of Algorithm (1)) for every scalar s (lines 4 to 8) presented in the list.

For the GTA case, where there is the constraint of the matrices that are parameters for functional elements, the apparent advantage is to evaluate it on the sparse part, since the value will be saved for later use. The alternative, *i.e.*, evaluating functions in the structured part, depending on the case, produces high overheads per iteration, because it must evaluate functions every time, rather then evaluating once and saving for afterwards.

What Split really does for the CTA case and for models which have at least hyper sparse matrices and with a high number of identities, is to discover in the tensor structure, for each event, huge blocks that must be multiplied by the scalars, which are in fact skipped by the structured part later. Split aggregates the necessary scalars together and precomputes its input and output indices beforehand, so it does it only one time, prior to the whole execution of the iterative method. It is possible to anticipate that for models with a particular set of events that synchronizes activities with a few number of elements, the Split algorithm will perform faster, because of the reduction of the complexity involved and will still use (for some cases), small amounts of memory.

This section helped the understanding of how a Kronecker descriptor is defined and how its inner matrices could benefit from the flexible solutions, *i.e.*, defined by the Split algorithm. There are a few number of characteristics that affects the execution of Split for the CTA and GTA cases, which are outlined as follows: (i) the matrices dimensions, *i.e.*, the number of states for each component; (ii) the number of non-zeroed elements; (iii) the number of functions and the number of parameters for evaluations and (iv) the number of identity matrices in the given tensor product. This four characteristics plays a major role on the computational complexity of Split presented in the Equation (9).

5.1 Practical Results

This section will present the main obtained results for the Kronecker models considered. To understand each strategy to divide and rearrange the tensor products a set of experiments were defined. The objective of the experiments are to test the effect of the different manners to reorder the descriptor applying Split method, according to the inner characteristics such as permutations, matrices types and sparsity, to name a few. The purpose is to unveil the adequate options and explain it, looking the associated computational complexity, trying to discover the class of models on which it is suitable to choose a given strategy. For that matter, we chose nine experiments varying the descriptor type (CTA or GTA) and the available strategies to apply a transformation and choose the division point γ, as shown in Table 3.

We are comparing the Split algorithm strategies against the classic Shuffle algorithm applying permutations where it is needed (for Shuffle, permutations are mainly used for GTA descriptors since CTA descriptors would suffer with an additional overhead related to indices calculation). Unless stated otherwise, all experiments CTA and GTA are using permutations.

Before presenting the results, we describe our methodology to conduct the experiments. The execution was done in a *Intel Pentium* 3.2 GHz with 4 GB of RAM and 2 MB of *cache* memory. The PEPS tool [4] was used as basis for the implementation where the Split algorithm was coded for both CTA and GTA. The storage and generation of the AUNFs listing and other auxiliary data structures were coded using the *GNU Compiler Collection* with g++ version $4.0.4$ with optimization options ($-O3$) and dynamic linking for the functional elements.

Exp.	VDP	Type	Description
1	*Split*	CTA	identities to the structure part - Case (f)
2	*Shuffle*	CTA	classic Shuffle method without permutations
3	*Split*	CTA	division point in the last constant matrix no permutations - Case (a)
4	*Shuffle*	CTA	classic Shuffle method with permutations
5	*Split*	GTA	functional evaluation on structured part - Case (b)
6	*Split*	GTA	functional evaluation on sparse part - Case (c)
7	*Split*	GTA	evaluations on the sparse part only identities on the structured part - Case (d)
8	*Split*	GTA	evaluations on the structured part only identities on the structured part - Case (e)
9	*Shuffle*	GTA	classic Shuffle method with permutations

Table 3: **Experiments to test the different strategies.**

The time spent are computed with confidence intervals of 95% for 50 sequential executions with 25 iterations each (where the mean value of the iterations is computed and stored). The results are showing the time needed to run a single iteration with Shuffle or Split. It is worth mentioning that this time is multiplied by the total number of iterations necessary for convergence, *i.e.*, even small gains in comparison with Shuffle will signify a huge gain when the whole iterative method is considered.

We are working with traditional models and we will divide our result section in two sections, firstly studying the models defined originally with functional elements (all models indicated by GTA after their identification names) and then they were translated to the equivalent classical version (all models indicated by CTA after their identification names). Secondly, studying the models having just CTA based descriptors. The identification names of models present in the results tables, include also the number of automata we are dealing with. The first set of results concerns the models *Wireless ad hoc Networks* (WN for short)[15] and the *Master-Slave* (MS) [2]. The rest of the models are the *Dining Philosophers* (DP), *Non-Uniform Memory Access* (NUMA) [7], *Queuing Networks* (QN), *Resource Sharing* (RS), *First Available Server* (FAS) and *Alternate Service Patterns* (ASP).

Table 4 shows the result for 14 and 16 nodes in a WN reality and for 8 and 10 slaves for the MS model. The table have two parts showing the time per iteration and the memory spent for every experiment. The table works with CTA and GTA models, separated by two parts. The first part, on top, shows the time computed for a single iteration for all the experiments that were studied (column *Exp.*). The second part, on bottom, is showing the memory spent for the execution of the different algorithms, *i.e.*, Split or Shuffle. The column PSS shows the Product State Space, which is the Cartesian product of the states of every element composing the model. From the table it is possible to infer when it is advantageous to convert a given descriptor based on GTA to CTA (by replacing the functional elements to synchronizing events [5]) given the time obtained for the different cases.

Model	PSS	Time (s) – 1 iteration								
		CTA				*GTA*				
		Exp. 1 *Split*	Exp. 2 *Shuffle*	Exp. 3 *Split*	Exp. 4 *Shuffle*	Exp. 5 *Split*	Exp. 6 *Split*	Exp. 7 *Split*	Exp. 8 *Split*	Exp. 9 *Shuffle*
wn14	2,125,764	**0.39**	2.74	0.50	2.18	2.65	1.10	**0.64**	2.51	6.83
wn16	19,131,876	**4.02**	28.10	5.07	22.67	29.86	11.58	**6.60**	28.36	79.31
ms08	807,003	**0.39**	1.47	0.63	1.03	0.55	**0.40**	0.52	0.55	1.20
ms10	7,263,027	**4.18**	16.11	6.65	11.34	5.98	**4.25**	5.41	5.98	13.38
		Memory – AUNFs (MB)								
wn14	2,125,764	≈**0**	–	12.61	–	4.09	0.02	0.02	≈**0**	–
wn16	19,131,876	≈**0**	–	113.52	–	38.44	0.03	0.03	≈**0**	–
ms08	807,003	**8.55**	–	40.65	–	**8.23**	8.63	24.25	**8.23**	–
ms10	7,263,027	**75.17**	–	364.35	–	**73.91**	77.51	218.07	**73.91**	–

Table 4: **Comparison between *WN* model and *MS* for CTA and GTA.**

In terms of time and observing only the Shuffle approach, for the WN model with 16 nodes case (wn16), we learn that it is appropriate to convert this class of model to CTA, since the time is reduced from 79.31 seconds (Exp. 9) to 22.67 seconds (Exp. 4), a gain of order 3.5 times in comparison. The same is not as trivial to detect for the MS model with 10 slaves (ms10) since the time is near between both approaches, *i.e.*, 13.38 seconds and 11.34 seconds for the respectively Experiments 4 and 9.

However, when comparing *Shuffle* with *Split* with the same type of descriptor, and specifically for the WN case, it is necessary 79.31 seconds to execute the first (Exp. 9) but 6.60 seconds for the second approach (Exp. 7), a ≈10 times faster result. When converting this case to CTA, the verified gain is inferior, of 22.67 seconds (Exp. 4) to 4.02 seconds (Exp. 1), but it is still a remarkable gain, since the memory remains bounded to the usage of small amounts. For the ms10 case, it does not matter whether to convert the model, since the time measurements are 4.25 (Exp. 6) and 4.18 (Exp. 1), in contrast with *Shuffle*, with 13.38 seconds (Exp. 9) and 11.34 seconds (Exp. 4). The memory for this case is still manageable, consisting of $\approx$74 MB. It is worth mentioning that this gains are for a single iteration, *i.e.*, the total gain should be observed when it reaches the stationary regime. Even small gains in terms of time influence the total time needed when we take the process as a whole. The examples showed the existence of a particular class of models on which it will produce faster results when the descriptor type is changed, from GTA to CTA. This is only possible when the functions are simple, only testing that the other elements are in a given set of states, causing the synchronizing events to be added as extremely sparse and with lots of identity matrices, the reason why the Split method was built to address efficiently.

Table 5 shows the results for classical models without a similar representation using GTA descriptors. It is noticeable that the additional memory necessary for all models is negligible, enforcing the fact that Split for this cases should be recommended, specially when we observe the time to perform an iteration. The table shows, except for the ASP model, that Split ran faster than Shuffle, sometimes with high orders of magnitude, which is the case of the NUMA model (27 times faster). The NUMA model have a large amount of synchronizations that affects only two components or automata. This causes the descriptor to have two highly sparse matrices and the rest is composed by identities. For the FAS model with 20 servers (fas20), the Split ran in 0.36 seconds (Exp. 1) per iteration whereas the best Shuffle time (Exp. 4) needed 2.80 seconds, *i.e.*, 7.78 times faster. This is also verified for the DP case with 15 philosophers, where Experiment 1 with Split needed 6.25 seconds and the best Shuffle (Experiment 4) took 16.63 seconds to compute one iteration, a considerable gain, since Split used a small amount of memory to be 2.66 times faster.

Just by cleverly applying restructurings to the Kronecker descriptor we produce faster results, exploring the fact that Split is flexible enough to work with any tensor product reorder and use an auxiliary list of scalars that, for some cases, uses limited additional memory per iteration to compute the same results as Shuffle does. The ASP model produced similar results in comparison with both VDP algorithms due to the fact that it has a constant number of components (in the case, four queues and one server, totalizing constantly 4 automata), varying only

Model	PSS	Time (s) – 1 iteration			
		Exp. 1 *Split*	Exp. 2 *Shuffle*	Exp. 3 *Split*	Exp. 4 *Shuffle*
fas19(CTA)	524,288	**0.17**	1.55	**0.17**	1.22
fas20(CTA)	1,048,576	**0.36**	3.44	**0.36**	2.80
rs14_11(CTA)	196,608	**0.11**	0.34	0.22	0.23
rs15_15(CTA)	524,288	**0.34**	1.02	0.64	0.68
asp12(CTA)	342,732	0.25	0.30	0.84	**0.23**
asp14(CTA)	399,854	0.33	0.40	1.14	**0.31**
numa6(CTA)	1,166,400	**0.38**	6.52	n/a	n/a
numa8(CTA)	55,427,328	**25.70**	730.19	n/a	n/a
qn1(CTA)	10,140,651	**1.87**	3.19	n/a	n/a
qn2(CTA)	15,813,251	**2.99**	5.14	n/a	n/a
dp14(CTA)	4,782,969	**1.99**	7.66	3.06	5.17
dp15(CTA)	14,348,907	**6.25**	24.23	9.82	16.63
		Memory – AUNFs (MB)			
fas19(CTA)	524,288	$\approx$0	–	$\approx$**0**	–
fas20(CTA)	1,048,576	$\approx$0	–	$\approx$**0**	–
rs14_11(CTA)	196,608	0.30	–	11.01	–
rs15_15(CTA)	524,288	0.44	–	30.01	–
asp12(CTA)	342,732	0.64	–	107.06	–
asp14(CTA)	399,854	0.87	–	145.68	–
numa6(CTA)	1,166,400	$\approx$0	–	n/a	–
numa8(CTA)	55,427,328	$\approx$0	–	n/a	–
qn1(CTA)	10,140,651	3.05	–	n/a	–
qn2(CTA)	15,813,251	3.82	–	n/a	–
dp14(CTA)	4,782,969	$\approx$0	–	137.85	–
dp15(CTA)	14,348,907	$\approx$0	–	413.56	–

Table 5: **Comparison for various CTA based models.**

the number of states (*i.e.*, the PSS). For such cases, it does not matter to use one technique or another, the Split would be more comparable if the number of services were to be added, producing different ways to divide the tensor products.

Table 6 shows the time-efficiency analysis of the VDP process. It exhibits, for a comprehensive set of different models, the number of iterations needed to reach convergence. The table shows the the time of a single iteration (column T_1), multiplied by the total number of iterations (column *#iter.*), calculating the total time for the method execution (column T_t) and presenting the time to solve the model, in minutes (min), hours (h) or even days. To compute the table we used the best time achieved for both Split and Shuffle. The column *Mem.* shows the extra memory spent, in KB. The PEPS tool is bounded to use at most 65 million states for both VDP methods. Both NUMA and QN cases consist of transient analysis, so we chose to use 10.000 iterations to compute the total time. For many cases we observe the gains of Split and we infer that the method described here helps to anticipate the analysis phase since it produces results faster when compared to classical approaches such as the Shuffle technique.

Model	PSS	#iter.	Time (s)				Mem. (KB)	Time to Solve	
			Shuffle		*Split*			*Shuffle*	*Split*
			T_1	T_t	T_1	T_t			
wn14(CTA)	2,125,764	78,029	2.1787	1.70E5	0.3940	3.07E4	≈0	≈1.96 days	≈8.54 h
wn14(GTA)	2,125,764	78,029	6.8353	5.33E5	0.6434	5.02E4	22	≈6.17 days	≈13.94 h
wn16(CTA)	19,131,876	93,126	22.6656	2.11E6	4.0235	3.75E5	≈0	≈24.43 days	≈4.33 days
wn16(GTA)	19,131,876	93,126	79.3119	7.39E6	6.6018	6.15E5	27	≈85.48 days	≈7.11 days
asp12(CTA)	342,732	766	0.2331	1.79E2	0.2478	1.90E2	657	≈2.97 min	≈3.16 min
asp14(CTA)	399,854	750	0.3130	2.35E2	0.3309	2.48E2	891	≈3.91 min	≈4.14 min
fas19(CTA)	524,288	396	1.2254	4.85E2	0.1725	6.83E1	≈0	≈8.08 min	≈1.13 min
fas20(CTA)	1,048,576	416	2.8037	1.17E3	0.3660	1.52E2	≈0	≈19.44 min	≈2.53 min
dp15(CTA)	14,348,907	1,002	16.6351	1.67E4	6.2543	6.27E3	≈1	≈4.63 h	≈1.74 h
dp16(CTA)	43,046,721	1,071	77.4296	8.29E4	19.7111	2.11E4	≈1	≈23.04 h	≈5.86 h
numa6(CTA)	1,166,400	10,000	6.5213	6.52E4	0.3793	3.79E3	5	≈18.11 h	≈1.05 h
numa8(CTA)	55,427,328	10,000	730.1900	7.30E6	25.7019	2.57E5	9	≈84.51 days	≈2.97 days
qn1(CTA)	10,140,651	10,000	3.1923	3.19E4	1.8723	1.87E4	3,132	≈8.87 h	≈5.20 h
qn2(CTA)	15,813,251	10,000	5.1428	5.14E4	2.9891	2.99E4	3,914	≈14.29 h	≈8.30 h
rs15_15(CTA)	524,288	153	0.6812	1.04E2	0.3448	5.28E1	14	≈1.73 min	≈0.88 min
rs20_25(CTA)	27,262,976	180	88.0908	1.59E4	24.7120	4.45E3	31	≈4.40 h	≈1.23 h
ms08(CTA)	807,003	3,094	1.0292	3.18E3	0.3915	1.21E3	8,756	≈53.07 min	≈20.18 min
ms08(GTA)	807,003	3,094	1.2023	3.72E3	0.3985	1.23E3	8,836	≈61.90 min	≈20.55 min
ms10(CTA)	7,263,027	2,696	11.3418	3.06E4	4.1841	1.13E4	76,974	≈8.49 h	≈3.13 h
ms10(GTA)	7,263,027	2,696	13.3783	3.61E4	4.2506	1.15E4	79,372	≈10.01 h	≈3.18 h
							Total	**≈206.50 days**	**≈16.62 days**

Table 6: **Comparison of times until convergence (or transient analysis) for Split.**

It is worth noticing that the number of iterations are dependent of the rates that are used for each model. For the examples presented in Table 6 we used the original rates for the publication in question and sometimes we scale some models to study the different state spaces and its influence on the executions. The table numerically shows the advantages to adopt a flexible technique to perform a faster VDP procedure allowing model refinements to be conducted more rapidly. For a lot of cases the use of Split was surprisingly not memory consuming, *i.e.*, it was necessary only $\approx$80 MB of memory for the worst case (MS model with 10 slaves), which is considerably small. Split in these cases proved to be a valid alternative to perform VDP, specially when we observe the specific characteristics of the descriptor formation (matrices sparsities and the number of identities).

6 Conclusion

One of the major drawbacks of computational modeling of systems is solution. Numerous efforts are been currently devoted to accelerate this process, providing numerical indices for its modelers. In the context of the present work we are particularly interested in faster executions of the Vector-Descriptor Product mechanism. The focus was directed to the Split Algorithm and its time-memory trade-offs when working with large Markovian models.

The modeling efforts and the addition of functional primitives to capture even more intricate behavior is crucial for performance evaluation of various realities. However, it is equally important to *solve* the model and capture its functionality by computing its performance indices. The literature of solving models within formalisms is ample and we are aware that there is a continuing effort to reduce the computational complexity involved and to accelerate iterations until convergence. We described a flexible method to perform VDP and achieve faster iterations explaining the gains for a large number of different models. Both classical Shuffle algorithm and the novel Split approach were compared altogether. The results showed that restructuring each tensor product in the Kronecker descriptor assigning clever cut-parameters reduces the numerical complexity and accelerates solution. With Split, for a large set of classes, performance indices are presented to modelers in a faster manner, anticipating analysis and model refinements.

We also learned that for models where synchronizations affects or interferes with a small amount of other entities, it produces descriptors which are extremely sparse and with lots of identity matrices, suitable for the execution of a more flexible method. We also demonstrated the constraints when GTA descriptors are present in the model and how to profit from converting it or not to CTA or evaluate the functional elements using a sparse approach or a structured one. We showed that the number of parameters and the function complexity plays a major role for GTA descriptors since it will directly affects the additional memory needed to store the scalars. The use of permutations were thoroughly studied with different experiments to detect suitable rearrangements for every tensor product. The objective was to detect the cases where Split would have lesser computational complexity than Shuffle. By translating a GTA model to its CTA version, we showed when it is better to convert functional models to classical descriptors since it will probably result in remarkable time savings. Our research efforts will focus on the precise definition of both classical and generalized descriptors and the most suitable classes to consider for the execution of Split using permutations to adapt the division point.

Finally, future researches will be directed towards to measurement and implementation of a parallel execution of the Split algorithm and how to use multicore architectures to produce even faster results, verifying the implications involved on this problematic. We are aware that Split is less dependable than Shuffle at the VDP core (since it does not decompose a term in normal factors), *i.e.*, Split precomputes indices, encapsulating this information in an auxiliary list (the AUNFs) containing precalculated indices and the scalar.

References

[1] G. Balbo. Introduction to stochastic Petri nets. *Lectures on formal methods and performance analysis: first EEF/Euro summer school on trends in computer science*, pages 84–155, 2002.

[2] 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.

[3] 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.

[4] L. Brenner, P. Fernandes, B. Plateau, and I. Sbeity. PEPS 2007 - Stochastic Automata Networks Software Tool. In *Proceedings of the 4th International Conference on Quantitative Evaluation of Systems (QEST 2007)*, pages 163–164. IEEE Press, September 2007.

[5] 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 (IJSIM)*, 6(3-4):52–60, February 2005.

[6] 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, July 2000.

[7] 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, June 2006.

[8] 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 2nd international conference on Performance evaluation methodologies and tools (ValueTools'07)*, volume 321 of *ACM International Conference Proceeding Series*, Brussels, Belgium, Belgium, 2007. Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering (ICST).

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

[10] S. Donatelli. Superposed generalized stochastic Petri nets: definition and efficient solution. In R. Valette, editor, *Proceedings of the 15th International Conference on Applications and Theory of Petri Nets*, volume 815 of *Lecture Notes in Computer Science (LNCS)*, pages 258–277, London, UK, 1994. Springer-Verlag Heidelberg.

[11] 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.

[12] N. Gotz, H. Hermanns, U. Herzog, V. Mertsiotakis, and M. Rettelbach. Stochastic process algebras. In F. Baccelli, A. Jean-Marie, and I. Mitrani, editors, *Quantitative Methods in Parallel Systems*, pages 3–17. Basic Research Series, Springer, 1995.

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

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

[15] J. Li, C. Blake, D. S. J. D. Couto, H. I. Lee, and R. Morris. Capacity of Ad Hoc Wireless Networks. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, pages 61–69. ACM Press, July 2001.

[16] A. S. Miner, G. Ciardo, and S. Donatelli. Using the exact state space of a Markov model to compute approximate stationary measures. In *Proceedings of the 2000 ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems*, pages 207–216. ACM Press, June 2000.

[17] J. R. Norris. *Markov Chains*. Cambridge University Press, New York, USA, 1998.

[18] J. L. Peterson. Petri Nets. *ACM Computing Surveys (CSUR)*, 9(3):223–252, September 1977.

[19] B. Plateau. On the stochastic structure of parallelism and synchronization models for distributed algorithms. In *Proc. of the 1985 ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems*, pages 147–154, Austin, Texas, 1985. ACM Press.

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

[21] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, Boston, MA, USA, 1995.

[22] W. J. Stewart. *Introduction to the numerical solution of Markov chains*. Princeton University Press, Princeton, NJ, USA, 1994.

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