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

Ricardo M. Czekster, Paulo Fernandes and Thais Webber
PUCRS – FACIN – Computer Science Department
Av. Ipiranga, 6681 - Porto Alegre – 90619-900 – Brazil
{ricardo.czekster, paulo.fernandes, thais.webber}@pucrs.br

ABSTRACT

The description of large state spaces through stochastic structured modeling formalisms like stochastic Petri nets, stochastic automata networks and performance evaluation process algebra usually represent the infinitesimal generator of the underlying Markov chain as a Kronecker descriptor instead of a single large sparse matrix. The best 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 newer option that offers a trade-off between time and memory savings, the Split algorithm. This paper presents a comparison of these algorithms solving some examples of structured Kronecker represented models in order to numerically illustrate the gains achieved considering each model's characteristics.

Keywords

Kronecker Products, Numerical Methods, Optimization of iterative methods.

1. INTRODUCTION

Performance evaluation of large modern systems is 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 techniques 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 of many components, structured formalisms introduce the possibility of describing more than one irreducible component, with interactions among them and individual behavior. Markovian structured formalisms like Stochastic Petri nets (SPN) [1], Stochastic Automata Networks (SAN) [28] and Performance Evaluation Process Algebra (PEPA) [23] offer particular 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, 13, 21] as a very effective way to store quite large and complex models that are extremely hard to deal with by traditional approaches *e.g.*, sparse matrices [31]. 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 individual 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 a *descriptor* [21]. Model storage using a descriptor undeniably reduces the memory requirements [7, 27], but it often entails slower executions to obtain stationary or transient solutions. Nevertheless, the use of descriptors is justifiable, since for many large problems it may be the only option for solution. This fact is illustrated by the definition of tensor format for structured formalisms like SPN [17] and PEPA [24], and not only for SAN where the descriptors were originally proposed.

The numerical algorithms known to efficiently compute the 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 has low CPU cost (when the memory costs are not prohibitive); or
- to use the *Shuffle* algorithm [20, 21], which has low memory costs, but demands a higher CPU cost for many practical cases, even though it is, at the authors best knowledge, the only option to solve generic structured models of tens of millions reachable states.

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 canonical matrix diagrams [26], which relies on clever data structures to hierarchically represent each component's transitions, and may skip entirely the Kronecker representation passing from the SPN expression to matrix diagrams. This solution is quite dependent of the choice of components, which are called SPN partitions, but it delivers a very efficient solution for some practical cases given a good choice of components;
- to use the flexible hybrid vector-descriptor algorithm called *Split* [11, 32], 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 are no functional rates or probabilities in the model. Once again, as in the canonical matrix diagrams approach, the efficiency of the Split algorithm depends on many internal choices.

*The order of authors is merely alphabetical. Paulo Fernandes is partially funded by CNPq grant 307284/2010-7.

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 by a good choice of internal details of the algorithm application.

This paper contribution, therefore, is two-fold:

- the overall computational costs (memory and floating point operations needed) for the Sparse, Shuffle and Split algorithms is formally defined; and
- the efficiency of the Split algorithm is analyzed with the proposal of a set of preprocessing concerns to achieve a better algorithmic performance presenting slight modifications in its solution core when applied to a complete model.

Specifically, we propose and analyze the benefits of different matrix permutations for each term of the descriptor aiming to identify the best algorithmic choices. But in order to do so, we need to compare the computational cost of algorithms intended to solve structured models with very large reachable state space. It is a known fact that a possible disadvantage of structured models of some specific systems is the inclusion of unreachable states. For models of such systems there are clever solutions based on adaptations of Kronecker representation to describe very large Markov models in terms of MTBDDs (Multi-Terminal Binary Decision Diagrams) taking advantage of a large number of unreachable states within the product state space [14]. However, for large models where the product and the reachable state spaces are equally very large decision diagrams techniques cannot help. It is for such cases that numerical solutions, such as those considered in this paper bring some benefits.

This paper is organized as follows. Section 2 presents the basic definitions of the available VDP algorithms with special emphasis on the Split algorithm. Section 3 presents the main contribution of this paper by proposing the preprocessing issue to achieve better performance of the Split algorithm by a flexible application of matrix permutations in each tensor product of the descriptor. Section 4 presents families of models and the numerical results showing the performance increase achieved for the optimized version of Split in comparison with Shuffle and the original Split proposition. Finally, the conclusion emphasizes this paper's contribution towards an optimized vector-descriptor product approach and outlines possible future work 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[1] model 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 terms as one single tensor sum describing all transitions that are independent within each component (transitions within partitions in SPN, or local transitions in SAN and PEPA), plus a pair of tensor product terms for each possible interaction among components (transitions between 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 product ($\otimes$) terms, all of them composed of N matrices (Eq. 1).

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

In this equation:

- $Q_{ind}^{(i)}$ represents the matrix with the rates and diagonal adjustment of the independent transitions of the i-th component;
- $Q_{int_j^+}^{(i)}$ represents the matrix with the rates of the j-th interaction between components for the i-th component;
- $Q_{int_j^-}^{(i)}$ represents the matrix with the diagonal adjustment of the j-th interaction between 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 represented by a single square matrix of the same order as the number of states. However, the order of each matrix in the descriptor (Eq. 1) will be equal the number of states of its corresponding component (n_i). Therefore, the use of a tensor format often represents a huge memory saving, at least for large models.

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

Each VDP 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.

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

2.1 Efficiency of Algorithms

The efficiency of each available VDP algorithm 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:

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

where the subscript j was abandoned to simplify the notation. To characterize the memory and CPU cost of each tensor product, we 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 algorithm, we consider the storage of sparse matrices using Harwell-Boeing format [19], which is a structure composed by three vectors. The first vector (aa) stores each of the nonzero 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 is necessary to store a vector of Real numbers with as many elements as the number of

[1] In the context of this paper only continuous-time models will be considered, since the formulation of discrete-time models is quite different and much more rare. Nevertheless, the reader may find interesting material on discrete-time tensor representation in [5, 29].

nonzeros, and two vectors of Integer numbers, one with as many elements as the number of nonzeros, and another with as many elements as the order of the matrix. Assuming ι bytes for an Integer and ρ bytes for a Real[2], the storage of a matrix with nz nonzeros and order n will take:

$$nz \times \rho + (nz + n) \times \iota \text{ bytes.}$$

For the CPU demand estimation, the number of required floating point multiplications is used, despite the fact that actual CPU processors efficiency is no longer exactly bounded by this number alone. Nevertheless, it is our experience 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.

2.2 Sparse Algorithm Efficiency

The Sparse Algorithm consists in multiplying all nonzero elements of the matrix equivalent to the tensor product and then multiply it by the elements of vector v. It is worth mentioning that the sparse technique is not a form of VDP since it considers every tensor product term as a matrix to be multiplied by the vector v rather than profiting from the tensor structure common to related vector descriptor techniques.

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

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

While the number of floating point multiplications will be:

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

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 clever shuffling of the elements of vector v multiplying it by each matrix $Q^{(i)}$.

Evidently, the memory efficiency of Shuffle in comparison with 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:

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

In contrast, the CPU efficiency of Shuffle is clearly disadvantageous for a general case. However, a simple optimization in the algorithm allows Identity matrices to be treated differently. In fact, the number of multiplications required for Shuffle will be composed of the product of all matrices order (product state space size) regardless of the matrices' characteristics multiplied by the ratio between the number of nonzeros and the order of the non-Identity matrices, *i.e.*:

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

[2]In all 64-bit architectures the storage of an Integer value 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).

2.4 Split Algorithm Efficiency

The Split algorithm is a hybrid approach, between the Sparse and Shuffle algorithms. 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. Still considering a tensor product term with N matrices (Eq. 3), γ denotes this cutting point, where $\gamma = 0$ denotes applying a pure Shuffle approach to all matrices, $\gamma = N$ denotes applying a pure Sparse approach whereas all other possibilities for γ from 1 to $N - 1$ are actual hybrid cases of Split. Figure 1 exemplifies 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 proposed in the next section, it is important to be aware of the basic steps in the Split algorithm. The application of the Split algorithm must be preceded by the choice of the cutting point γ and the preprocessing of the Sparse-like part matrices. This preprocessing fills a data structure named the *Additive Unitary Normal Factor* (AUNF). An AUNF is a triplet consisting of three values, AUNF($sc, \vec{i}, \vec{j}$) defined by:

- (sc) is a scalar value obtained by picking one nonzero element of each matrix in the Sparse-like part (from the first matrix until the matrix of index γ), and multiplying them;
- an input slice of the vector v identified by the row coordinates ($\vec{i}$) of the nonzero elements multiplied; and
- an output slice of the vector v identified by the column coordinates ($\vec{j}$) of the multiplied nonzero elements.

Note that every AUNF is appended to a list containing all triplets that were produced. 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 fetches 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 Shuffle algorithm for the Shuffle part of the 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 an efficient way to store all AUNF's (scalar, row and column indications), and the same sparse structure can be used to store the small matrices of the Shuffle part. Therefore, the memory demand for Split application consists of the individual memory requirements for each part, *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 AUNFs, *i.e.*:

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

3. IMPROVING SPLIT EFFICIENCY

Split must balance 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-like 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 appropriate γ for each tensor product of a descriptor, considering also the possibility of changing the original 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.*, an efficient order and cutting point γ for a given tensor product depends on the characteristics of its matrices, which is not necessarily the same for the other tensor products of the descriptor;
- Reordering tensor products represents a very small computational cost [22], since it corresponds to a simple indirection in access to the coordinates of the multiplying vector according to a permutation;
- The Shuffle algorithm is extremely efficient in handling Identity matrices, as a matter of fact, it just skips their processing, since it is based on a multiplicative decomposition.

Based on the pseudo-commutativity property [21], the Split algorithm can rearrange the original tensor term order (Eq. 3) as follows:

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

where σ is a permutation on the interval $[1..N]$ for each tensor product term and $\sigma(i)$ returns the rank of the matrix $Q^{(i)}$ 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.

3.1 Proposed Heuristic

The use of matrix permutations in the Split algorithm aims to optimize the generation of AUNFs and to reduce the Shuffle part multiplications to a minimum, even to zero if possible. A heuristic to establish the cutting point γ, considering matrix permutations for each tensor product term, becomes quite straightforward by putting all Identity matrices on the right-hand side and the non-Identity matrices on the left-hand side. It results in an optimal reduction of the Split algorithm computational cost, since Identities are skipped in the Shuffle part. In this case γ will be the index of the last non-Identity matrix of the tensor product term.

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 AUNFs. This option only will be required when the number of AUNFs is too big to be stored in memory, which is a rare case in models generated by structured formalisms.

The computational cost to choose such order and cutting point γ to each tensor term is not relevant, since the number of matrices, *i.e.*, the number of automata in the model, is comparatively much smaller than the product state space size. Additionally, it is important to keep in mind that such choice must be made only once in the beginning of the model solution, while the time gains brought by this choice impact in all VDP operations made at each iteration. In models experimented within this paper the solution took from 119 to 93,126 iterations, *i.e.*, for these experiments, the time and memory benefits of reordering were from 2 to 4 orders of magnitude more relevant than the reordering procedure.

It is important to recall that the number of floating point multiplications alone does not guarantee which order and cutting point choice actually is the best one with respect to CPU efficiency. Therefore, starting from the basic concepts advanced here, the next section instantiates the proposed improvements for several models.

4. NUMERICAL ANALYSIS

This section presents the numerical results for classes of models with different state spaces. The objective of our study is to compare Shuffle and Split algorithms and the time to solve the model. The Split algorithm is presented in two implemented versions named *Original* (or orig.) and *Optimized* (or opt.), respectively conserving matrices in the original model order, and permuting matrices according to the optimization discussed in this paper.

The optimized Split algorithm 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, matrix permutations are used extensively, rearranging each tensor product to possess only non-Identity matrices in the Sparse part and Identities in the structured part. In fact, for all examples, the memory usage due to the number of AUNFs was never a restriction.

The experiments in this paper were made on a software package for solving Kronecker descriptors, called GTAEXPRESS [12]. This package was implemented using some of the PEPS tool [6] primitives for Shuffle and adding new code for the Split implementations for original and optimized versions. GTAEXPRESS is coded in C++ and for these experiments it was compiled using g++ version 4.0.4 (GCC – *The GNU Compiler Collection*) with optimization options (-O3) and dynamic linkage.

The chosen execution platform was a 3.2 GHz Intel(R) Xeon(TM) machine with 2 Mb of L2 cache and 4 Gb of RAM. The results were produced running 50 sequential runs of 25 iterations (fixed) to compute the time per iteration information. These runs were statistically handled to obtain a 95% confidence level.

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

Model		PSS	Shuffle		Original Split		#iter.	Time do Solve		Add. mem.	Time
			time(s)	mem.(Kb)	time(s)	mem.(Kb)		Shuffle	Split (orig.)	(Kb)	gain
(i)	RS (P=14;R=10)	180,224	0.31	96	0.20	10,335	119	0.61 min	0.40 min	10,239	1.55×
	RS (P=15;R=15)	524,288	0.68	266	0.34	30,985	153	1.73 min	0.87 min	30,719	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	**112.3 days**	**11.4 days**	max: **407 Mb**	

The five classes of examples presented here are the following SAN models in the descriptor format using only classical tensor algebra, *i.e.*, considering only models without functional rates:

(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, totaling $(4P)$ tensor products with $P+1$ matrices. The product state space is given by $[2^P \times (R+1)]$ states.

(ii) Dining Philosophers (DP) [15] – 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 while waiting for two available forks in order to eat. To avoid deadlock an ordering to get the forks is established, 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 given by $[3^K]$ states.

(iii) Wireless ad hoc Networks (WN) model [18] – 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 *Sink* (destination) node (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. The model has 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 dynamics follow the bag of tasks principle. The processor (master) distributes the work. All remaining automata (slaves) will run the Propagation Algorithm storing the results in the buffer. Assynchronously, the master polls the buffer and processes final computations. For more information regarding this particular model, please refer to [3]. The model descriptor presents $(3S-3)$ synchronizing events, in a total 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 processes and processors in the Operating System, 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 $[(4^R+1) \times 6^R]$.

4.1 Comparison between Shuffle and Original Split

Table 1 presents a comparison between the Shuffle algorithm and the original (non-optimized) version of Split considering one iteration of VDP for each model. It shows the gains obtained applying original Split with the same order for all tensor terms inside an example (no permutation), but with possibly a different cutting point γ for each tensor term. The choice of cutting points was made based on experimentation on all possible cutting points, and the results presented are the best choices according to time efficiency, *i.e.*, the faster cutting point choices.

The columns in this table present the CPU and memory needs expressed in seconds (s) and Kilobytes (Kb), respectively, for Shuffle and Split. The column "#iter." indicates how many iterations were actually needed to solve the model within a 10^{-10} tolerance using the Power method. The Power method was used just as an example since to test the performance of the Split algorithm we are interested in the VPD procedure alone, not in analyzing how quickly the overall method will converge. In this sense, any iterative method could be applied, for example Arnoldi or GMRES (both also implemented in the GTAexpress package). However, these methods may be unaffordable for large models, since they demand additional probability vectors that may not fit into the available memory.

The column "Time to Solve" indicates how much time was actually necessary to reach the solution in each algorithm, *i.e.*, the product of the cost of one iteration and the required number of iterations. These values are presented to bring the benefits in a real application perspective. In fact, according to the actual model rates it may change. Nevertheless, these values offer an empirical perception of the real world time saving brought by Split algorithm. Finally, the last two columns indicate how much memory is necessary to execute the Split algorithm and how much faster it was compared to the Shuffle solution.

The first interesting observation in the results of Table 1 is that the Split algorithm is faster than Shuffle for the set of selected models, for both original and optimized version of the Split algorithm. However, we notice that, as expected, the original version of Split demands more memory than Shuffle, as seen for example in model (v) - NUMA (R=8) - where a reduction of ≈687 seconds per iteration cost ≈373 Mb of additional memory.

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

Model		PSS	Shuffle		Optimized 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**	**7.9 days**	max:**76 Mb**	

Nevertheless, the gains in the overall solution fully justify the application of the Split algorithm, since the memory usage was not too high. This fact is indicated in the last row of Table 1, where the application of Shuffle to solve all models took ≈112 days, whereas the application of the Split original version took ≈10 times less (≈11 days) with an additional use of memory of approximately 400 Mb in the most demanding example, the model (ii) - DP (K=15).

4.2 Comparison between Shuffle and Optimized Split

Table 2 presents a comparison of the Shuffle algorithm results and the optimized version of Split. The choice of cutting point γ was made before starting the iterative method. This choice was made for each tensor product term by the analysis of the term composition and available memory. This procedure has no relevant computational cost considering the gains we can achieve after running many iterations until convergence.

One practical result is that the inclusion of ultra-sparse matrices, *i.e.*, matrices with the number of nonzero elements smaller than the matrix order, in the Sparse-like part does not result in a significant increase of additional memory. In fact, the inclusion of a matrix with a single nonzero element does not represent an increase in the number of AUNFs, causing massive gains in time and memory.

This is an interesting point to analyze models with synchronizing events between components because each event generates an ultra-sparse tensor product, *i.e.*, a tensor product of ultra-sparse matrices. In those cases, such as the examples (ii) and (iii), the combinations of nonzero elements of these matrices do not require a substantial amount of additional memory. An aspect to be considered is the the trade-off between memory usage and CPU time, *i.e.*, if one have lots of memory and wants performance, the γ could be shifted to use the sparse solution approach, while in limited memory systems the choice should allow more weight in the Shuffle part.

The more successful experiments were those for models with interactions only between two components. For these models it is possible, in the Sparse-like part, to aggregate just the matrices corresponding to these interacting components and leave the other matrices (which are Identities) in the Shuffle part. The additional memory is quite small and the processing time is considerably faster than using the pure shuffling approach for each matrix as shown in the results for the models (i) and (v).

Models with large matrices and massive interactions, *e.g.*, the same synchronizing event in more than one transition in a component, spent more memory since the AUNF quantities are drastically increased if this matrix is aggregated in the Sparse-like part. One of these cases occurred in the model (iv) with one of the matrices with order n=41, but even a matrix of order n=20 could produce similar effect. Unfortunately, these models are quite hard to optimize because it is even less interesting to include such large matrices in the Shuffle part. The large order of the matrix increases significantly the number of multiplications for the Split algorithm as may be observed in the last part of Eq. 9.

Nevertheless, 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 interactions occurring mainly between only two components, such as model (iii) that generates few AUNFs to be multiplied and lots of identities to be skipped.

5. CONCLUSION

This paper presented the analysis of tensor product permutations using the Split algorithm and taking into account intrinsic matrix details such as type, order, total number of nonzero elements and their computational cost in both memory and CPU demands. Analysis of these parameters opens the possibility of a deeper understand of the theoretical cost of the Split algorithm and descriptor restructuring to balance memory and execution time.

The solution of classical descriptors, *i.e.*, descriptors having only constant rates, can be numerically interesting when matrix permutations are applied. The bottleneck for performing VDP is still bounded by the probability vector memory requirements. However, the Split algorithm can explore, in future versions, the application of more sophisticated ways to enhance the overall solution of complex Markovian models. An example of such refinements is the use of sparse vector implementations, which access only reachable vector positions. Another approach to consider are data structures such as multivalued decision diagrams (MDD) [10, 30], which deal only with the reachable state space.

However, prior to this effort, it may be also interesting to compare (when possible) the efficiency of traditional RSS-based approaches, *e.g.*[14] and [26], with the VDP-based methods discussed in this paper. Such comparison is, if the RSS size is not prohibitive, a natural work yet to be done.

Another interesting future work is to take advantage of the independence of Split decomposition into additive factors (AUNFs), whereas Shuffle has more dependent operations since it relies on the decomposition in an ordinary matrix product of normal factors. Parallel and even distributed versions of the Split algorithm could be implemented profiting from the finer grain of the algorithm tasks. In this case, the major concern is the needed synchronizations of the resulting probability vector at each iteration of VDP. For the sequential version, memory and time efficiency are

dealt with as a single demand, but parallel implementations should consider other metrics such as the amount of memory needed, the volume of data exchanged and other processing demands to evenly distribute tasks among processing nodes.

We are aware that further research needs to be done towards the impact of functional dependencies on the use of the Split algorithm. In this paper we adapted the models with functional rates using only synchronizing events to represent the same behavior [7]. Our main interest was to study the Split algorithm itself and its numerical complexity. Functional modeling is a strong and useful technique and future work will tackle this issue accordingly. Our efforts will help to propose faster VDP alternatives that could be adapted and used in other Kronecker based formalisms [24, 25, 16]. It is worth noting that functional dependency analysis changed completely the performance of the Shuffle algorithm [21] by taking advantage of generalized tensor algebra properties. It is only natural to suppose that similar gains, and possible matrix permutations, could bring benefits to the Split algorithm as well.

Despite all these interesting areas of future work, it is clear that the gains achieved by the optimized version of the Split algorithm are remarkable. Observing the time needed to solve all the examples presented in this paper with the Shuffle algorithm, we verify that almost three months are required (112.3 days), while the application of the optimized version of the Split algorithm reduced this to little more than a week (7.9 days actually) requiring only 76 Mb of additional memory for the most demanding example (example (iv) MS with S=10;K=40). Note that for the other groups of examples the additional memory was kept within a few Kilobytes. These results are also impressive even when considering the gains achieved in comparison with the original version of the Split algorithm, that resulted in reductions both in memory and CPU needs. Finally, this new version of the Split algorithm upgrades the solution of VDP, and therefore, Kronecker-based solution of structured stochastic models to a higher level of efficiency.

6. 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, May 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, June 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, INPG, Grenoble, France, 2009.

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

[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 (IJSIM)*, 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*, 12(3):203–222, July 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] G. Ciardo, G. Lüttgen, and A. S. Miner. Exploiting interleaving semantics in symbolic state-space generation. *Formal Methods in System Design*, 31(1):63–100, 2007.

[11] R. M. Czekster, P. Fernandes, J.-M. Vincent, and T. Webber. Split: a flexible and efficient algorithm to vector-descriptor product. In *International Conference on Performance Evaluation Methodologies and tools (ValueTools'07)*, volume 321 of *ACM International Conferences Proceedings Series*, New York, NY, USA, 2007. ACM Press.

[12] R. M. Czekster, P. Fernandes, and T. Webber. GTAexpress: a Software Package to Handle Kronecker Descriptors. In *International Conference on Quantitative Evaluation of Systems (QEST 2009)*, pages 281–282, Washington, DC, USA, September 2009. IEEE Computer Society.

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

[14] L. de Alfaro, M. Kwiatkowska, G. Norman, D. Parker, and R. Segala. Symbolic model checking of probabilistic processes using MTBDDs and the Kronecker representation. In S. Graf and M. Schwartzbach, editors, *Proc. 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, volume 1785 of *LNCS*, pages 395–410. Springer, March 2000.

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

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

[17] S. Donatelli. Superposed generalized stochastic Petri nets: definition and efficient solution. In R. Valette, editor, *International Conference on Applications and Theory of Petri Nets (ICATPN'94)*, volume 815 of *Lecture Notes in Computer Science*, pages 258–277, London, UK, 1994. Springer-Verlag Heidelberg.

[18] F. L. Dotti, P. Fernandes, A. Sales, and O. M. Santos. Modular Analytical Performance Models for Ad Hoc Wireless Networks. In *International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOpt'05)*, pages 164–173, Washington, DC, USA, April 2005. IEEE Computer Society.

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

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

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

[22] P. Fernandes, B. Plateau, and W. J. Stewart. Optimizing tensor product computations in stochastic automata

networks. *RAIRO, Operations Research*, 3:325–351, 1998.

[23] J. Hillston. *A compositional approach to performance modelling*. Cambridge University Press, UK, 1996.

[24] J. Hillston and L. Kloul. An Efficient Kronecker Representation for PEPA models. In *Proceedings of the Joint Int. Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM-PROBMIV'01)*, volume 2165 of *Lecture Notes in Computer Science*, pages 120–135. Springer-Verlag, 2001.

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

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

[27] A. S. Miner and G. Ciardo. Efficient Reachability Set Generation and Storage Using Decision Diagrams. In *Int. Conf. on Applications and Theory of Petri Nets (ICATPN'99)*, volume 1639 of *Lecture Notes in Computer Science*, pages 6–25, Williamsburg, VA, USA, 1999. Springer-Verlag.

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

[29] 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, INPG, Grenoble, France, 2009.

[30] A. Sales and B. Plateau. Reachable state space generation for structured models which use functional transitions. In *Proceedings of the sixth International Conference on the Quantitative Evaluation of Systems (QEST'09)*, pages 269–278, Budapest, Hungary, September 2009. IEEE Computer Society.

[31] W. J. Stewart. *Probability, Markov Chains, Queues, and Simulation: The Mathematical Basis of Performance Modeling*. Princeton University Press, USA, 2009.

[32] 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.