2010 22nd International Symposium on Computer Architecture and High Performance Computing Workshops

OpenMP-based Parallel Algorithms for Solving Kronecker Descriptors*

Antonio M. de Lima, Thais Webber, Marco A. S. Netto
Ricardo M. Czekster, Cesar A. F. De Rose, Paulo Fernandes
PUCRS – 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 such as Bioinformatics, Economics, and Engineering. These models can be represented in a compact fashion using Kronecker algebra. The Vector-Descriptor Product is the key operation to obtain stationary solutions of Kronecker-based descriptors. Due to its complexity, the numerical algorithms are usually CPU intensive, requiring alternatives such as data partitioning in order to produce results in less time. This paper proposes three OpenMP-based parallel implementations for solving descriptors to be deployed on shared-memory machines. We evaluated the implementations in a multi-core machine and obtained a speed-up near to eight when using eight cores with Intel Hyper-Threading technology.

1. Introduction

Kronecker descriptors [15] are compact structures commonly used to describe very large Markovian systems. A myriad of structured formalisms [2] that use Kronecker (tensor) algebra as a compact representation is available to the research community, *e.g.*, Stochastic Petri Nets (SPN), Process Algebra (PEPA), and Stochastic Automata Networks (SAN) [12], among others.

Vector-Descriptor Product (VDP) is the key operation to achieve a stationary regime and subsequent performance indices of systems represented by descriptors. This operation multiplies a probability vector by a descriptor, which is composed of tensor product terms [12]. Each term corresponds to a set of small matrices and tensor product operators. Specialized numerical algorithms have been proposed throughout the years, namely the traditional *Shuffle algorithm* [11, 12] and the flexible *Split algorithm* [10]. The main difference between them concerns the additional memory requirements and the computational cost in terms of floating-point multiplications.

The processing power and storage of the current computing resources have enabled the evaluation of Markovian models with large state spaces; which are required in several fields from science and engineering. However, as numerical solution consists of an iterative process, the step named VDP is repeated several times [15] (*e.g.* Power method, Arnoldi, and GMRES). Moreover, with the increasing complexity related to the size of the models, the processing time becomes considerably high. Therefore, as most of the current machines are based on multi-core technology, the creation of parallel solutions to accelerate the achievement of results becomes essential.

We have developed a parallel solution of Kronecker Descriptors [8], which considered data partitioning strategies for the Split Algorithm in a cluster architecture. However, the parallel approach using MPI [13] routines presented a low scalability, mainly due to the vector update operation at each iteration. Tadonki and Philippe [16] have proposed a parallel multiplication of a vector by a Kronecker product of matrices, which differs from our work since we multiply a vector by a Kronecker descriptor. In the context of continuous time Markov chains, Kemper [14] has modified the Kronecker representation for a parallel matrix-vector multiplication. His implementation, based on POSIX threads, uses a fast multiplication scheme and no write conflicts on iteration vectors. Kemper obtained a speed-up of approximately five using eight processors.

In order to achieve better speed-ups in comparison to existing solutions, this paper proposes and evaluates three parallel implementations of the Split algorithm for shared-memory machines. These implementations are based on different scheduling strategies using OpenMP (OpenMP Multi-Processing) [6]. We have developed the parallel implementations using existing techniques of static and dynamic load balancing and data partitioning presented in the literature [17, 3]. From our experimental results, we have obtained a speed-up near to eight, using eight cores with Intel Hyper-Threading technology.

*This work is partially supported by grants from Petrobras (0050.0048664.09.9) and CNPq-Brazil (307272/2007-9).

978-0-7695-4276-8/10 $26.00 © 2010 IEEE
DOI 10.1109/SBAC-PADW.2010.13

2. Split Algorithm

The Split algorithm is known as a Vector-Descriptor Product algorithm [9, 10, 12] for solving Kronecker based descriptors. Basically, the method performs the multiplication of a probability vector (state space sized) by sets of matrices that compose a generic tensor product term of N matrices [11, 12]. This process is repeated for all T tensor product terms in a descriptor to complete a step in the iterative numerical method (*e.g.* Power method).

The algorithm defines a cut-parameter σ which separates the tensor product term in two different sets of matrices [9]. The first selected set is treated in a sparse-like manner (matrices aggregation) performing non-zero elements combinations. Each combination done in this part is called AUNF (*Additive Unitary Normal Factor*) or simply *scalar*. This scalar has a given value and indices to access determined positions in the probability vectors during the VDP operation. A contiguous slice of the input vector is taken to be multiplied by this scalar. The second selected set is composed of the remaining matrices of the tensor product term, and they are treated with shuffling operations, maintaining the original tensor structure (named shuffle-like part).

Figure 1 illustrates the operation flow in a sequential implementation of the Split algorithm. The first steps (a) and (b) are executed to initially setup the method loading both matrices and vectors into the memory. The step (c) referred to the creation of AUNFs (scalars) using the first set of matrices. These non-zero values are stored in memory with information about their related indices in the vector. Steps (d) and (e) are correspondent to the processing core.

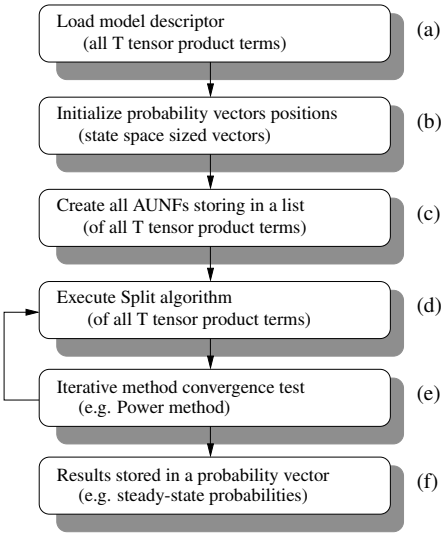


Figure 1. Sequential VDP method.

The execution of the iterative method in an efficient manner is dependable of several factors: the size of the analyzed model, the number of matrices represented in a Kronecker fashion, the computational cost related to the sparsity of these matrices, and the behavior adopted by Split setting the cut parameters for each tensor product term. More details about the numerical methods for Vector-descriptor Product can be found in the literature [12, 10, 9].

3. Descriptor Partitioning

3.1 Partitioning per 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 available 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 j-th term 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. As presented, the cost of each tensor product term is defined mainly by the number of AUNFs 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 balancing and scalability of the parallel solution.

3.2 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 computation of each AUNF represents an independent task, where a slice of the probability vector is multiplied for AUNF, and then the result is accumulated in a probability vector. The total number of AUNFs per term 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. In comparison to the previous approach described in Section 3.1, we have a assembled a number of tasks possibly larger with lower computational costs, thus enabling better load balancing and scalability.

4. Parallel Implementation

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

At the beginning of each iteration of the numerical method, a parallel region is created. The Split algorithm works with loops for the solution of each tensor product term and each AUNF. 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 [6].

4.1 OpenMP-based scheduling

The first two parallel implementations use the *for* worksharing 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 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 [6]. Moreover, the *dynamic* schedule can be more suitable to unbalanced workloads. Algorithm 1 presents the first parallel implementation that uses partitioning per term.

Algorithm 1, 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 terms to be distributed among NT 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 vector π. 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 parallel block occurs after line 7. Algorithm 2 presents the second implementation that uses partitioning per AUNF.

Algorithm 1: OMPa - Partitioning per term j

```
1 #pragma omp parallel for private(j,k,υ) schedule(dynamic,1)
  num_threads(NT)
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 works in a similar way to Algorithm 1. However, to perform the partitioning per AUNF, 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.

Algorithm 2: OMPb - Partitioning per AUNF k

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

4.2 Manual static scheduling

As the *schedule(static)* clause 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 [7]. The main idea of this strategy is to sort the tasks in descending order based on the computational costs of each task and then schedule one by one, beginning from the least loaded thread. Figure 2 presents an example.

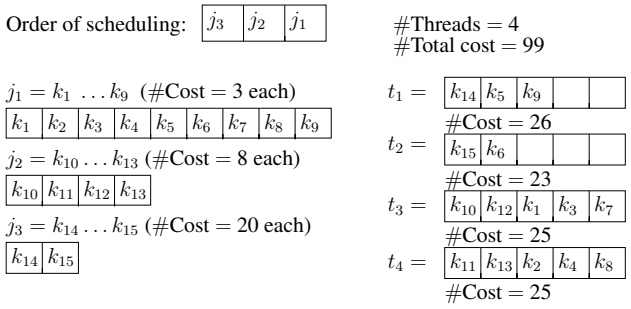


Figure 2. Static scheduling strategy.

Figure 2 exemplifies the static scheduling based implementation, where there are fifteen 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, the first tasks of the term j_3, which have the highest costs, are distributed one by one for the less loaded thread, and second the tasks of the term j_2 do the same, and so on.

Algorithm 3 introduces the third implementation that performs a partitioning per AUNF in order to achieve better load balancing. The algorithm starts by creating a parallel region (line 1), which defines the number of threads and the private variables. The tasks that each thread handle are defined by two indices, *start* and *end*, stored in the B structure (line 4). 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.

Algorithm 3: Manual - Partitioning per AUNF k

```
1 #pragma omp parallel private(j,k,tid,v) num_threads(NT)
2 tid = omp_get_thread_num()
3 for j ∈ [1 . . . T] do
4     for k ∈ [B[tid].term[j].start . . . B[tid].term[j].end] do
5         v = A[j].scalar[k] × π0
6         . . .
7         #pragma omp atomic
8         π += v
```

5. Performance Analysis

We have performed experiments in a shared-memory machine composed of two Intel Xeon E5520 (Nehalem) Quad-Core processors with Intel Hyper-Threading technology and 16 GB of main memory. Each processor works with 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 O.S. with g++ 4.2.4 compiler that implements the OpenMP 2.5 version.

The experiments consider two types of models and two input sizes for each model. The main difference between the models is heterogeneity and number of tasks involved in the computation. Each model requires a hundred iterations of the numerical method to understand its behavior. We have executed each model five times for 2, 4, 8 and 16 threads in order to obtain their speed-up. In addition, we have collected data related to the models, such as number of tasks and average costs of processing each task.

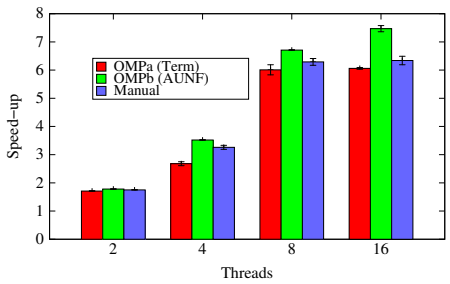
5.1 Resource Sharing model

The classical *Resource Sharing model* maps R shared resources to P processes. The correspondent Markovian model [4] presents each process with two states: *idle* or *busy*. The number of available resources is represented by a function that only grants access to the *busy* state if there is less than R processes in the *busy* state. The descriptors analyzed are composed of more than 40 tensor product terms (for $P = 22$ and $R = 14$; and $P = 24$ and $R = 18$) to be multiplied by a probability vector using Split algorithm, in an iterative process. The details of the input parameters for this model are presented in Table 1. For space constraint reasons, we present only two input sizes, one medium ($P = 22$ and $R = 14$) and one large ($P = 24$ and $R = 18$). For a small size model, we believe a user would run it sequentially on a single processor.

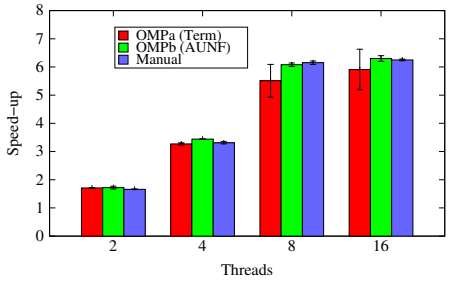
Figure 3 (a) depicts the speed-up of the three implementations: OMPa (Algorithm 1), OMPb (Algorithm 2), and Manual (Algorithm 3). Although the three implementations have a similar speed-up curve, OMPb has a better speed-up, obtaining up to 7.50. This occurs because OMPb uses dynamic scheduling, different from the Manual approach, and has smaller granularity compared to OMPa, which allows OpenMP to have a better load balancing and scalability.

Table 1. Resource Sharing model parameters.

Medium model size: P=22, R=14		
Tensor Product Terms j	**Total of AUNFs**	**Total of Mults**
1 .. 44	616	1,291,845,632
Size of vector π		62,914,560
Average Time per iteration (seconds)		43.25
Total iterations		148
Total time spent (hours)		1.78
Large model size: P=24, R=18		
Tensor Product Terms j	**Total of AUNFs**	**Total of Mults**
1 .. 48	864	7,247,757,312
Size of vector π		318,767,104
Average Time per iteration (seconds)		235.54
Total iterations		N/A
Total time spent (days)		> 15



(a) Medium input size.



(b) Large input size.

Figure 3. Resource Sharing model.

Figure 3 (b) presents the speed-up with a large input size ($P = 24$ and $R = 18$). In this case, the three implementations still scale up and their speed-up curves are similar. However, different from the previous model size, the speed-up values of the three implementations are approximately the same. The reason is that the task cost is four times bigger than the previous model (Table 1). The higher the cost the longer the processors remain working, thus making the task distribution strategy used less important.

5.2 Master-slave model

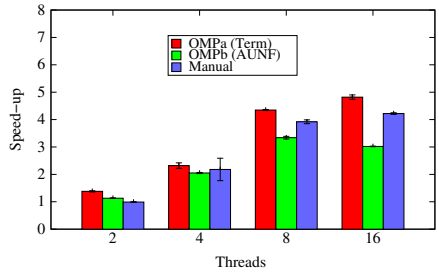
The *Master-slave model* refers to an evaluation of the Master-slave parallel implementation of the Propagation algorithm considering asynchronous communication. The Markovian model [1] contains a *Master* automaton, one huge *Buffer* automaton, and N *Slaves*. The *Master* presents three states: Tx (transmitting), Rx (receiving) and ITx (idle). The *Buffer* has K positions (states) plus an empty state 0. A *Slave* presents three states: I (idle), Pr (processing) and Tx (transmitting). The model has events related to the *Master* and *Slaves* coordinated activities controlling also the *Buffer*, which is also accessed by the slaves. The descriptors analyzed in this section are composed of more than 50 tensor product terms (for $N = 12$ and $K = 60$; and $N = 14$ and $K = 12$) to be multiplied by a probability vector also using Split algorithm.

Table 2. Master-slave model parameters.

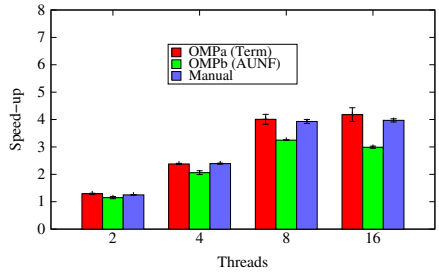
Medium model size: N=12, K=60		
Tensor Product Terms j	**Total of AUNFs**	**Total of Mults**
1 .. 12	12	389,014,812
13 .. 24	12	129,671,604
25 .. 25	1	61
26 .. 26	32,417,901	32,417,901
27 .. 27	245,760	245,760
28 .. 39	720	127,545,840
40 .. 51	1,440	765,275,040
Total AUNFs for all terms		32,665,846
Size of vector π		97,253,703
Average time per iteration (seconds)		37.61
Total iterations		3,969
Total time spent (hours)		41.46
Large model size: N=14, K=12		
Tensor Product Terms j	**Total of AUNFs**	**Total of Mults**
1 .. 14	14	803,538,792
15 .. 28	14	267,846,264
29 .. 29	1	12
30 .. 30	57,395,628	57,395,628
31 .. 31	180,224	180,224
32 .. 45	154	22,320,522
46 .. 59	308	66,961,566
Total AUNFs for all terms		57,576,343
Size of vector π		172,186,884
Average time per iteration (seconds)		71.60
Total iterations		2,260
Total time spent (hours)		44.95

Figure 4 (a) presents the speed-up curve for the medium model size. Although task heterogeneity makes task scheduling more complex, our parallel implementations generated speed-up of up to 4.80, which is approximately 80% of improvement compared to the sequential version. Moreover, OMPb did not perform as good as in the previous model. One of the reasons is that the *schedule(dynamic,1)* clause produces an overhead when the number of iterations to distribute tasks is large. The number of tasks to distribute across the threads is considerably larger in OMPb compared to OMPa (32,665,846 vs 51, see Table 2). We are currently investigating the use of task ordering, which may improve speed-up factors especially when the number of tasks and their heterogeneity is high. The manual static task distribution eliminates the overhead of the dynamic scheduling, resulting in a better speed-up compared to OMPb. Although one could use the *schedule(static)* clause, such a clause does not consider the task heterogeneity imposed by this second model, proving then poor performance gains.

The results for the large model size ($N = 14$ and $K = 12$), presented in Figure 4 (b), are similar to the medium model size. The main difference is that OMPa and Manual produce similar speed-up values due to the same reason presented in the previous model; *i.e.* the higher the computational cost to process tasks and the number of tasks, the less important the task distribution strategy.



(a) Medium input size.



(b) Large input size.

Figure 4. Master-slave model.

5.3 Synchronization overhead analysis

Parallel solutions developed via the OpenMP API could have overheads related to the thread management, scheduling clauses, time spent in barriers, among others [6]. In order to measure overheads in the 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 [5] by executing the parallel implementation of OMPa using one thread with and without the *atomic* clause.

From Table 3 we observe the higher overheads in Master-slave model compared to the Resource Sharing model. In order to verify the cause of the high overheads, we computed the number of accesses to atomic regions.

Table 3. Number of atomic region accesses.

Model	Overhead (%)	Accesses (omp atomic)
RS large input size	15.99	7,247,757,312
MS large input size	49.77	4,454,718,698
MS medium input size	40.88	2,222,200,642
RS medium input size	14.51	1,291,845,632

The results show that there is no relation between the number of accesses to atomic regions with the level of overhead measured. During our experiments, we observed that the overheads are actually due to the number of accesses to the *same* vector positions, *i.e.* the access pattern is the main overhead cause.

6. Concluding Remarks

This paper presented three parallel implementations of Vector-Descriptor Product using the Split algorithm. Our implementations were developed using OpenMP for shared-memory architectures. We performed experiments using two types of models with two input sizes each. From our experiments, we obtained a speed-up value of up to eight using eight cores with Intel Hyper-Threading technology.

The differences of the three implementations lay in the task scheduling strategy and task granularity. Two of the presented implementations 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-grain tasks is more suitable than the static one. The reason is that the number of the tasks is minimal to generate an overhead using the clause *schedule(dynamic)*.

On other hand, for the second model consisting of heterogeneous tasks and a large number of iterations, the overhead imposed by the clause *schedule(dynamic)* produces negative effects in the speed-up. To minimize the overhead effect, two approaches can be used: increase the granularity of tasks keeping the dynamic scheduling clause, or use the same task granularity, but with a manual static scheduling. Another source of overhead found in our experiments is the use of the atomic clause, which depending on the usage pattern, may have a considerable impact on the execution time.

The implementations presented in this paper, thus achieve faster results compared to previous solutions, which have a direct impact on large Markov models based in Kronecker representation. These models are common in several fields such as Bioinformatics, performance prediction of systems, as well as economics and finance applications.

References

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

[2] M. Bernardo and J. Hillston, editors. *Formal Methods for Performance Evaluation, SFM 2007, Advanced Lectures*, volume 4486 of *LNCS*. Springer, 2007.

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

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

[5] J. Bull. Measuring synchronization and scheduling overheads in OpenMP. In *European Workshop on OpenMP*, 1999.

[6] B. Chapman, G. Jost, and R. Van Der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2007.

[7] E. Coffman Jr, M. Garey, and D. Johnson. Approximation algorithms for bin packing: A survey. In *Approximation algorithms for NP-hard problems*, 1996.

[8] R. M. Czekster, C. A. F. De Rose, P. Fernandes, A. M. Lima, and T. Webber. Kronecker Descriptor Partitioning for Parallel Algorithms. In *Proc. of the Spring Sim. Multiconf. 2010 (SpringSim 2010)*, pages 1–4, 2010.

[9] R. M. Czekster, P. Fernandes, A. Sales, and T. Webber. Restructuring tensor products to enhance the numerical solution of structured Markov chains. In *Proc. of the 6th Int. Conf. on the Num. Sol. of Markov Chains (NSMC '10)*, pages 1–4, September 2010.

[10] R. M. Czekster, P. Fernandes, J.-M. Vincent, and T. Webber. Split: a flexible and efficient algorithm to vector-descriptor product. In *Proc. of the 2nd Int. Conf. on Perf. Eval. Meth. and Tools (ValueTools 2007)*, volume 321, pages 1–8, 2007.

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

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

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

[14] P. Kemper. Parallel randomization for large structured markov chains. In *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN'02)*, 2002.

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

[16] C. Tadonki and B. Philippe. Parallel multiplication of a vector by a kronecker product of matrices. pages 71–89, 2001.

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