Performance Evaluation of OpenMP-based Algorithms for Handling Kronecker Descriptors

Antonio M. Lima, Marco A. S. Netto, Thais Webber,
Ricardo M. Czekster, Cesar A. F. De Rose, Paulo Fernandes

Catholic University of Rio Grande do Sul (PUCRS) – Faculty of Informatics (FACIN)
Av. Ipiranga, 6681 – Porto Alegre – 90619-900 – Brazil

Abstract

Numerical analysis of Markovian models is relevant for performance evaluation and probabilistic analysis of systems' behavior from several fields in science and engineering. These models can be represented in a compact fashion using Kronecker algebra. The Vector-Descriptor Product (VDP) is the key operation to obtain stationary and transient solutions of models represented by Kronecker-based descriptors. VDP algorithms are usually CPU intensive, requiring alternatives such as data partitioning to produce results in less time. This paper introduces a set of parallel implementations of a hybrid algorithm for handling descriptors and a detailed performance analysis on four real Markovian models. The implementations are based on different scheduling strategies using OpenMP and existing techniques of static and dynamic load balancing, along with data partitioning presented in the literature. The performance evaluation study contains analysis of speed-up, synchronization and scheduling overheads, task mapping policies, and memory affinity. The results presented here provide insights on different implementation choices for an application on shared-memory systems and how this application benefited from this architecture.

Keywords: Parallel Algorithms, OpenMP, NUMA Machines, Markovian Models, Kronecker Descriptors, Performance Evaluation, Scientific Computing.

1. Introduction

Markovian modeling is an important tool to understand problems from several fields,e.g., Bioinformatics, Economics, Engineering, and more specifically to predict the behavior in the Computer Systems domain. These systems normally require large amounts of memory and processing power for a comprehensive description and fast solutions. Kronecker descriptors [1] can minimize memory consumption as they are compact structures to represent very large Markovian systems. A myriad of structured formalisms that use Kronecker (tensor) algebra as a compact representation is available to the research community [2], e.g., Stochastic Petri Nets (SPN), Process Algebra (PEPA), and Stochastic Automata Networks (SAN), among others.

There are many numerical alternatives to extract results from analytical models such as simulation and iterative numerical methods. As Kronecker descriptors are represented in a different structure than traditional Markovian systems new solution algorithms had to be designed. Specialized numerical algorithms were then developed throughout the years to provide support for the stationary solution of models. In particular, the most effective solutions are obtained by Vector-Descriptor Product (VDP) algorithms, such as *Shuffle* [3] and *Split* [4] algorithms. The main difference between these algorithms resides in the required additional memory and computational cost in terms of floating-point multiplications.

In this sense, VDP is the key operation to achieve numerical solution for systems represented by descriptors. The VDP operation multiplies a probability vec-

tor by a descriptor, which is composed of tensor product terms [3]. Each term corresponds to a set of small matrices and tensor product operators. The numerical solution is usually achieved by several VDP iterations until convergence, and the processing time of each product is proportional to the descriptor complexity, i.e. the number, size, and sparsity of tensor product terms. Among several solution methods [5, 1], we have applied the Power method as an example of iterative process containing VDP calls, since to test the performance of the Split algorithm we are mainly interested in the VPD procedure alone, rather than in analyzing how quickly the overall method will converge. Other methods such as Arnoldi and GMRES can be also composed of iterative VDP calls, but these other methods may be unaffordable for large models, since they demand additional probability vectors that may not fit into the available memory.

Algorithms' evolution, processing power and storage of the current computing resources have enabled the evaluation of large Markovian models. Although this resource capability is very powerful to handle the system's complexity, it is still not enough to handle several iterations in feasible time. Therefore, as most of the current machines are based on multi-core technology, the development of parallel solutions to accelerate VDP operations becomes essential. Czekster et al. [6] have developed a parallel solution of Kronecker Descriptors considering data partitioning strategies for the Split algorithm. However, this first parallel approach was only based on MPI [7] primitives and presented low scalability on a distributed memory computing platform.

This paper introduces a set of parallel implementations for shared-memory machines of the Split algorithm running inside Power method iterations, and a detailed performance analysis on four real Markovian models. These implemen-

tations are based on different scheduling strategies using OpenMP (Open Multi-Processing) [8] and existing techniques of static and dynamic load balancing, along with data partitioning available in the literature [9, 10]. The performance evaluation study contains analysis of speed-up, synchronization and scheduling overheads, task mapping policies, and memory affinity. The results presented here provide insights on different implementation choices for an application on shared-memory systems and how this application benefited from this architecture.

2. Solving Structured Markovian Models

Markovian models are widely used in the analysis of computer system performance, reliability, availability, and dependability [1]. Although, in general, when a more complex behavior needs to be represented by a Markov chain, one can take advantage of structured Markovian formalisms [2]. Kronecker (tensor) algebra [11, 12] operators are employed to represent the underlying Markov chain [3] of structured models in a compact format. A system represented in a tensor structure, i.e., the model, is also referred to in the literature as Markovian descriptor.

Descriptors are composed of a set of tensor product terms [12] representing the dependent entity behavior (each one with its own states and transitions), and a tensor sum gathering independent state transitions [3] in each entity. These tensor operations are composed of low dimensional matrices, sometimes highly sparse, responsible for conveying the transition system being represented, e.g., using a high level formalism such as Stochastic Automata Networks (SAN) [13, 3].

Figure 1 depicts the two options of the mapping process of large Markovian models in a structured description that undeniably reduces the needs of memory, avoiding the storage of one single, and usually large, flat matrix. Remark that the

set of tensor product terms composed of smaller matrices combined through Kronecker operators, is equivalent to the underlying Markov chain transition matrix that is never stored in memory.

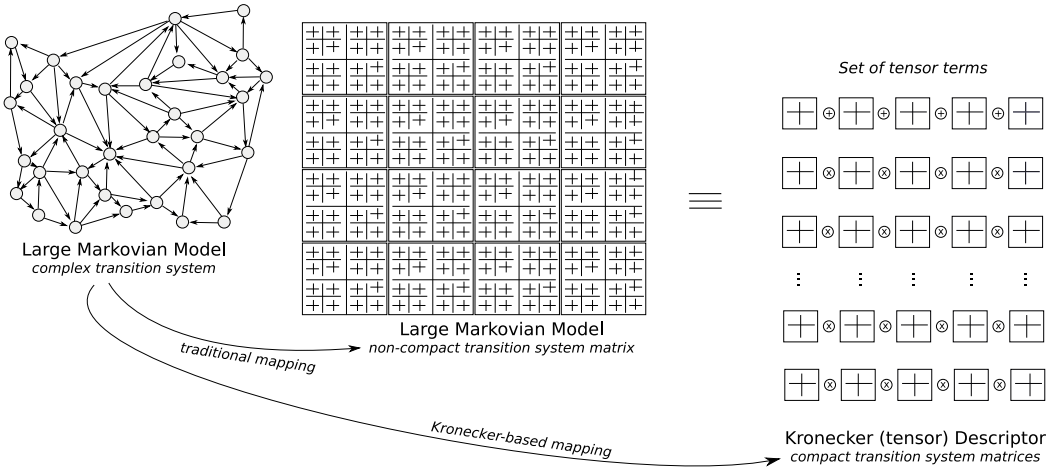


Figure 1: Mapping process of large Markovian models to a compact representation.

Among several structured formalisms present in the literature [2], those using a tensor structure are dependent of specialized numerical solutions. These solutions are concerned in multiplying pieces of a probability vector (as large as the model state space) by a set of matrices that composes the descriptor. It is well known that inside product state spaces there are sets of unreachable states, i.e., some approaches are already proposed to reduce the size of the probability vector to contain only references to the reachable state space. However, some models present characteristics enabling the product state space to be almost comparable to the reachable state space, and particularly for these models, the solution need to be calculated with vectors sized as the product state space, independent of the compact storage of the transition matrix. Nevertheless, it is a fact that operating large probability vectors combined with sets of different sparse matrices directly

influences the total time spent on performing floating-point multiplications.

Despite the state space explosion problem, often responsible for the growth in requirements to store the solution vector, the basic idea for numerically solving models represented by tensor structures is to deal efficiently with vector multiplications by blocks of non-zero elements inside the descriptor. This operation is called *Vector-Descriptor Product* (VDP). There are two known VDP algorithms: the Shuffle algorithm [3] and the recent Split method [4, 14].

2.1. Shuffle Algorithm

The Shuffle algorithm implements the product of a probability vector by the descriptor taking advantage of tensor algebra properties to conduct the overall multiplication process [11, 3]. A probability vector is successively multiplied by each tensor product applying the tensor algebra property for the decomposition of a tensor product term in a product of normal factors [3]. This decomposition allows the treatment of each matrix in a tensor product term in a way that subvectors composed of shuffled elements (from the original vector) are used in the multiplication. Briefly explaining, the property consists in breaking, e.g., decomposing, a tensor product term into a product of new tensor product terms (the normal factors), each one with one single matrix and every other matrix as an Identity matrix. For more information on the application of Kronecker algebra properties, please refer to Fernandes et al. [3]. Therefore, the Shuffle algorithm does not require extra memory to store other matrices or extra large vectors.

However, one of the drawbacks of the Shuffle algorithm is its complexity in accessing descriptors' data, since it is stored in a compact form. Note that descriptors are relatively complex to operate due to the tensor structure despite their advantage in terms of memory efficiency [3]. Additionally, because of the decom-

position in normal factors executed for every tensor product term, the numerical computations depend on one another to complete the multiplications. The dependency established among normal factors makes it extremely hard to devise means to parallelize the multiplication of tensor product terms without compromise performance. The problems aforementioned have motivated the development of a hybrid numerical algorithm handling the trade-off to balance memory usage, and maximizing efficiency in terms of execution time.

2.2. Split Algorithm

The Split algorithm [4, 15] is a hybrid method that executes matrix permutations and aggregations to reduce the cost in floating point multiplications inside iterative methods. Moreover, recent developments [16, 14] have proposed a heuristic, flexible enough to perform fast iterations, for optimizing the execution time of the VDP without impairing the memory, i.e., the algorithm can rearrange matrices in tensor product terms with different strategies, reorganizing the descriptor to balance data storage and numerical operations. On its essence, the Split algorithm deals with the generation of additive unitary normal factors, removing the Shuffle algorithm's constraint related to the sequential computation of each normal factor. In other words, Split does not rely on breaking the tensor product term into dependable normal factors; on the contrary, Split generates independent normal factors, due to the additive decomposition property. This characteristic implies that Split can be more suitable for parallelization efforts than Shuffle due to the granularity allowed for tasks.

Figure 2 illustrates the Split algorithm handling Kronecker based descriptors by gathering the set of tensor product terms, i.e., selecting matrices in a tensor product term based on a *cut-parameter* σ which separates the tensor product term

in two different sets of matrices [15]. The set of matrices at the left side of σ (i.e., matrices to combine) is treated in a sparse-like manner, where the non-zero elements are combined through ordinary multiplications. Each combination of elements in this part is called *Additive Unitary Normal Factor* (AUNF). An AUNF is represented by a scalar, and its coordinates (i,j) in the matrix. From a memory efficiency point of view all AUNFs can be stored in a single sparse matrix. The set of matrices at the right side of σ is composed of the remaining matrices of the tensor product term. These matrices are treated with shuffling operations depending on the heuristic adopted, maintaining the original tensor structure (named shuffle-like part).

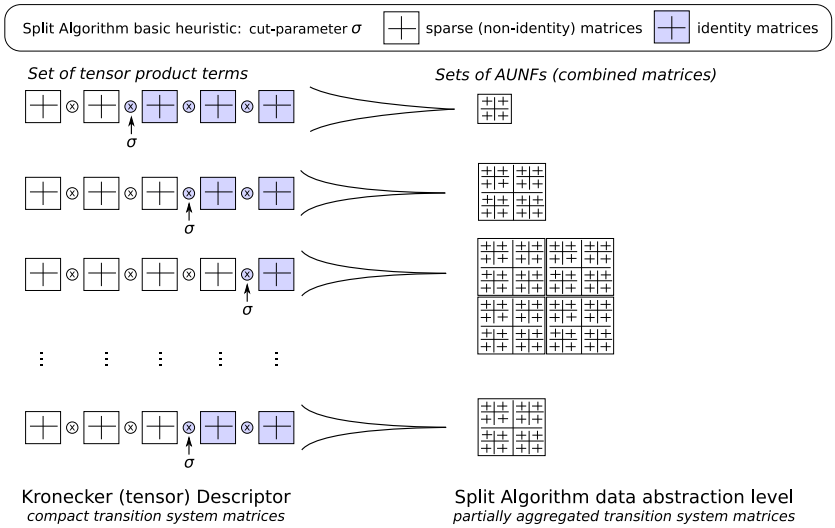


Figure 2: Set of tensor product terms manipulated by the Split algorithm to generate AUNF sets.

Different heuristics [4, 15, 14] could be used to properly select and combine matrices of tensor products on each side of σ to speed-up execution time. In this paper we apply the heuristic proposed for Split [14], profiting from the well-known *identities optimization* [3]. According to this heuristic, only the identity

matrices are placed at right of σ to skip the whole shuffle-part, operating only in the combined matrices. The multiplication is simply performed considering the product of a probability vector (state space sized) by a set of AUNFs. This multiplication process is repeated for all tensor product terms in a descriptor for each step of an iterative numerical method (e.g., Power method, Arnoldi).

Observe that in Figure 2, the sparse-like part is converted in a new sparse matrix, with its dimension related to the tensor combination of the matrices at left of σ. This matrix contains the scalars that must be multiplied by the probability vector. Moreover, the matrices at the right side of σ are omitted in the Split algorithm data abstraction level, because using the basic heuristic in this case implicates just in more multiplications of scalars (generated in the sparse-like part), as the size of the shuffle-like part (the product of the matrices order).

The operation flow of a sequential implementation of the iterative numerical method using the Split algorithm is described as follows. First, a model descriptor (the set of tensor product terms) is loaded to memory. All initial probability vector positions are initialized (of size determined by the cardinality of the state space). Then, for all tensor product terms, all AUNFs are precomputed and stored in a list. The Split algorithm is executed using this list to access and multiply vector positions. Once Split is executed for a single iteration, the Power method is called and tests if the model has reached convergence observing the probability vectors. If not, the Split algorithm is repeatedly executed until stationary regime is achieved. The results are present in the final probability vector containing the steady state information for the model.

The execution of the iterative numerical method in an efficient manner is dependent on several factors: the size of the analyzed model, the number of matrices

represented in the tensor structure, the computational cost related to the sparsity of these matrices, and the behavior adopted by Split setting a *cut-parameter* σ for each tensor product term. Details on VDP methods can be found in the literature [3, 4, 15, 14].

2.3. Parallel Implementation Issues

The complexity in solving Kronecker based models is associated with the state space size and the issues related to the practical application of specialized iterative numerical methods. As the AUNFs can be divided in self-contained groups for computation using the Split algorithm, it is a natural alternative to decompose descriptors within parallel environments. However, from a distributed computing point of view, it is important to focus the study on performance related issues such as the memory and computation bounds, and mainly on the complexity of operations involved in each step of the algorithm.

We have studied how the algorithms can profit from parallel algorithm versions and some observations have emerged. Shuffle is memory efficient but demands several numerical intermediary calculations to work properly. Split stores AUNFs and precomputes positions that it will need afterwards making it not so memory efficient but faster when solving Kronecker descriptors. Split also combines the strengths of Shuffle and sparse storage techniques, using a clever mechanism to address several positions within the Markovian matrices. One could say that Split is more prone to have similar access patterns in memory hierarchies than Shuffle. All the precomputations that Split performs only once in the overall method are used throughout the VDP procedure. However, due to its memory efficiency, Shuffle always computes positions without saving auxiliary structures.

There are two main approaches to implement parallel VDP algorithms: one

based on message passing, for clusters, and another based on shared-memory, for multi-core machines. For both approaches, the main challenge is to define the most suitable task set and size to assign to each processor. For clusters, this is challenging since gathering tasks for reducing communication overhead may cause a poor load balance. For multi-core machines, the challenge of the task assigning comes from properly defining data locality and thread load balancing. For the specific case of this numerical algorithm, a potentially large vector is to be manipulated and thus these parallel processing issues must be taken into account.

3. Markovian Model Examples

This section describes four families of stochastic automata network models varying the product state space sizes, i.e., the models are classified by their probability vector sizes (*Small*, *Medium*, and *Large*). We chose these models because they are representative cases with heterogeneous characteristics, allowing us to evaluate our implementations. We show different models' characteristics, such as state space size, number of local and synchronizing matrices (composing respectively a tensor sum term and the tensor product terms), memory to store the descriptor (in Kb), quantity and the memory to store all AUNFs (generated from all tensor product terms), total time used by each iteration (and the total number of iterations), and sequential execution time for solution. Note that the number of iterations of each model solution is only related to the events rate values, having no correlation with the state space size.

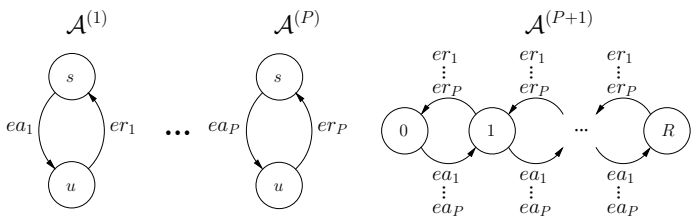
Additionally, for each model, we indicate that a model can be extended (for parallel tests execution, refer to Section 5.2) using the following variables: the number of automata and the number of synchronizing events. The memory sav-

ings can be estimated looking at the number of matrices composing the descriptor and their number of non-zero elements (refer to Section 2.2 for more information as to how generate AUNFs from nonzero combinations). With all these parameters and using each indicated simplified formula, it is possible to calculate the number of tensor product terms in a descriptor for a given number of automata and the related number of events.

3.1. Resource Sharing (RS) model

The classical SAN model for resource sharing [17], where P is the number of processes (descriptor contains squared matrices with two rows, i.e., automata with two states: *idle* and *occupied*) and R is the number of resources (squared matrix with R+1 rows, indicating an automaton in which the states are from 0 to R resources occupied).

Figure 3 graphically shows the SAN model with its synchronizing events (considering $i = 1..P$): event ea_i (acquiring a resource) and event er_i (releasing a resource). The model descriptor presents generally $(2P)$ synchronizing events, totaling $(4P)$ tensor products with P+1 matrices.



Type	Event	Rate
syn	ea_1	λ_1
syn	ea_2	λ_2
⋮	⋮	⋮
syn	ea_P	λ_P
syn	er_1	μ_1
syn	er_2	μ_2
⋮	⋮	⋮
syn	er_P	μ_P

Figure 3: RS Stochastic Automata Network model.

Note that the diagonal adjustment of the event rates are represented in $2P$ tensor product terms and stored in a separated vector to be multiplied, which is

an optimization in VDP methods [3], then there are $2P$ tensor products remaining to multiply using Split. The state space is given by $[2^P \times (R{+}1)]$ states, which is the size of the input/output probability vector to be calculated in the numerical solution. Table 1 illustrates the RS model's variations and characteristics.

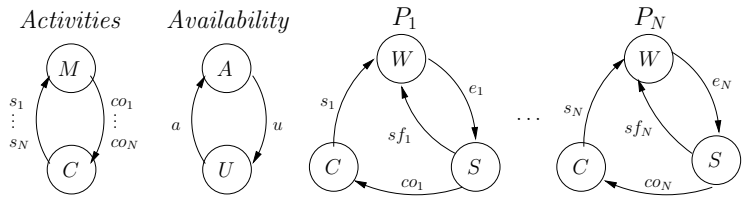
Table 1: Resource Sharing (RS) model configurations.

Characteristics (RS)	**Small P=22; R=4;**	**Medium P=22; R=16;**	**Large P=24; R=8;**
State Space (vector size)	20,971,520	71,303,168	150,994,944
Total Local Matrices (tensor sum)	*none*	*none*	*none*
Total Terms (tensor products for Split)	44	44	48
Normalized Descriptor size (Kb)	10,267	34,844	73,763
Total AUNFs	176	704	384
Split Extra memory for AUNFs (Kb)	≈2.75	≈11.00	≈6.00
Total multiplications (VDP)	369,098,752	1,476,395,008	3,221,225,472
Time per iteration (s)	≈12	≈47	≈105
Power method iterations	30	131	57
Total sequential time (s)	≈350	≈6,197	≈5,997

3.2. Software Development Team (SDT) model

This section shows a model (Figure 4) that depicts a software development team (SDT) communication pattern with a main team, called Central team, in a globally distributed project [18], to analyze the probabilities of waiting periods to solve project issues by different participants.

The model is composed of a central team with two-state automata representing its availability for cooperation with N participants: *Availability* automaton (states A and U, i.e., *Available* and *Unavailable* respectively, related to time-zone overlap in a typical workday), and *Activities* automaton (with states M and C, i.e., *Management* and *Collaboration*). The model also contains a SDT composed of N



Type	Event	Rate
loc	a	λ_a
loc	u	λ_u
loc	$e_1...e_N$	$\mu_1...\mu_N$
loc	$sf_1...sf_N$	$\sigma_1...\sigma_N$
syn	$co_1...co_N$	$\alpha_1...\alpha_N$
syn	$s_1...s_N$	$\beta_1...\beta_N$

Figure 4: SDT Stochastic Automata Network model.

three-state automata as follows: W state means the participant is *working*; S state represents the participant *seeking for a specific information*; and C state means the participant is collaborating to solve technical questions.

Figure 4 illustrates the stochastic automata network corresponding to this scenario. The local behavior of a team member describes that, when members are actually working, they can stop for a while (event e) seeking a solution on their own (event sf), or preferably move to cooperate with the central team (event co), returning to the working state after that (event s). The model descriptor presents generally $(2\times N)$ synchronizing events, totaling $(4\times N)$ tensor products with 2+N matrices. Note that local events are stored together in a tensor sum term. $(2 \times N)$ tensor terms are treated using Split. The state space is given by $[2 \times 2 \times (3^N)]$ states. Table 2 illustrates the SDT model's variations and characteristics. Note that we waited for the entire execution of the sequential programs to find out the total number of iterations. Also, note that the number of iterations for the sequential and parallel program is the same as we paralelized the iterations themselves.

3.3. Alternate Service Pattern (ASP) model

This section shows a model (Figure 5) for open queueing networks [12] having four queues ($\mathcal{A}^{(1)}$, $\mathcal{A}^{(2)}$, $\mathcal{A}^{(3)}$, $\mathcal{A}^{(4)}$) with finite capacities K_1, K_2, K_3, K_4.

Table 2: Software Development Team (SDT) model configurations.

Characteristics (SDT)	**Small (N=14)**	**Medium (N=15)**	**Large (N=16)**
State Space (vector)	19,131,876	57,395,628	172,186,884
Total Local Matrices (tensor sum)	15	16	17
Total Terms (tensor products for Split)	28	30	32
Normalized Descriptor size (Kb)	9,351	28,036	84,088
Total AUNFs	72	77	82
Split Extra memory for AUNFs (Kb)	≈1.13	≈1.21	≈1.29
Total multiplications (VDP)	376,260,228	1,205,308,188	3,845,507,076
Time per iteration (s)	≈8	≈26	≈86
Power method iterations	78,045	71,057	75,259
Total sequential time (s)	≈633,304	≈1,849,428	≈ 6,250,034

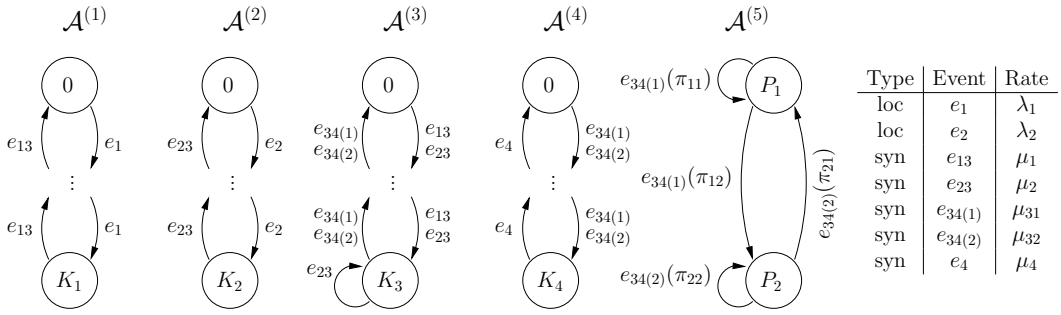


Figure 5: ASP Stochastic Automata Network model.

In the routing pattern of customers they arrive in $\mathcal{A}^{(1)}$ and $\mathcal{A}^{(2)}$ with constant rates λ_1 and λ_2, respectively. Customers may leave from $\mathcal{A}^{(1)}$ to $\mathcal{A}^{(3)}$, if and only if there is room in that queue (blocking behavior), whereas customers may leave from $\mathcal{A}^{(2)}$ to $\mathcal{A}^{(3)}$ whether there is room, or leave the model otherwise (loss behavior). Customers may also leave from $\mathcal{A}^{(3)}$ to $\mathcal{A}^{(4)}$ with blocking behavior. While $\mathcal{A}^{(1)}$, $\mathcal{A}^{(2)}$ and $\mathcal{A}^{(4)}$ have standard (single) service behavior, i.e., considering the same average service rate for all customers (μ_1, μ_2, and μ_4, respectively), queue $\mathcal{A}^{(3)}$ has an Alternate Service Pattern (ASP) behavior. The service rate for

this queue varies according to P different service patterns ($\mu_{31} \ldots \mu_{3P}$). $\mathcal{A}^{(3)}$ can exchange its service pattern simultaneously with the end of service of a customer. Therefore, when a customer is served by the service pattern P_i, automaton $\mathcal{A}^{(3)}$ can remain serving the next customer in the same pattern with probability π_{ii}, or it can alternate to a different service pattern P_j, with probability π_{ij} (for all service patterns $P_i : \sum_{j=1}^{P} P_{ij} = 1$).

Local events e_1 and e_2 represent the arrival in queues $\mathcal{A}^{(1)}$ and $\mathcal{A}^{(2)}$ respectively, and local event e_4 represents the departure from $\mathcal{A}^{(4)}$. Synchronizing events e_{13} and e_{34} represent the routing between queues $\mathcal{A}^{(1)}$ to $\mathcal{A}^{(3)}$ and $\mathcal{A}^{(3)}$ to $\mathcal{A}^{(4)}$ respectively, and synchronizing event e_{23} represents both the routing from $\mathcal{A}^{(2)}$ to $\mathcal{A}^{(3)}$, and the departure from $\mathcal{A}^{(2)}$ due to lack of room in $\mathcal{A}^{(3)}$ (loss).

Note that the extension to a higher number of service patterns will correspond to the addition of more local states to automaton $\mathcal{A}^{(5)}$, which will always have P local states. Event $e_{34}(1)$ and $e_{34}(2)$ have constant rates μ_{31} and μ_{32} respectively. Moreover, a model with P service patterns will contain P synchronizing events $e_{34}(1) \ldots e_{34}(P)$ with rates given by $\mu_{31} \ldots \mu_{3P}$. The model descriptor presents generally $(2 + P)$ synchronizing events, totaling $(4 + 2P)$ tensor products with five matrices (four matrices representing the queues and one matrix representing the service patterns). $(2 + P)$ tensor terms are treated using Split. Note that local events are stored together in a tensor sum term. The state space is given by $[K_1 \times K_2 \times K_3 \times K_4 \times P]$ states. Table 3 illustrates the ASP model's variations.

3.4. Master-slave architecture (MSA) model

This section describes a model for an evaluation of the master-slave parallel implementation of the Propagation algorithm [19] considering asynchronous communication. The model in Figure 6 is composed of one *Master* automaton of three

Table 3: Alternate Service Pattern (ASP) model configurations.

Characteristics (ASP)	**Small (P=5)**	**Medium (P=12)**	**Large (P=16)**
State Space (vector size)	33,826,005	69,177,612	126,247,696
Total Local Matrices (tensor sum)	3	3	3
Total Terms (tensor products for Split)	7	14	18
Normalized Descriptor size (Kb)	16,517	33,780	61,647
Total AUNFs	67,700	336,576	697,840
Split Extra memory for AUNFs (Kb)	≈1,057.82	≈5,259.00	≈10,903.75
Total multiplications (VDP)	327,726,000	1,134,040,320	2,561,448,448
Time per iteration (s)	≈12	≈48	≈110
Power method iterations	1,105	987	1,014
Total sequential time (s)	≈13,332	≈47,089	≈111,336

states (*transmitting*, *receiving* and *idle*), S slaves (automata) with three states each (*idle*, *processing* and *transmitting*), and one large *Buffer* of K+1 positions.

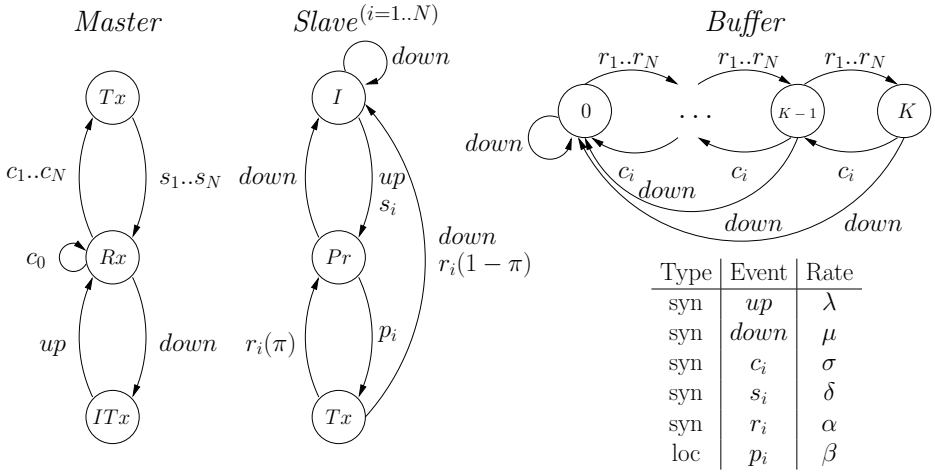


Figure 6: MSA Stochastic Automata Network model.

The *Master* automaton is responsible for the distribution of tasks to slaves and for the analysis of the results evaluated by them. A synchronizing event named

up sends the initial tasks to all slaves, and a synchronizing event *down* ends one execution of an application. The occurrence of the event *up* indicates that all automata must change their actual state for the initial one. Synchronizing event s_i represents the sending of a new task to the i-th slave. Automaton *Master* consumes the *Buffer* content through the synchronizing event c. Finally, *Slave*$^{(i)}$ automaton finishes a task through the occurrence of local event p_i. Synchronizing event r_i represents the reception of completed tasks by the *Buffer*. The model descriptor presents $(3S{+}3)$ synchronizing events, in a total of $(6S{+}6)$ tensor product terms. Note that local events are stored together in a tensor sum term. $(3S{+}3)$ tensor terms are treated using Split. The state space is given by $[3^{(S+1)} \times (K{+}1)]$. Table 4 illustrates the MSA model's variations and characteristics.

Table 4: Master-slave Architecture (MSA) model configurations.

Characteristics (MSA)	**Small S=10; K=256;**	**Medium S=12; K=70;**	**Large S=14; K=14;**
State Space (vector)	45,526,779	113,196,933	215,233,605
Total Local Matrices (tensor sum)	10	12	14
Total Terms (tensor products for Split)	33	39	45
Normalized Descriptor size (Kb)	22,241	55,283	105,109
Total AUNFs	15,445,438	38,021,576	71,974,528
Split Extra memory for AUNFs (Kb)	≈241,334.97	≈594,087.13	≈1,124,602.00
Total Multiplications (VDP)	570,498,594	1,683,360,438	3,598,616,402
Time per iteration (s)	≈15	≈43	≈98
Power method iterations	10,160	3,433	1,986
Total sequential time (s)	≈152,211	≈149,070	≈194,628

4. OpenMP-based Split Algorithm

To achieve parallelism in the VDP method, we have to consider descriptor partitioning, i.e., use of techniques of data partitioning to exploit parallelism. This

section presents data partitioning strategies for the VDP with the Split algorithm, describes the computational costs of the tasks generated on each partitioning strategy, and the OpenMP-based Split implementations.

4.1. Data Partitioning Strategies

There are two ways to derive concurrency in the VDP method with the Split algorithm: partitioning per tensor product term and partitioning per AUNF. This section presents these two approaches, describing the number of tasks and computational costs involved on each one.

4.1.1. Partitioning per tensor product term

One partitioning approach is based on the total number of Kronecker tensor product terms, i.e., a set of tensor terms that form a bag-of-tasks to be distributed among processors. The computational cost in multiplications related to each term is given by $\left(\prod_{i=1}^{\sigma_j} nz_j^{(i)}\right)\left(\prod_{i=\sigma_j+1}^{N} n_j^{(i)}\right)$, where $nz_j^{(i)}$ corresponds to the total number of non-zero elements in the i-th matrix of the term j and $\prod_{i=\sigma_j+1}^{N} n_j^{(i)}$ is the size of the vector to be multiplied. The total number of tasks to be performed in parallel depends on the model characteristics, i.e., the number of tensor product terms in the descriptor. Remark that in the left side of Figure 2, we have a set of Kronecker products (tensor product terms). Considering a partitioning approach per tensor product term, each tensor product term composing the descriptor is considered as a task to be assigned to one processor. Thus, the processor executes all multiplications related to this specific tensor product term. For example, for the models RS, SDT, ASP and MSA we have the total number of tensor product terms given by: $2P$, $2N$, $2{+}P$, and $3S{+}3$ (Section 3), respectively. The total number of tensor product terms refer to the number of tasks in this approach.

As presented, the cost of each tensor product term is defined mainly by the number of nonzero elements and the value of the cut-parameter σ. In this approach, if we have tasks with very different costs and in limited number, it can be difficult to achieve an efficient load balance and scalability of the parallel solution.

4.1.2. Partitioning per AUNF

A different partitioning approach is to distribute the computation of each AUNF, or a set of them, to each processor. In the Split algorithm, every tensor product term is subdivided in smaller tasks corresponding to AUNFs. All the K AUNFs of the j-th term have the same cost, and if summed, the amount is equal to the total cost of the term. The multiplication of each AUNF by a slice of probability vector represents an independent task, after then the result is accumulated in a probability vector. The size of this slice of vector is given by $\prod_{i=\sigma_j+1}^{N} n_j^{(i)}$. The total number of AUNFs per term j is given by the equation $K_j = \prod_{i=1}^{\sigma_j} nz_j^{(i)}$. This approach is possible because every term has at least one AUNF. Observing the right side of Figure 2, we have a set of AUNFs resulting from matrices combinations in a tensor product term. Considering a partitioning approach per AUNF, each additive unitary normal factor (AUNF) composing the descriptor is considered an independent task to be assigned to a given processor. So, the processor will execute just the multiplications related to this AUNF. Note that each tensor product term can generate K_j AUNFs, i.e., independent K_j tasks.

In comparison to the previous data partitioning approach, we have assembled a larger number of tasks with lower computational costs, thus enabling better load balance and scalability.

4.2. Parallel Implementations

We have developed three parallel implementations of the Split algorithm for shared-memory machines using the OpenMP API and the C++ language. The implementations differ in data partitioning and task scheduling strategies.

At the beginning of each iteration of the numerical method, a parallel region is created. Split is a loop-based algorithm that iterates among the tensor product terms and AUNFs. Thus, the parallelization is accomplished through the distribution of loop iterations across the threads. The probability vector π is a shared variable that is updated at the end of each task. Therefore, this variable access must be protected to avoid data race conditions. For enabling multiple threads to update the shared vector π, we have used the *atomic* construct that is an efficient alternative to the *critical* construct [8].

4.2.1. OpenMP-based scheduling

The first two parallel implementations use the *for* work-sharing construct from OpenMP. They also use the *schedule* clause, which specifies how the iterations of the loop are assigned to the threads. We choose the *dynamic* schedule type with task granularity equals to one. In this scheduling strategy, one iteration at a time is assigned to each thread, until there are no more iterations available [8]. The *dynamic* schedule is more suitable to unbalanced workloads and very useful when the computational cost of the tasks is unknown. Additionally, the chosen task granularity is more flexible and generic concerning load balancing and scalability than larger ones. On the other hand, by using a generic setting we can see more clearly the differences among different input models.

Algorithm 1 presents the first parallel implementation that uses partitioning per tensor product term. A parallel region is created with the directive *#pragma*

omp parallel (line 1) and the loop is parallelized via the *for* construct (line 2). In this implementation, there are T tensor product terms to be distributed among the threads following a dynamic scheduling strategy. Using the *private* clause, we specify that each thread has its own copy of variables j, k, and vector υ. In addition, the shared variables are the list A of AUNFs and the global state vector π. The update of π is performed in the inner loop (line 7), where we have the needed information to compute the indices of π to be updated. As multiple threads may simultaneously write at the same positions of π, we treat the region (line 7) with the *atomic* construct. The end of the parallel block occurs after line 7.

Algorithm 1: **TP-Dyn** - *Partitioning per term* j

```
1 #pragma omp parallel for private(j,k,υ) schedule(dynamic,1)
2 for j ∈ [1...T] do
3     for k ∈ [1...K_j] do
4         υ = A[j].scalar[k] × π_0
5         ...
6         #pragma omp atomic
7         π += υ
```

Algorithm 2: **AUNF-Dyn** - *Partitioning per* AUNF k

```
1 #pragma omp parallel for private(k,υ) schedule(dynamic,1)
2 for k ∈ [1...K] do
3     υ = A.scalar[k] × π_0
4     ...
5     #pragma omp atomic
6     π += υ
```

Algorithm 2 uses a partitioning per AUNF and works similarly to Algorithm 1. For this matter, Algorithm 2 has a global list of AUNFs and contains a single loop to iterate over the tasks. Therefore, there is one set of tasks consisting of all

AUNFs of the descriptor to be distributed across the threads.

4.2.2. Manual static scheduling

As the *static* schedule from OpenMP does not handle heterogeneous tasks, we have implemented a manual static scheduling, which is based on *worst-fit decreasing* solution for the *bin packing* problem [20]. This strategy sorts the tasks in descending order based on the computational costs of each task and then schedules one by one, beginning from the least loaded thread. By using this strategy all threads probably will have tasks assigned impacting in the scalability of the implementation. Furthermore, when larger tasks are scheduled first the load balance is impacted (i.e., it is easier to obtain load balance working with small tasks).

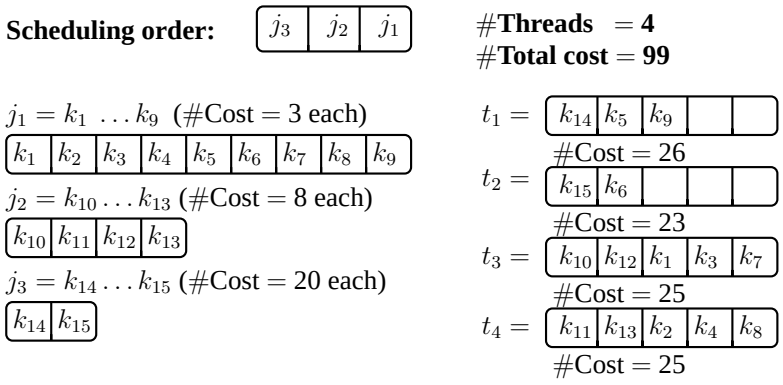


Figure 7: Static scheduling strategy.

Figure 7 exemplifies the static scheduling based implementation, where there are 15 AUNFs $k_1 \ldots k_{15}$ to be distributed among four threads $t_1 \ldots t_4$. All AUNFs of each tensor product term j have the same computational cost. After ordering all the tasks, those of the term j_3 having the highest costs are distributed one by one for the least loaded thread, proceeding to the tasks of the term j_2, and so on.

Algorithm 3 introduces the third implementation that performs a partitioning

per AUNF to achieve better load balance. The algorithm starts by creating a parallel region (line 1), which defines the private variables. The tasks that each thread handles are defined by two indices, *start* and *end*, stored in the B structure (line 4) which is filled through an algorithm that implements the strategy illustrated by Figure 7. Each thread reads the indices of its tasks through its identifier, called *tid*. The value stored in the variable *tid* corresponds to the thread number returned by the function *omp_get_thread_num*, available in the OpenMP library. Note that B is filled in a preprocessing step, which was not considered in the experiments presented in Section 5 due to the negligible overhead of this operation.

Algorithm 3: **AUNF-Man** - *Partitioning per* AUNF k

```
1  #pragma omp parallel private(j,k,tid,υ)
2  begin
3      tid = omp_get_thread_num()
4      for j ∈ [1...T] do
5          for k ∈ [B[tid].term[j].start..B[tid].term[j].end] do
6              υ = A[j].scalar[k] × π_0
7              ...
8              #pragma omp atomic
9              π += υ
10 end
```

5. Performance Evaluation

This section presents an evaluation of the three parallel implementations of the Split algorithm (Section 4.2), considering analysis of speed-up, synchronization and scheduling overhead, memory affinity, and task mapping policies. Moreover, it describes a strategy for automatically selecting the best implementation for each Markovian model.

We prioritize the use of examples with large state spaces reaching the limit for machines with 4GB or even 8GB of RAM to demonstrate how the parallelization improves the model solution in overall. Note that there is a natural increase in terms of solution power using parallelism because one could potentially store in modern machines even bigger auxiliary vectors. We point out that our aim is to look at the time spent for each iteration in the VDP using different partitioning approaches so the gains are replicated in all numerical method iterations needed.

5.1. Environment Setup

We have performed experiments in a shared-memory machine composed of two Intel Xeon E5520 (Nehalem) Quad-Core processors with Intel Hyper-Threading technology (totalizing 16 logical processors) and 16 GB of memory. This machine is a Non-Uniform Memory Access (NUMA) system [21], where each processor access its local memory and with a higher cost the remote memory through the Intel Quick Path Interconnect (QPI). Each processor runs at 2.27 GHz frequency, 8 MB L3 cache shared by all cores, 1 MB L2 cache and 128 KB L1 cache per core. The software stack is a Linux OS with g++ 4.2.4 compiler that implements the OpenMP version 2.5. All implementations were compiled using the compiler optimization flag *-O3*.

Additionally, the experiments were performed using the interleaving mode from NUMA API [22] via the *numactl* Linux command. The interleaving memory allocation policy [22] is commonly used to improve memory access performance for bandwidth and its impact in our experiments is discussed in Section 5.4. Furthermore, to avoid thread migration overheads and core resource sharing between threads (for less than 16 threads) we have performed thread binding via *sched_setaffinity* routine from GNU C library.

5.2. Metrics, Models, and Algorithms

The experiments consider four models and three input sizes for each model. A detailed description of each model is presented in Section 3, i.e., the variation on the input sizes follow the number of tensor product terms in each descriptor, which is based mainly on the number of synchronizing events present in each model. The main difference between the models is heterogeneity and the number of tasks involved in the computation. The models RS, SDT, ASP, and MSA (described in Section 3) have different number of tensor product terms in each defined input size, thus determining the number of tasks included on each partitioning approach. Additionally, we subdivided the models based on their task patterns: *homogeneous tasks*, *mixed tasks*, and *heterogeneous tasks*. The task pattern named as *homogeneous tasks* is related to those task sets where tasks have the same (or very similar) computational costs in terms of floating-point multiplications. In the *mixed tasks* pattern, we have a set of tasks with equal costs and other individual tasks with different costs. Finally, *heterogeneous tasks* indicate that no matter which partitioning was used (per tensor product term, i.e., *Coarse-grain*; per AUNF, i.e., *Fine-grain*), each task has a different computational cost in comparison to others. The main characteristics of each model and their analyzed task types can be seen in Table 5.

In Table 5, the column *Coarse-grain* presents the number of product tensor terms that are present in each model descriptor for all configurations discussed in Section 3. We also present the number of generated AUNFs in the column that refers to the *Fine-grain* tasks.

Table 5: Model classification based on its task costs and a description of the number of tasks for each granularity. The models differ in terms of number of tasks and their computational costs that are equal for the coarse and fine granularity, i.e., the smallest RS model has 44 coarse-grained tasks that have a computational cost in terms of multiplications equals to 369,098,752. In the fine granularity the RS smallest input size has 176 tasks with the same computational costs.

Task type	**Model**	**Coarse-grain**			**Fine-grain**		
		Small	**Medium**	**Large**	**Small**	**Medium**	**Large**
Homogeneous	RS	44	44	48	176	704	384
Mixed	SDT	43	46	49	72	77	79
	ASP	10	17	21	$\approx$68 x 10^3	$\approx$337 x 10^3	$\approx$698 x 10^3
Heterogeneous	MSA	43	51	59	$\approx$16 x 10^6	$\approx$38 x 10^6	$\approx$72 x 10^6

We have executed each model for 2, 4, 8 and 16 threads to obtain their speedup. For each experiment, we have computed the standard deviation and the speedup based on the execution time of the Split algorithm measured over five hundred iterations of the Power method. For fair comparison reasons we used this value even for models that converge in less iterations. We evaluated three parallel implementations, namely **TP-Dyn** (Algorithm 1, where we followed a partitioning per tensor product term (coarse-grained tasks) following a dynamic scheduling strategy), **AUNF-Dyn** (Algorithm 2, by partitioning per AUNF (fine-grained tasks)) following a dynamic scheduling strategy, and **AUNF-Man** (Algorithm 3, where a manual static scheduling using partitioning per AUNF was conducted). Although we performed profiling analysis during the development of our application, we found that more interesting results are related to OpenMP implementation choice aspects. Therefore, we focus on those aspects in the next sections.

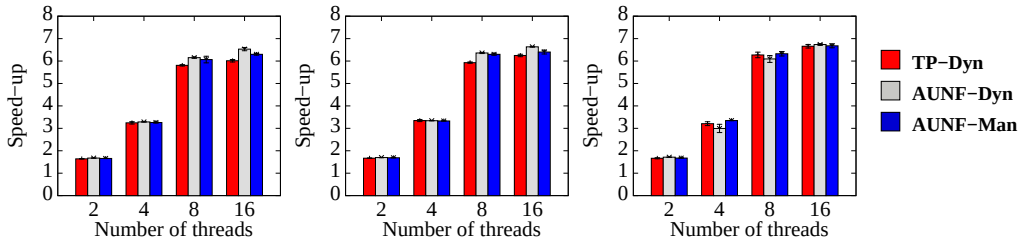
5.3. Results and Analysis

Here, we present the main results of the three OpenMP-based implementations of the Split algorithm for four Markovian models (Section 3). After describing an overview of the results, Sections 5.4 and 5.5 present the impact of the interleaving

policy and the overhead analysis, respectively.

5.3.1. Homogeneous tasks

This section presents the performance results for the RS model. Each input size generates a different number of coarse-grained and fine-grained tasks (Table 5). Moreover, RS is classified as homogeneous-task-type model since it has a set of tasks with the same computational costs for each granularity.



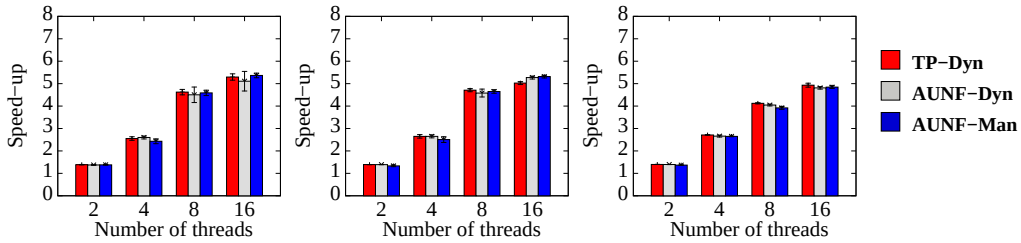
(a) Small. (b) Medium. (c) Large.

Figure 8: Speed-ups for the RS model and three input sizes – *homogeneous-task-type model*.

Figure 8 depicts the speed-up of the three aforementioned implementations. Note that each one has a similar speed-up curve with a maximum difference between the highest and the lowest speed-ups about eight percent (*e.g.*, for the small input size (a)). AUNF-Dyn has the best results for all input sizes, obtaining a speed-up value of up to 6.8. This occurs because AUNF-Dyn has smaller granularity compared to TP-Dyn, which allows OpenMP to have a better scalability.

In addition, different from the AUNF-Man approach, AUNF-Dyn uses dynamic scheduling, which overcomes overheads caused by resources contention during the computation. On the other hand, AUNF-Man has its load balancing strategy based on precomputed theoretical costs, not considering that kind of cost.

One important issue to observe in Figure 8 is that all implementations obtained a better scalability for eight threads than the obtained one for 16 threads. This



(a) Small. (b) Medium. (c) Large.

Figure 9: Speed-ups for the SDT model and three input sizes – *mixed-task-type model*.

issue is confirmed on the other input models as well and its cause is explained by the core resource sharing, which occurs in the experiments for 16 threads where there are two threads running on each core.

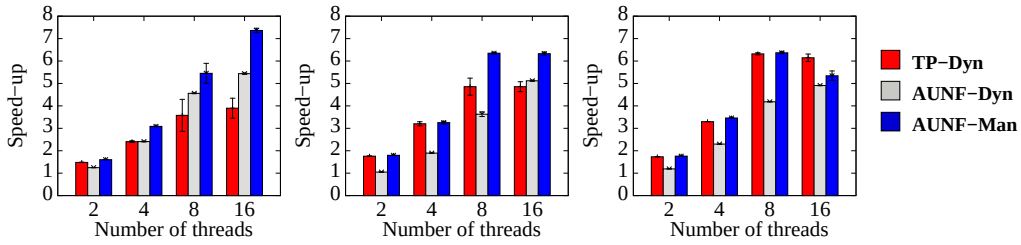
5.3.2. Mixed tasks

This section discusses the performance results for the ASP and SDT models. In a general way, these models have tasks with different computational costs. However, a wide range of tasks has the same computational costs. Therefore, the ASP and SDT models have been classified as mixed-task-type models.

Figure 9 presents the speed-up curve for the SDT model. The three parallel implementations scale up and have a similar speed-up curve for all input sizes. The speed-ups are very similar, because SDT model has a regular number of tasks in both granularities (see Table 5). The number of tasks is enough to scale up until 16 threads. AUNF-Man approach obtains the highest speed-up value up to 5.4. However, the performance gains are lower than those obtained for the RS model. The bottleneck for the SDT model is the synchronization overhead as presented in Section 5.5.2.

Figure 10 presents the speed-up curve for the ASP model. AUNF-Man implementation has a better speed-up of up to 7.4. The small input size (a) has few

tasks to distribute and hence not enough tasks to obtain a good scalability with 16 threads on coarse granularity (TP-Dyn). The performance results of (b) and (c) inputs show that TP-Dyn scales up better, but still does not scale up well with 16 threads. AUNF-Dyn showed the lowest speed-up values because the number of loop iterations (tasks) is large enough to generate an overhead of dynamic scheduling. AUNF-Man works in the same granularity of AUNF-Dyn, but with a static scheduling strategy that does not generate the same overhead as AUNF-Dyn.

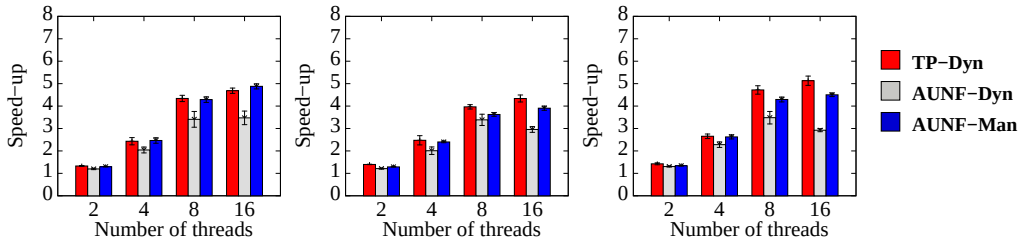


(a) Small. (b) Medium. (c) Large.

Figure 10: Speed-ups for the ASP model and three input sizes – *mixed-task-type model*.

5.3.3. Heterogeneous tasks

The Master-Slave Architecture (MSA) model is composed of a set of heterogeneous tasks on both granularities and a very large number of tasks (Table 5). Figure 11 presents the performance results. AUNF-Dyn did not achieve good speed-up results for the same reason as the ASP model. The number of tasks to distribute across threads is very large and the *schedule(dynamic,1)* clause can produce an considerable overhead in this situation. Furthermore, another parameter that impacts in the dynamic scheduling overhead is the number of threads involved. This can be observed in the experiments for 16 threads where the performance results difference between the implementation AUNF-Dyn and the other ones are higher in comparison to the experiments with a smaller number of threads.



(a) Small. (b) Medium. (c) Large.

Figure 11: Speed-ups for the MSA model and three input sizes – *heterogeneous-task-type model*.

In addition, similar to the SDT model, the MSA model presented a large synchronization overhead which is discussed in Section 5.5.2. Despite this issue, our parallel implementations generated a speed-up value of up to 5.1.

5.4. Impact of the interleaving policy

To improve performance and scalability in NUMA machines, it is important to take into account issues such as memory and thread affinity. Data placement and thread binding become important aspects because local memory access is faster than remote memory access and OpenMP 2.5 has no support for controlling it [8]. There are several techniques that can help optimizing memory access performance for latency or bandwidth. Well-known strategies to perform data placement are *first-touch* and *next-touch* [23], *interleaving policy* [22], among others.

The sequential Split algorithm has a static memory access pattern, i.e., each task accesses the same data during the entire application execution. Therefore, we could reduce memory access latency in the parallel implementations by placing each task into the memory bank of the processor executing it. However, many tasks can read from and write to the same data during the execution, not making it possible to take completely advantage of the local data placement. Furthermore, implementations that use dynamic scheduling normally have an irregular memory

access pattern, which is another reason to not use local data placement strategies.

Therefore, we optimized our parallel Split implementations for bandwidth using an *interleaving policy* [22]. The interleaving memory allocation policy defines that each memory page is assigned in a round-robin fashion over the memory banks. We improved the memory access performance for most of our experiments. Table 6 presents a summary of the improvements in comparison to the default memory allocation policy (local memory allocation).

Table 6: Interleaving mode impact for 16 threads and the medium input size.

Model	*Performance Improvement*		
Medium input size	**TP-Dyn**	**AUNF-Dyn**	**AUNF-Man**
RS	7.2%	24.9%	23.8%
SDT	7.8%	14.1%	17.0%
ASP	0.1%	11.3%	-4.0%
MSA	-2.5%	8.6%	2.2%

The interleaving strategy generated improvements of up to 25%. However, ASP model with TP-Dyn and MSA model with AUNF-Man implementation, had no considerable improvement. On the other hand, the ASP model/AUNF-Man and MSA model/TP-Dyn, we obtained negative results causing a small performance loss. The reason is that each implementation requires a different memory access pattern, accessing the remote memory more than expected.

5.5. Overhead Analysis

Parallel solutions developed via the OpenMP API can have overheads related to the thread management, scheduling clauses, time spent in barriers, among others [8]. This section presents the analysis of two kinds of overhead in OpenMP: dynamic scheduling and synchronization.

5.5.1. *Dynamic scheduling overhead*

Overheads of dynamic scheduling are a well-known drawback in OpenMP [8]. We have performed experiments to show the impact of the *schedule(dynamic,1)* strategy with the increasing number of loop iterations, specially in the range of our Markovian models. Furthermore, a common solution for this problem is to increase the chunk size of the *schedule* clause [8]. However, this solution is not suitable for heterogeneous workloads, leading to unsatisfactory load balance. The *schedule(guided,1)* strategy is a better option that initially defines a large chunk size and at each assigned chunk, decreases its size to 1.

In order to evaluate the scheduling clauses we have developed a benchmark with a loop, which performs a summation. The parallelization is accomplished via the *for* work-sharing construct and all threads update a private variable. Moreover, the *for* construct is not combined with the *parallel* construct to correctly profile the execution time without the influence of thread creation overhead.

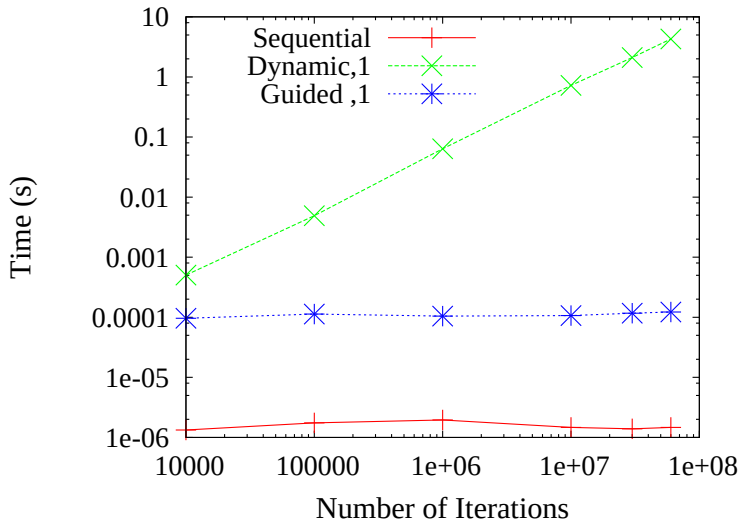


Figure 12: Overhead of dynamic scheduling strategies in function of the number of loop iterations.

Figure 12 presents the processing time in seconds for the sequential benchmark, parallel benchmark using the *schedule(dynamic,1)*, and the *schedule(guided,-1)* clauses. The results demonstrate that the overhead of the *dynamic,1* scheduling strategy is related to the increasing number of loop iterations. With a small number of loop iterations, the generated overhead can be considered low. From the graph, we also observe that the overhead remains the same, even with a high number of iterations, when using the *guided,1* scheduling strategy. This result motivated us to explore such strategy in our algorithms.

Thus, the use of guided scheduling type is a good solution to improve our results from AUNF-Dyn with models ASP and MSA, since they have a large number of loop iterations (tasks) in the fine granularity. However, using the guided scheduling for heterogeneous workloads is not straightforward. Large chunks are initially distributed across the threads, so if the first tasks assigned have high costs and the next tasks assigned have low costs, some threads may become overloaded, causing load imbalance. We have performed experiments with the use of *schedule(guided,1)* clause in the AUNF-Dyn implementation by sorting the tasks based on their computational costs in ascending and descending order.

Table 7: Comparison between *dynamic,1* and *guided,1* scheduling speed-up values with different task ordering strategies for 16 threads.

Model	**Input Size**	**dynamic,1**	**guided,1**	
			Ascending order	*Descending order*
ASP	*Small*	5.44	7.92	5.72
	Medium	5.12	6.97	5.60
	Large	4.91	6.79	5.56
MSA	*Small*	3.47	4.27	1.13
	Medium	2.96	4.49	1.12
	Large	2.92	4.75	1.18

Table 7 presents the performance results for the guided scheduling type with different task ordering strategies. The results show how our application is influenced by overheads of dynamic scheduling. Moreover, sorting the tasks in descending order of computational cost causes a considerable performance loss for the MSA model. The same did not occur with the ASP model because it has less heterogeneous tasks than MSA, minimizing load imbalance effects. In addition, models ASP and MSA are more affected by overheads of dynamic scheduling because the number of loop iterations in AUNF-Dyn implementation is considerably higher compared to the other models.

5.5.2. Synchronization overhead

In order to measure overhead in a shared-memory parallel implementation, one can make a comparison between the time spent to execute the sequential program against the time spent to execute the parallel program using 1 thread. Here we measured the synchronization overhead [24] by executing the parallel implementation of TP-Dyn using one thread with and without the *atomic* clause (without any memory affinity optimization). Figure 13 presents the percentage of overhead computed for all Markovian models presented in Section 3.

From Figure 13 we observe the high synchronization overhead in the execution for the models MSA and SDT compared to the RS and ASP models, where we obtained better performance results. In order to verify the cause of the high overhead, we computed the number of accesses performed to atomic regions. The results from Table 8 show that there is no relation between the number of accesses to atomic regions with the level of overhead measured.

As the execution occurs with only one thread, the reason is not related to common concurrency issues, such as race conditions or cache coherency problems.

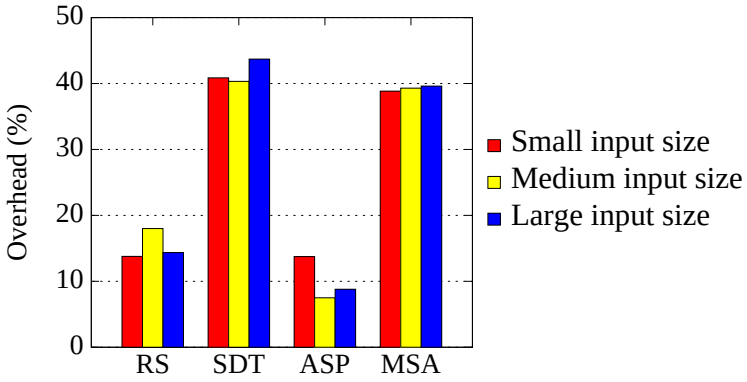


Figure 13: Atomic construct overhead.

Table 8: Relation between the increasing number of atomic region accesses and overhead.

Model	Atomic region accesses	Overhead
SDT large input size	5,682 x 10^6	$\approx 44\%$
ASP large input size	2,568 x 10^6	$\approx 9\%$
SDT small input size	554 x 10^6	$\approx 41\%$
ASP small input size	329 x 10^6	$\approx 14\%$

During our experiments, we observed that the overheads actually occur in a specific part of the algorithm, where atomic updates are performed to a entire vector in a high number of iterations. Additionally, the problem was observed even using other compilers and machines, for instance. Therefore, from our experiments we believe the synchronization overhead can come from a specific Linux kernel or compiler issue regarding lock management.

5.6. Automatic strategy choice

Although, we could use guided scheduling with task ordering (ascending order of task size) for models with large number of tasks and dynamic scheduling for models with small number of tasks, the results produced by each implementation are highly dependent on other model characteristics and system architecture, such

as memory access pattern and synchronization overheads. Therefore, there is no implementation that is able to produce the best results for all models. In order to solve this problem, we have measured the performance gains of each parallel implementation at the beginning of the numerical method to know what it is the best implementation for each case. Thus, we have computed the speed-up for five iterations and five hundred iterations for all implementations and inputs.

Table 9 presents the results of the experiment. As we observed that for most of the results, the best implementation for five iterations is also the best (numbers in bold) implementation for five hundred iterations for all the Markovian models. When the results did not match, the difference between the speed-ups is minimal, which means either implementations can be used.

Table 9: Comparison between the speed-up values for 16 threads obtained in five iterations and five hundred iterations for the three implementations and the Markovian models. The obtained results show that is often possible to know what is the best implementation just looking at few iterations. The values of the table represent the results based on the best implementations.

Model size	# iterations	RS model			SDT model			ASP model			MSA model		
		TP-Dyn	**AUNF-Dyn**	**AUNF-Man**	**TP-Dyn**	**AUNF-Dyn**	**AUNF-Man**	**TP-Dyn**	**AUNF-Dyn**	**AUNF-Man**	**TP-Dyn**	**AUNF-Dyn**	**AUNF-Man**
Small	5	6.01	**6.47**	6.30	5.20	**5.30**	5.24	3.93	**8.01**	7.14	4.52	4.27	**4.91**
	500	6.01	**6.53**	6.30	5.30	5.11	**5.37**	3.90	**7.92**	7.40	4.69	4.27	**4.88**
Medium	5	6.28	**6.64**	6.50	5.07	5.25	**5.28**	4.79	**7.17**	6.39	4.33	**4.70**	3.92
	500	6.25	**6.63**	6.40	5.02	5.27	**5.31**	4.86	**6.97**	6.33	4.34	**4.49**	3.90
Large	5	6.46	6.79	**6.88**	4.90	4.96	**5.02**	6.12	**6.80**	5.40	**4.73**	4.59	4.54
	500	6.66	**6.74**	6.68	**4.93**	4.81	4.85	6.15	**6.79**	5.34	**5.13**	4.75	4.50

There is a cost associated to select the best implementation. As we can execute each implementation sequentially, without necessarily restarting the numerical method, we can keep the last computed results and resume execution after choosing the best implementation. The cost is basically the summation of the

execution time of the two worst implementations for five iterations minus the execution time of the best one for ten iterations.

Table 10: Cost details of the automatic strategy choice for the medium input size, considering an entire execution. **Cost** is the ratio of the total execution time using the automatic strategy choice to the total execution time with the best implementation.

Model *(Medium)*	**Total of iterations**	**Total (s) (Best impl.)**	*First 15 iterations* **TP-Dyn**	**AUNF-Dyn**	**AUNF-Man**	*Remaining iter.* **Chosen impl.**	**Total (s)**	**Cost**
RS	131	≈ 935	37.85	**35.68**	36.96	827.69	≈ 938	≈ 0.37%
SDT	71,057	≈ 348,291	25.92	24.69	**24.51**	348,218	≈ 348,293	≈ 0.0005%
ASP	987	≈ 6,756	49.08	**34.22**	37.68	6,653	≈ 6,774	≈ 0.27%
MSA	10,160	≈ 98,258	50.03	**48.36**	55.67	98,112	≈ 98,267	≈ 0.009%

Table 10 summarizes the cost for finding the best implementation in relation to an entire execution. The cost is higher when the difference of the execution time of the best implementation compared to the other ones is higher as well. However, for ASP and MSA models the cost is diminished by the large number of Power method iterations. In this sense, although the number of iterations for the RS model is small, the cost of the automatic strategy choice is also very small.

6. Related Work

The related work for this paper comes from two research areas: large Markovian system solvers and performance evaluation of OpenMP-based programs. This section provides an overview of research projects from these two areas.

6.1. Parallel solutions of Markovian systems

Parallel algorithms for solving large and sparse Markovian systems only require data loading into processors before starting computation. However, Kronecker-based algorithms introduced data dependency and locality that must be analyzed

prior to the data loading and execution. This is required because these solutions are iterative and their convergence control demands explicitly synchronizing tasks.

Da Cunha and Hopkins [25] considered the basic GMRES iteration with the Arnoldi process. Nevertheless, the work was based on Markov Chains with the state space explosion problem since it makes it difficult for modeling and solving on parallel systems. Erhel [26] proposed a parallel implementation of Arnoldi and GMRES methods using the Single Program Multiple Data (SPMD) programming style. Gimenez et al. [27] developed a parallel implementation for the Power Method for solving linear equations obtained through Markov Chains models.

Tadonki and Philippe [28] have proposed a recursive version for the parallel multiplication of a vector by a product of matrices, in contrast to our approach that multiplies a vector by a descriptor. In the context of continuous time Markov chains, Kemper [29] has modified the Kronecker representation for a parallel matrix-vector multiplication. His implementation, based on POSIX threads, uses a fast multiplication scheme with no write conflicts on iteration vectors.

Deavours and Sanders [30] devised a method to efficiently store a Markovian transition matrix on disk, thus overlapping computation and data transferring on a standard workstation. They use two processes that communicate via shared memory, efficiently utilizing the system disk and CPU. Knottenbelt and Harrison [31] proposed a distributed software architecture to embed the matrix-vector multiplication solution algorithm, allowing two processes per core, and achieving good speed-ups for models up to 50 million states. Bell and Haverkort [32] presented distributed disk-based algorithms for matrix-vector multiplications in the context of CSL model checking-based performance. Results illustrate the effectiveness of the approach proposed for models with several hundreds of million states running

on a cluster with 26 dual-processor nodes.

Kwiatkowska et al. [33] mixed parallel and symbolic techniques to tackle the state space explosion problem proposing an out-of-core solution to matrix-vector multiplication for models near 216 million states. Dingle et al. [34] investigated hypergraph partitioning schemes to minimize inter-processes communication when applying a uniformization-based technique to derive response time densities for large models. The authors showed results for Generalized Stochastic Petri Nets [35] and flat representations of Markov chains.

Blom et al. [36] used a bisimulation approach to consider stochastic model variants to enable the model checking of smaller and probabilistic equivalent models. Bisimulation techniques usually allow substantial model reduction in terms of state space size, however, the gains in storage affect the time to solve the models. The authors present a distributed signature-based algorithm for the bisimulation quotient and demonstrate the feasibility for a broad variation of case studies.

Previous works have addressed iterative methods, disk-based approaches, bisimulation, model checking, all for matrix-vector multiplications in parallel (or distributed) settings. Our approach is to consider VDP instead of matrix-vector products focusing on the speed-up of the overall process by partitioning descriptors into more manageable and scalable tasks. Moreover, we would like to mention that we have studied the trade-offs between commonly used VDP techniques such as Shuffle and Split algorithms. To the best of the authors' knowledge, the most closely related work concerns Kemper [29], however, the author transformed the Kronecker representation into a flat representation that often incurs in high memory costs. The direction taken in the present work differs from Kemper's research as we are working with Kronecker algebra operations. Using Kronecker storage

features combined with parallel mechanisms allows faster solutions even for the models where some extra memory is used (the Split algorithm's case).

6.2. OpenMP-based programs

OpenMP (Open Multi-Processing) [8] is an API for shared memory multi-processing programming. Since then, several researchers and developers have been evaluating the performance of OpenMP-based applications in comparison with other APIs, especially MPI. For example, Krawezik and Cappello [37] compare MPI and three OpenMP programming styles using a subset of the NAS benchmark along with two data set sizes and shared memory processors. The authors concluded that OpenMP provides competitive performance compared to MPI with the price of a strong programming effort. Mallón et al. [38] have also compared MPI and OpenMP and concluded that data locality is one of the main obstacle for obtaining good performance in OpenMP applications.

Mattson [39] developed a framework to evaluate OpenMP considering its main features and possible enhancements for the API. One of his remarks is that compared to other parallel programming APIs, constructs in OpenMP are the most part semantically neutral compiler directives. Therefore, the semantics of a parallel and sequential program are equivalent. And this is one of the main reasons why several programs have been developed in OpenMP.

Several software systems have been then developed in OpenMP and their efficiency evaluated. Bungartz et al. [40] implemented an OpenMP-based Black-Scholes solver, which is used for option pricing, and evaluated it on multi-core architectures. Their experiments mainly considered different number of threads. Terboven et al. [41] presented implementation choices for an OpenMP-based Navier-Stokes Solver and also mainly varied the number of threads in their experiments.

Maris and Wannamaker [42] described modifications made to a 3D magnetotelluric inversion program to run efficiently in parallel on a multi-core desktop PC. Their experiments focused on varying the loop frequency and number of cores used by their program. Different from these existing projects, our work provides a deeper analysis considering different number of threads, overhead, memory affinity, and task mapping policies.

7. Concluding Remarks

This paper presented three parallel implementations of the Split algorithm for handling Kronecker descriptors. Our implementations were developed using OpenMP for shared-memory architectures. We performed extensive experiments using four types of models with three input sizes each. We analyzed speed-up, synchronization and scheduling overheads, task mapping policies, and memory affinity. Our experiments demonstrated a speed-up value of up to eight using eight cores with Intel Hyper-Threading technology. We observed that the choice of the implementation depends on the input size and the model characteristics to be evaluated. Therefore, as the solvers are iterative applications, by executing a few iterations of the three implementations it is possible to automatically select the best one to solve and analyze models.

The differences of the three implementations lay in the task scheduling strategy and task granularity. Two mplementations are based on OpenMP standard and the third one is based on manual static task scheduling. For the model consisting of homogeneous tasks, the dynamic scheduling strategy using fine-grained tasks is more suitable than the static one. The reason is that, for such model, the scheduling overhead using the clause *schedule(dynamic,1)* is negligible due to the small

number of tasks. Moreover, for the model composed of few tasks in the coarse granularity, the scalability of the parallel execution is not as good as in the fine granularity of the same model. This happens because there is not enough work to be distributed among the processors in order to scale it up.

For the models composed of a very large number of tasks, the overhead imposed by the clause *schedule(dynamic,1)* produces negative effects in the speed-up. To minimize the overhead effect, the dynamic scheduling can be changed to the guided one. However, guided scheduling can cause load imbalance without applying any task ordering strategy. Considering that the guided clause initially distributes large chunks of work, it is important to sort tasks by ascending order of their sizes to improve load balance and hence reduce execution time.

Another source of expected overhead is the number of atomic operations. However, from our experiments, we observed that there is no strong relation between the number of atomic operations and the overhead imposed by the use of the atomic construct from OpenMP. Furthermore, applying an interleaving memory allocation policy improves the memory access performance in NUMA machines, mainly in applications that cannot take advantage of the *first-touch* technique. For example, applications which use dynamic scheduling strategies, i.e., the memory access pattern becomes irregular. Regarding code optimization, as future work, we will perform a deeper analysis of memory and cache and the impact of the architecture (i.e., NUMA based systems, different cache levels, and Simultaneous Multithreading technology), and other data partitioning options.

The implementations presented in this paper achieve high performance results, which have a direct impact on the solution of large Markovian models based on Kronecker representations. The discussions presented here could also be used for

researches working in similar programming models. In addition, this paper is another example of the successful use of OpenMP for solving scientific applications.

References

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

[2] M. Bernardo, J. Hillston (Eds.), Formal Methods for Performance Evaluation, SFM 2007, Advanced Lectures, Vol. 4486 of LNCS, Springer, 2007.

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

[4] R. M. Czekster, P. Fernandes, J.-M. Vincent, T. Webber, Split: a flexible and efficient algorithm to vector-descriptor product, in: Proceedings of the 2nd International Conference on Performance Evaluation Methods and Tools (ValueTools 2007), Vol. 321, 2007, pp. 1–8.

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

[6] R. M. Czekster, C. A. F. De Rose, P. Fernandes, A. M. Lima, T. Webber, Kronecker Descriptor Partitioning for Parallel Algorithms, in: Proceedings of the Spring Simulation Multiconference 2010 (SpringSim 2010), no. 242, 2010, pp. 1–4.

[7] W. Gropp, E. Lusk, A. Skjellum, Using MPI: portable parallel programming with the message passing interface, MIT Press, 1999.

[8] B. Chapman, G. Jost, R. van der Pas, Using OpenMP: Portable Shared Memory Parallel Programming, The MIT Press, 2007.

[9] B. Wilkinson, M. Allen, Parallel programming: techniques and applications using networked workstations and parallel computers, Prentice Hall, 1999.

[10] R. Blikberg, T. Sørevik, Load balancing and OpenMP implementation of nested parallelism, Parallel Computing 31 (10-12) (2005) 984–998.

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

[12] L. Brenner, P. Fernandes, 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) (2005) 52–60.

[13] B. Plateau, On the stochastic structure of parallelism and synchronization models for distributed algorithms, ACM SIGMETRICS Performance Evaluation Review 13 (2) (1985) 147–154.

[14] R. M. Czekster, P. Fernandes, T. Webber, Efficient Vector-Descriptor Product Exploiting Time-Memory Trade-offs (accepted), ACM SIGMETRICS Performance Evaluation Review (2012) 1–8.

[15] R. M. Czekster, P. Fernandes, A. Sales, T. Webber, Restructuring tensor products to enhance the numerical solution of structured Markov chains, in: Proceedings of the 6th International Conference on the Numerical Solution of Markov Chains (NSMC '10), 2010, pp. 36–39.

[16] R. M. Czekster, P. Fernandes, T. Webber, GTAexpress: a Software Package to Handle Kronecker Descriptors, in: Proceedings of the 6th International Conference on Quantitative Evaluation of Systems (QEST 2009), IEEE Computer Society, Washington, DC, USA, 2009, pp. 281–282.

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

[18] P. Fernandes, A. Sales, A. R. Santos, T. Webber, Performance Evaluation of Software Development Teams: a Practical Case Study, Electronic Notes in Theoretical Computer Science (ENTCS) 275 (C) (2011) 73–92.

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

[20] E. G. Coffman Jr., M. R. Garey, D. S. Johnson, Approximation algorithms for bin packing: a survey, PWS Publishing Co., Boston, MA, USA, 1997, pp. 46–93.

[21] M. J. Quinn, Parallel Programming in C with MPI and OpenMP, McGrawHill, 2003.

[22] A. Kleen, A NUMA API for Linux, Novell, Inc., technical Whitepaper (2005).

[23] C. Terboven, D. an Mey, D. Schmidl, H. Jin, T. Reichstein, Data and thread affinity in openmp programs, Conference On Computing Frontiers: Proceed-

ings of the 2008 workshop on Memory access on future processors: a solved problem? ACM. 8p, Ischia, Italy, 2008.

[24] J. M. Bull, Measuring synchronization and scheduling overheads in OpenMP, in: European Workshop on OpenMP, 1999.

[25] R. D. da Cunha, T. Hopkins, A parallel implementation of the restarted GMRES iterative algorithm for nonsymmetric systems of linear equations, Advances in Computational Mathematics 2 (3) (1994) 261–277.

[26] J. Erhel, A parallel GMRES version for general sparse matrices, Electronic Transactions on Numerical Analysis 3 (1995) 160–176.

[27] D. Gimnez, C. Jimnez, M. J. Majado, N. Marn, A. Martn, Solving Eigenvalue Problems on Networks of Processors, in: Third International Conference on Vector and Parallel Processing (VECPAR '98), Springer-Verlag (LNCS 1573), 1999, pp. 85–99.

[28] C. Tadonki, B. Philippe, Parallel multiplication of a vector by a kronecker product of matrices, Parallel numerical linear algebra (2001) 71–89.

[29] P. Kemper, Parallel Randomization for Large Structured Markov Chains, in: Proceedings of the International Conference on Dependable Systems and Networks (DSN'02), IEEE Computer Society, Washington, DC, USA, 2002, pp. 657–668.

[30] D. D. Deavours, W. H. Sanders, An Efficient Disk-Based Tool for Solving Large Markov Models, Performance Evaluation 33 (1) (1998) 67–84.

[31] W. J. Knottenbelt, P. G. Harrison, Distributed Disk-based Solution Techniques for Large Markov Models, in: Proceedings of the 3rd International Workshop on the Numerical Solution of Markov Chains (NSMC '99), 1999, pp. 58–75.

[32] A. Bell, B. R. Haverkort, Distributed disk-based algorithms for model checking very large Markov chains, Formal Methods in System Design 29 (2) (2006) 177–196.

[33] M. Kwiatkowska, R. Mehmood, G. Norman, D. Parker, A Symbolic Out-of-Core Solution Method for Markov Models, Electronic Notes in Theoretical Computer Science (ENTCS) 68 (4) (2002) 589–604.

[34] N. J. Dingle, P. G. Harrison, W. J. Knottenbelt, Uniformization and hypergraph partitioning for the distributed computation of response time densities in very large Markov models, Journal of Parallel and Distributed Computing 64 (8) (2004) 908–920.

[35] G. Chiola, A. M. Marsan, G. Balbo, G. Conte, Generalized stochastic Petri nets: A definition at the net level and its implications, IEEE Transactions on Software Engineering 19 (2) (1993) 89–107.

[36] S. Blom, B. R. Haverkort, M. Kuntz, J. van de Pol, Distributed Markovian Bisimulation Reduction aimed at CSL Model Checking, Electronic Notes in Theoretical Computer Science (ENTCS) 220 (2) (2008) 35–50.

[37] G. Krawezik, F. Cappello, Performance comparison of MPI and three OpenMP programming styles on shared memory multiprocessors, in: Pro-

ceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'03), ACM, 2003, pp. 118–127.

[38] D. Mallón, G. Taboada, C. Teijeiro, J. Touriño, B. Fraguela, A. Gómez, R. Doallo, J. Mourino, Performance evaluation of MPI, UPC and OpenMP on multicore architectures, Recent Advances in Parallel Virtual Machine and Message Passing Interface (2009) 174–184.

[39] T. G. Mattson, How good is OpenMP, Scientific Programming 11 (2) (2003) 81–93.

[40] H. J. Bungartz, A. Heinecke, D. Pfluger, S. Schraufstetter, Parallelizing a Black-Scholes solver based on finite elements and sparse grids, in: Proceedings of the IEEE International Symposium on Parallel & Distributed Processing (IPDPS'10), IEEE, 2010, pp. 1–8.

[41] C. Terboven, A. Spiegel, D. an Mey, S. Gross, V. Reichelt, Experiences with the OpenMP Parallelization of DROPS, a Navier-Stokes Solver written in C++, in: Proceedings of the first international workshop on OpenMP (IWOMP 2005), 2005.

[42] V. Maris, P. E. Wannamaker, Parallelizing a 3D finite difference MT inversion algorithm on a multicore PC using OpenMP, Computers & Geosciences 36 (10) (2010) 1384–1387.