Scalable Computing: Practice and Experience
Volume 7, Number 2, pp. 53–63. http://www.scpe.org
ISSN 1895-1767

PARALLEL IMPLEMENTATION OF UNIFORMIZATION TO COMPUTE THE TRANSIENT SOLUTION OF STOCHASTIC AUTOMATA NETWORKS

HAÏSCAM ABDALLAH*

Abstract. Analysis of Stochastic Automata Networks (SAN) is a well established approach for modeling the behaviour of computing networks and systems, particularly parallel systems. The transient study of performance measures leads us to time and space complexity problems as well as error control of the numerical results. The SAN theory presents some advantages such as avoiding to build the entire infinitesimal generator and facing the time complexity problem thanks to the tensor algebra properties.

The aim of this study is the computation of the transient state probability vector of SAN models. We first select and modify the (stable) uniformization method in order to compute that vector in a sequential way. We also propose a new efficient algorithm to compute a product of a vector by a tensor sum of matrices. Then, we study the contribution of parallelism in front of the increasing execution time for stiff models by developing a parallel algorithm of the uniformization. The latter algorithm is efficient and allows to process, within a fair computing time, systems with more than one million states and large mission time values.

Key words. Parallel systems, stochastic automata networks, transient solution, uniformization, parallelism.

1. Introduction. This paper presents a parallel version of the *transient analysis* for Continuous Time Markov Chains (CTMCs) via Stochastic Automata Networks (SANs). The computation of the transient distribution of CTMC gives the main performance measures such as reliability and availability. Generally, we are facing the problem of computation time due to the explosive growth of the state space and the stiffness. SANs, introduced by Brigitte Plateau [1], may be a good solution to that problem.

The use of SANs is becoming important in performance modeling issues related to parallel and distributed computer systems [2]. Those systems are often viewed as collections of components that operate more or less independently. They require only infrequent interaction such as synchronizing their actions or operating at different rates depending on the state of parts of the overall system. The components are modeled as individual stochastic automata that interacts with each other. A module (automaton) is modeled by a set of *states*, and the event or action of the module is modeled by a *transition* from a state to another. A single automaton represents only the state of one module; additional information is used to express *interactions* among the modules. On each transition, a label gives information about the timing and the probability of events occurrence. The transitions and the events are described by some matrices, for each automaton. We focus on synchronizing dependence, i. e., *local and events matrices are constant*. Under appropriate Markovian assumptions, the behaviour of the set of automata may be modeled by a CTMC, which state space is the cartesian product of the all the space states of the automata. It has been shown that the infinitesimal generator of the resulting CTMC, also known as the *descriptor*, can be obtained automatically into a compact formula, by means of Kronecker (tensor) algebra [3]. The consequence is that the state transition matrix is not stored, not even generated.

We are interested in the computation of the transient solution of SANs while they are very often analyzed in the stationary or quasi-stationary cases [4, 5]. The challenge is to choose a method that, at the same time, bounds the global error and deals with the time complexity. Moreover, the algorithms of that method must be parallelizable. Some studies have been done in the transient case, when the state space is reasonable. Among them, IRK3 (Implicit Runge-Kutta method of order 3) has been used [6, 7]. This method deals efficiently with stiff models, for large mission time t. But unfortunately, its time complexity is unpredictable and the global error is difficult to bound. A consequence of this latter drawback is that an accuracy cannot be chosen *a priori*. The Uniformized Power technique (UP) has also been proposed [8]. This method is very fast for systems with large values of t, but only usable in the case of moderate state spaces. The Standard Uniformization method (SU) has proved its efficiency for reasonable values of t, even when the state space size is important [9, 10]. This efficiency is altered when t increases. The main advantage of this technique is the possibility of bounding the global error and predicting the time complexity. The most part of the algorithms are sequential.

In this study, we first adapt the SU method to compute the transient solution of SANs. Next, we make a parallel implementation of the SU method in order to deal with the case of large values of t. We also derive a new parallel algorithm which computes the multiplication of a vector by a Kronecker tensor sum of matrices. This implementation uses an efficient parallel multiplication of a vector by a tensor product of matrices [11].

*Dpt MASS, Université de Rennes 2 Place du Recteur Henri Le Moal, CS 24307 35043 Rennes cedex, France, haiscam.abdallah@uhb.fr

The resulting global algorithm has a speedup which remains in average greater than 80%. It allows us to deal efficiently with problems that has not been solved yet. The structure of the paper is as follows. Section 2 sets the problem and includes a detailed sequential analysis. Section 3 is dedicated to the parallel algorithms. The numerical results on an Asynchronous Transfer Mode (ATM) network are presented in Section 4. The paper is concluded with Section 5.

2. Problem formulation. We consider a SAN that consists of N individual automata, $\mathcal{A}^{(k)},\ k = 1,\ldots,\ N$. The number of states in the k^{th} automaton is denoted by n_k for $k = 1,\ 2,\ \ldots,\ N$. Let $X^{(k)}$, for $k = 1,\ldots,\ N$, be the unidimensional CTMC associated with the automaton $\mathcal{A}^{(k)}$, $Q^{(k)}$ the infinitesimal generator of $X^{(k)}$ and $\Pi^{(k)}(0)$ its initial distribution vector. Let us denote by $\Pi^{(k)}(t)$ the state probability vector of $X^{(k)}$ at time t. The overall model (the SAN) is described by a multidimensional CTMC $X = \{X_t,\ t \geq 0\}$, which state space and size are respectively $E = \prod_{k=1}^{N} E^{(k)}$ and $M = \prod_{k=1}^{N} n_k$. Its infinitesimal generator (descriptor) and its initial distribution are denoted by Q and $\Pi(0)$. At time t, the transient distribution of the CTMC X is given by the vector $\Pi(t)$. Our goal is the computation of $\Pi(t)$ for a given value of the system's mission time t. The vector $\Pi(t)$ is solution of the first homogenous linear differential equations, known as Chapman-Kolmogorov equations:

$$\frac{\partial}{\partial t}\Pi(t) = \Pi(t)\mathrm{Q};\ \ \Pi(0) \text{ given.} \tag{2.1}$$

When the automata are independent, the descriptor Q is the tensor sum of the N infinitesimal generators $Q^{(k)},\ k = 1,\ldots,\ N$, resulting from the local transitions:

$$Q = \bigoplus_{k=1}^{N} Q^{(k)}.$$

It follows that:

$$\Pi(t) = \bigotimes_{k=1}^{N} \Pi^{(k)}(t). \tag{2.2}$$

Relation (2.2) leads us to the computation of vector $\Pi^{(k)}(t),\ k = 1,\ldots,\ N$, for CTMCs with moderate state space size n_k. An efficient method, like SU or UP, can be selected according to the problem's size and mission time t.

2.1. Dependent automata. Let T be the total number of the system's synchronizing events and for all $i = 1,\ldots,\ T$, $E_{i+}^{(k)}$ the matrix of event i over automaton $\mathcal{A}^{(k)}$, $k = 1,\ldots,\ N$; $E_{i-}^{(k)}$ is the regularisation matrix. The descriptor Q is then expressed as an ordinary sum of a local part and a synchronizing part [3, 4]:

$$Q = \bigoplus_{k=1}^{N} Q^{(k)} + \sum_{i=1}^{2T} \bigotimes_{k=1}^{N} E_i^{(k)} \ \text{ where } \ E_i^{(k)} \in \left\{E_{i+}^{(k)}, E_{i-}^{(k)}\right\}. \tag{2.3}$$

It is important to note that $\bigoplus_{k=1}^{N} Q^{(k)}$ can be transformed into ordinary sum of tensor products of matrices as follows

$$\bigoplus_{k=1}^{N} Q^{(k)} = \sum_{k=1}^{N} I_{n_1} \otimes \cdots \otimes I_{n_{k-1}} \otimes Q^{(k)} \otimes I_{n_{k+1}} \otimes \cdots \otimes I_{n_N}, \tag{2.4}$$

where $I_{n_k},\ k = 1,\ \ldots,\ N$, is the n_k order identity matrix. Consequently, the descriptor Q can be written under the form:

$$Q = \sum_{i=1}^{N+2T} \bigotimes_{k=1}^{N} Q_i^{(k)}, \ \text{ with } \ Q_i^{(k)} \in \{I_{n_i}, Q^{(k)}, E_i^{(k)}\}. \tag{2.5}$$

Most of the time, expression (2.5) is used to solve the overall model (the CTMC X), in order to transform the problem into the computation of a product vector-matrix, where the matrix is a tensor product of several

matrices. *We adopt expression (2.3) to compute the transient distribution.* Indeed, we develop, at the end of this section, a specific algorithm that computes the product of a vector by a tensor sum of matrices. This algorithm has the same time complexity as that which computes the product of a vector by a tensor product of matrices. Consequently, compared to (2.5), the form (2.3) allows an important saving in time complexity.

Let us remind that the major problem is the choice of methods that solve (2.1) taking into account the global error. An efficient error control mechanism is used by UP and SU methods. The UP technique performs well for CTMCs with large values of t and moderate state spaces while the SU method works better for CTMCs with moderate values of t and large state spaces. *The basic idea is, first of all, to implement the SU method for the SAN and next, analyze the contribution of parallelism when t increases.*

For $Q = (q_{ij})_{i,j=1,\,...,\,M}$ given, the expression of $\Pi(t)$ obtained by the SU method is [6, 12]:

$$\Pi(t) = \sum_{n=0}^{\infty} p(n, qt)\Pi(0)\tilde{P}^n, \tag{2.6}$$

where $q \geq \ max_{1\leq i\leq M} \mid q_{ii} \mid$, $p(n, qt) = e^{-qt}\frac{(qt)^n}{n!}$, and $\tilde{P} = I + Q/q$; I is the M order identity matrix.

The previous infinite sum can be truncated at a step N_T such that

$$1 - \sum_{n=0}^{N_T} p(n, qt) \leq \varepsilon, \tag{2.7}$$

where ε is a tolerance, given *a priori* by the user. That tolerance also bounds the global error on $\Pi(t)$ by SU. Let us note that for a given value of ε, N_T is always greater than qt.

Let $\tilde{\Pi}^{(n)}$, $n = 1, \ldots,\ N_T$, be the vector $\Pi(0)\tilde{P}^{(n)}$. These N_T vectors are computed by the following recurrence relation:

$$\tilde{\Pi}^{(n)} = \tilde{\Pi}^{(n-1)}\tilde{P}\ ,\ n \geq 1\ ;\ \tilde{\Pi}^{(0)} = \Pi(0). \tag{2.8}$$

From a time complexity point of view, the computation of $\Pi(t)$ requires one vector-matrix product by iteration (relation (2.8)). Using a compact storage for Q, the time complexity may be reduced to $O(N_T\eta)$ where η is the number of non zero elements in Q.

The SAN methodology has a first advantage of avoiding to build the whole generator Q. Using relations (2.3) and (2.8), we have:

$$\begin{aligned}\tilde{\Pi}^{(n)} = \tilde{\Pi}^{(n-1)} &+ \frac{1}{q}\left[\tilde{\Pi}^{(n-1)} \oplus_{k=1}^{N} Q^{(k)}\right] \\ &+ \frac{1}{q}\sum_{i=1}^{2T}\left[\tilde{\Pi}^{(n-1)} \otimes_{k=1}^{N} E_i^{(k)}\right],\ \ n \geq 1,\end{aligned} \tag{2.9}$$

with $\tilde{\Pi}^{(0)} = \Pi(0) = \otimes_{k=1}^{N}\Pi^{(k)}(0)$ given.

Another advantage of the SAN is the possibility of developing specific sequential and parallel algorithms to compute a vector-tensor product or sum of matrices. Those algorithms are often faster than the classical ones for which Q is entirely given (cf. 2.2.2 and 3).

2.2. Sequential approach.

2.2.1. Product of a vector by a tensor product of matrices. The computation of the vector $y = x\bigotimes_{k=1}^{N} A^{(k)}$ given the vector x and the N (small) matrices $A^{(k)}$, can be done by expressing each element of $\bigotimes_{k=1}^{N} A^{(k)}$ as a product of N elements of $A^{(k)}$, $k = 1, ..., N$. The time complexity of the algorithm is $O(N\prod_{k=1}^{N}\eta(A^{(k)}))$, where $\eta(A^{(k)})$ is the number of non-zero elements in the matrix $A^{(k)}$.
Because this time complexity is generally very large and the matrix $A^{(k)}$ (here $Q^{(k)}$) have a small value of $\eta(A^{(k)})$, an algorithm based on the perfect shuffle is used [1, 13]. Such an algorithm, called $TENS$, has the following time complexity

$$O\left(M\sum_{k=1}^{N}\frac{\eta\left(A^{(k)}\right)}{n_k}\right) \tag{2.10}$$

Let $\alpha_k = \frac{\eta(A^{(k)})}{n_k}$ be the mean number of non-zero terms in rows or columns of $A^{(k)}$ and let us set it to α. It is clear that an algorithm based on the perfect shuffle is better than a classical one if $\alpha > N^{\frac{1}{N-1}}$. The SANs satisfy very often the case.

2.2.2. Product of a vector by a tensor sum of matrices. For computing the vector $z = x \bigoplus_{k=1}^{N} A^{(k)}$, we transform $\bigoplus_{k=1}^{N} A^{(k)}$ into an ordinary sum of tensor product of matrices using relation (2.4). More precisely, if we define

$$M_l^u = \prod_{k=u}^{l} n_k \tag{2.11}$$

and $\bar{n}_k = M/n_k$, we have

$$x \bigoplus_{k=1}^{N} A^{(k)} = \sum_{k=1}^{N} x \left(I_{M_1^{k-1}} \otimes A^{(k)} \otimes I_{M_{k+1}^N} \right). \tag{2.12}$$

The computation of the vector z is based on the product

$$x \left(I_{M_1^{k-1}} \otimes A^{(k)} \otimes I_{M_{k+1}^N} \right). \tag{2.13}$$

At each iteration k, all the matrices, except $A^{(k)}$, are set to identity matrix. Algorithm 1, called $TENS_k$ (k^{th} iteration of $TENS$), is a particular case of a perfect shuffle to compute (2.13). In this algorithm, we consider $n_{left} = M_1^{k-1}$ and $n_{right} = M_{k+1}^N$. The time complexity of the algorithm $TENS_k$ is $O\left(\bar{n_k} \,.\, \eta\left(A^{(k)}\right)\right)$. Lines 5-8 and 11-16 describe the permutations required by this algorithm. If $TENS^+$ is the algorithm which computes z by calling N times $TENS_k$, the time complexity of such an algorithm is

Input : $n_k,\ A^{(k)},\ x,\ n_{left},\ n_{right}$
Output : $Y = x \left(I_{M_1^{k-1}} \otimes A^{(k)} \otimes I_{M_{k+1}^K} \right)$

1: $base \leftarrow 0\ ;\ jump \leftarrow n_k.n_{right}$
2: **for** $block = 0, \ldots,\ n_{left} - 1$ **do**
3: **for** $offset = 0, \ldots,\ n_{right} - 1$ **do**
4: $index \leftarrow base + offset$
5: **for** $h = 0, \ldots,\ n_k - 1$ **do**
6: $Z_h \leftarrow x_{index}$
7: $index \leftarrow index + n_{right}$
8: **end for**
9: $Z' = Z.A^{(k)}$
10: $index \leftarrow base + jump$
11: **for** $h = 0, \ldots,\ n_k - 1$ **do**
12: $Y_{index} \leftarrow Y_{index} + Z'_h$
13: $index \leftarrow index + jump$
14: **end for**
15: **end for**
16: $base \leftarrow base + jump$
17: **end for**

Algorithm 1: Algorithm $TENS_k$ for computing $Y = x \left(I_{M_1^{k-1}} \otimes A^{(k)} \otimes I_{M_{k+1}^N} \right)$

$$O\left(\sum_{k=1}^{N} \bar{n}_k \,.\, \eta\left(A^{(k)}\right) \right) = O\left(M \sum_{k=1}^{N} \frac{\eta\left(A^{(k)}\right)}{n_k} \right) \tag{2.14}$$

It is important to note that this time complexity is identical to that of the computation of $x \bigotimes_{k=1}^{N} A^{(k)}$ given by relation (2.10). In the following section, we give a parallel version of this algorithm.

3. Parallel implementation. Our goal is the parallelization of the computation algorithm of the vector $\Pi(t)$ based on relation (2.9). Taking into account the quantity of data (matrices and vectors) to be processed, it is quite necessary to choose a program scheme in which each processor owns some data, to which it applies some instruction streams. These instruction streams can be chosen identical for all the processors (SPMD: Single Program, Multiple Data). This scheme is easier to implement than the case where several algorithms are built, one for each processor (MIMD: Multiple Instruction streams, Multiple Data). We place ourselves in the SPMD mode. A crucial problem in this kind of implementation is the load balancing. Each node must have the same amount of work in order to assure a certain equilibrium and efficiency. It is therefore important to use a good decomposition and task repartition technique. This repartition must minimize a function, relative to the execution time of the program over P processors.

In order to make a parallel implementation in SPMD mode over P processors, we need a data decomposition into P subsets, determining each processor's task. This task is mainly composed with computation phases ended by synchronization phases.

The first part of our work consists in implementing relation (2.9) over P processors. At each step, this relation is essentially made with matrix-vector products, where the matrix is a tensor product or a tensor sum of matrices. First, we are going to deal with the tensor product case. Next, we shall proceed with the tensor sum. We shall end by the global algorithm.

3.1. Product of a vector by a tensor product of matrices. In order to compute $y = x \bigotimes_{k=1}^{N} A^{(k)}$ over P processors, we are going to use the algorithm proposed in [11]. The data are distributed according to the following scheme:

— Decomposition of P in N integers $d_1,\ d_2,\ \ldots, d_N$ such that d_k divides n_k. Let us notice that this decomposition is not unique, but all the possible decompositions are equivalent from a time complexity point of view. Nevertheless, a good criterium of choosing a decomposition instead of another is that the sum of the terms must be minimum.

— For each $k \in \{1,\ \ldots,\ N\}$, build a partition of $G_k = \{1,\ \ldots,\ n_k\}$ in d_k subsets G_{kl}, $l = 1,\ \ldots,\ d_k$, i. e.,

$$\bigcup_{l=1}^{d_k} G_{kl} = G_k.$$

That partition must be made respecting the load balancing between the P processors.
Let us consider the following indexation scheme.

$$(l_1,\ l_2, \ldots,\ l_{N-1},\ l_N) = l_{[1,N]} \rightleftharpoons (\ldots((l_1)n_2 + l_2)n_3 \ldots) = \sum_{k=1}^{N} l_k M_{k+1}^{N}, \tag{3.1}$$

where M_l^u is given by relation (2.11). For any processor p, considering relation (3.1), we have the following correspondence:

$$p \rightleftharpoons w_{[1,N]}$$

and thus the vector allocation is done the following way:

$$w_{[1,N]} \longleftarrow y_{(l_1,\ l_2,\ldots,\ l_N)},\ \textit{with}\ l_k \in G_{kw_k},\ k = 1,\ \ldots,\ N.$$

The proposed algorithm is recursive with N steps. It is based on the canonical factorization of the tensor product, such that *at each step, only one matrix of the product is used*. Each processor uses its own data for its computations, then sends the results to the processors that will need them in the following step. In the same time, it receives the data it will need for the next step. The communications between processors are expressed using simple primitives:

send: a processor send a message to a single processor.

receive: a processor receives a message from a single processor.

broadcast: a processor send a message to several processors.

These primitives are efficiently implemented over almost all the existing architectures [14]. The described algorithm uses an overlapping of communications and computations, avoiding bufferization. Another advantage

of that algorithm is a message is sent to a processor if and only if it needs it. The number of communications is therefore reduced to the minimum.

The execution time of a parallel program depends essentially on the communications. The transmission time of a message of $\mathcal{M}$ bytes between two processors p_1 and p_2 over a distance $d = dist(p_1, p_2)$ is represented by the linear model [15]:

$$t(d, \mathcal{M}) = \mathcal{M}.t_c(d, \mathcal{M}) + \tau(d, \mathcal{M}),$$

where $t_c(d, \mathcal{M})$ is the transmission time of one byte and $\tau(d, \mathcal{M})$ is the start-up time. It depends on d, but both of the parameter may be function of $\mathcal{M}$ if the computer uses different protocols of communication according message size (e. g. Intel iPSC/860). Finally, the execution time is the product of one communication time (the above linear function) by the number of communications.

The algorithm of a product vector-tensor product of matrices, that we shall refer to as *PARATENS*, executes at most $\Gamma(P)$ communication steps, $\Gamma(P)$ being the number of communication steps necessary to broadcast a message to P processors. This number depends on the topology, for example $\Gamma(P) = log(P)$, for hypercube topology. Let us note that the arguments of *PARATENS* are the vector and the matrices of the tensor product.

3.2. Product of a vector by a tensor sum of matrices. Let us remind that (relation (2.4)):

$$\bigoplus_{k=1}^{N} A^{(k)} = \sum_{k=1}^{N} \left(I_{M_1^{k-1}} \otimes A^{(k)} \otimes I_{M_{k+1}^N} \right). \tag{3.2}$$

The obvious way of computing $z = x \bigoplus_{k=1}^{N} A^{(k)}$ is as follows: at each iteration k, all the matrices arguments, except one, are set to identity. Then, PARATENS is used to complete the step. This results in executing PARATENS N times. But, we notice that at any iteration k, the expression

$$V \left(I_{M_1^{k-1}} \otimes A^{(k)} \otimes I_{M_{k+1}^N} \right)$$

is computed, where V is the vector resulting from the iteration $k-1$. Therefore, the result may be obtained by executing an algorithm such that the execution time is equal to that of PARATENS. That algorithm, called $PARATENS^+$, is presented below (Algorithm 2). In this algorithm, $PARATENS_k$ denotes the procedure that carries out the k^{th} iteration of PARATENS.

Input: $x,\ n_k,\ A^{(k)},\ k = 1, \ldots,\ N$
Output: $z = x \bigoplus_{k=1}^{N} A^{(k)}$
  $Y = 0$
  for $k = 1, \ldots, N$ **do**
    $Y \leftarrow Y + PARATENS_k(n_k, A^{(k)},\ Y)$
  end for

Algorithm 2: Parallel algorithm of the product vector-tensor sum of matrices

3.3. Implementation of the global algorithm. Algorithm 3 computes the vector $\Pi(t)$ over P processors. The first step of this algorithm consists in computing the uniformized rate q by following way. By definition,

$$q \geq \max_i \mid q_{ii} \mid,\ i = 1, \ldots, M_1^N,$$

where q_{ii} are the diagonal elements of the descriptor Q (relation (2.3)). Because the square matrix Q is an infinitesimal generator, then we have

$$q_{ii} = \sum_{j=1}^{M} q_{ij},\ j \neq i,$$

Input: $Q^{(k)}$, $E_i^{(k)}$, $\Pi(0)$, t, ε
Output: $\Pi(t)$
  Compute q and N_T * Relation (2.7)*\
  Compute Q/q
  $e_0 \leftarrow 1$; $\tilde{\Pi}^p(0) \leftarrow \Pi_0^p$; $\Pi^p(t) \leftarrow \Pi_0^p$
  for $j = 1, \ldots, N_T$ **do**
    $e_j \leftarrow \frac{qt}{j} e_{j-1}$
    $PARATENS^+(Q^{(1)}, \ldots, Q^{(N)}, \tilde{\Pi}^{(j-1)})$ * $\tilde{\Pi}^{(j-1)} \oplus_{k=1}^N Q^{(k)}$ *\
    for k=1,...,2T **do**
      $PARATENS(E_k^{(1)}, \ldots, E_k^{(N)}, \tilde{\Pi}^{(j-1)})$ * $\tilde{\Pi}^{(j-1)} \otimes_{k=1}^N E_i^{(k)}$*\
    end for * Results in $\tilde{\Pi}^{(j)^p}$ for each processor p *\
    $\Pi^p(t) \leftarrow \Pi^p(t) + e_j \tilde{\Pi}^{(j)^p}$
  end for

Algorithm 3: Parallel algorithm of the computation of $\Pi(t)$ over P processors

where $M = M_1^N$ is the size of the matrix Q. Finally,

$$q \geq \max_i \mid \sum_{j=1}^{M} q_{ij} \mid,\ i = 1, \ldots, M.$$

The rate q is obtained by computation of the vector

$$Y = Q \times \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix},$$

where all the diagonal elements of Q are substituted by zero. Next, we choose q such that

$$q = \max_i \mid Y_i \mid,\ i = 1, \ldots,\ M.$$

The computation of $\Pi(t)$ requires, at each step, one execution of $PARATENS^+$ and $2T$ executions of $PARATENS$. *The time complexity of the global algorithm is minimum due to the fact that all the algorithms are optimum in communications number.*

4. Numerical results.

4.1. The ATM example. In this example, we describe a congestion control of an ATM (Asynchronous Transfer Mode) network. The ATM [16] was conceived to face the transmission of new types of data (voice, video, etc.). It is a specific packet oriented transfer mode based on fixed length cells of 53 bytes. These cells result from the splitting of the input streams. A source connected to the network inserts its cells into free spaces not used by the other sources. High speed connections of this kind may exceed $600 Mbits/s$.

A crucial problem in such networks is the congestion control. Moreover, this problem must be treated with integration of quick adaptation and reaction to high speed connections. This problem is responsible for loss of information (buffers saturation) and transmission time increase. Solving this problem consists in reducing the congestion with a preventive and adaptive method. The classical techniques are not always applicable, on account of the high speed in the ATM networks. It is therefore necessary to establish adapted control mechanisms. The congestion control in ATM network may be executed at different level according to the kind of information carried and the traffic's characteristics. Three levels are possible:

— Admission level
— Burst level
— Cell level.

At admission level, the system determines whether a connection can be progressed or should be rejected based on the resource availability in the network: it is an access control. At burst level, a control mechanism (such as

the leaky bucket) checks, permanently, that the input flow respects the negotiated traffic contract: it is a flow control. At cell level, the control is done using the CLP bit (Cell Loss Priority) contained in the header of each cell. This bit makes the difference between cells according to their priority. In congestion case, low priority cells are destroyed. Only high priority cells are kept in the network.

The Leaky Bucket (LB) [17] is an access control mechanism in the ATM network. This control is performed using tokens. These tokens are given to each cell when it enters the network. This mechanism may be implemented in several ways. We focus on the one called the Virtual Leaky Bucket (VLB) [18]. In this kind of LB, three buffers are needed. The first one welcomes the user's cells, and the others are respectively used for green and red tokens. When a cell arrives in the first buffer B_c, if that one is not full, it is kept there waiting to be served. A service to a cell consists in giving it a green token coming from the buffer B_g. This token represents its permission to access into the network. Otherwise, if B_c is full, the cell may be lost (rejected), even if the network has sufficient resources to accept it, without altering the quality of service. In order to avoid this situation, red tokens are generated by the buffer B_r. A threshold S is fixed. If B_g is empty while there are less than S cells in B_c, those cells should wait for new green tokens to be generated, before they enter the network. On the contrary, if there are more than S cells in B_c when B_g is empty, they should be able to access the network with red tokens if, of course, B_r is not empty. This mechanism is described by figure 4.1.

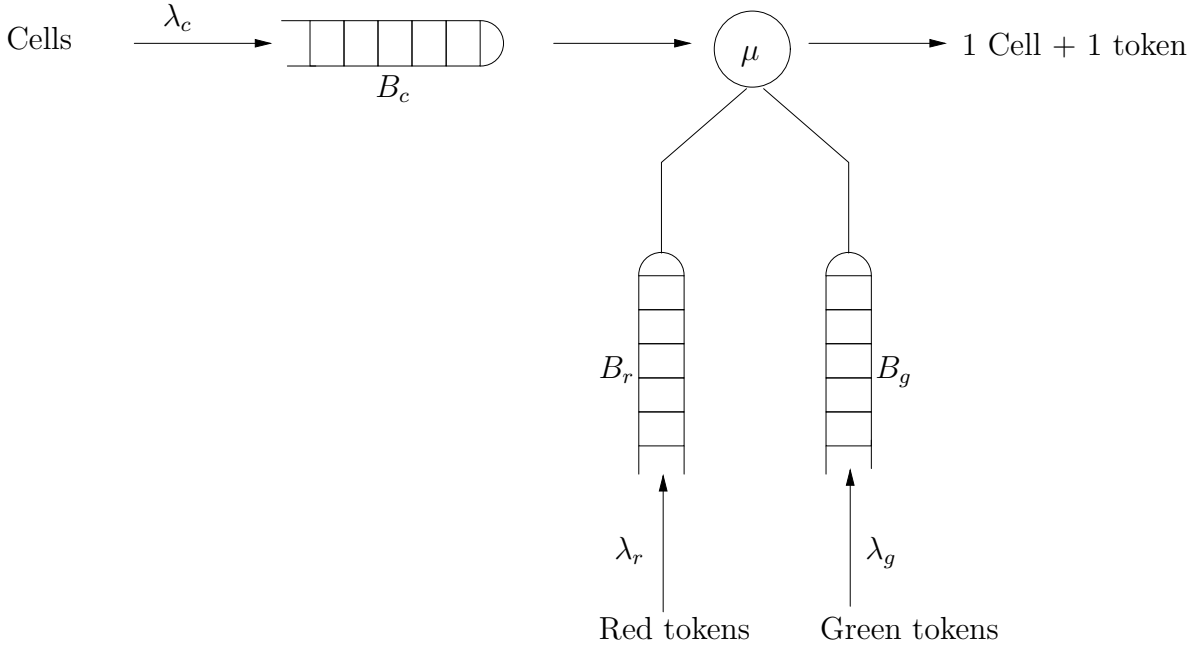

FIG. 4.1. *The Virtual Leaky Bucket*

The ATM mechanism is modeled by a SAN composed with three automata $\mathcal{A}^{(1)}$, $\mathcal{A}^{(2)}$ and $\mathcal{A}^{(3)}$. These automata represent respectively the content of buffers B_c, B_g and B_r. Each automaton $\mathcal{A}^{(k)}$ is supposed to haves n_k states, $k = 1, 2, 3$. These states are numbered from 0 to $n_k - 1$. The threshold of buffer B_c is set to S. The events that occur in the system are as follows:

- Local events
  - — arrival of a cell with rate λ_c, a local event to $\mathcal{A}^{(1)}$
  - — arrival of a green token with rate λ_g, a local event to $\mathcal{A}^{(2)}$
  - — arrival of a red token with rate λ_r, a local event to $\mathcal{A}^{(3)}$.
- Synchronizing events
  - — s_1: departure of a cell with a green token (rate μ), acting on both $\mathcal{A}^{(1)}$ and $\mathcal{A}^{(2)}$
  - — s_2: departure of a cell with a red token (rate μ), acting on both $\mathcal{A}^{(1)}$ and $\mathcal{A}^{(3)}$.

Figure 4.2 shows the behaviour of each automaton for the case where $n_1 = 4$, $n_2 = n_3 = 3$ and $S = 1$.

The transitions in $\mathcal{A}^{(1)}$ are either cells arrivals (with rate λ_c) or cells departures (service). A departure is also synchronization transition because a cell leaves B_c with a token (green if the number of cells in B_c is less

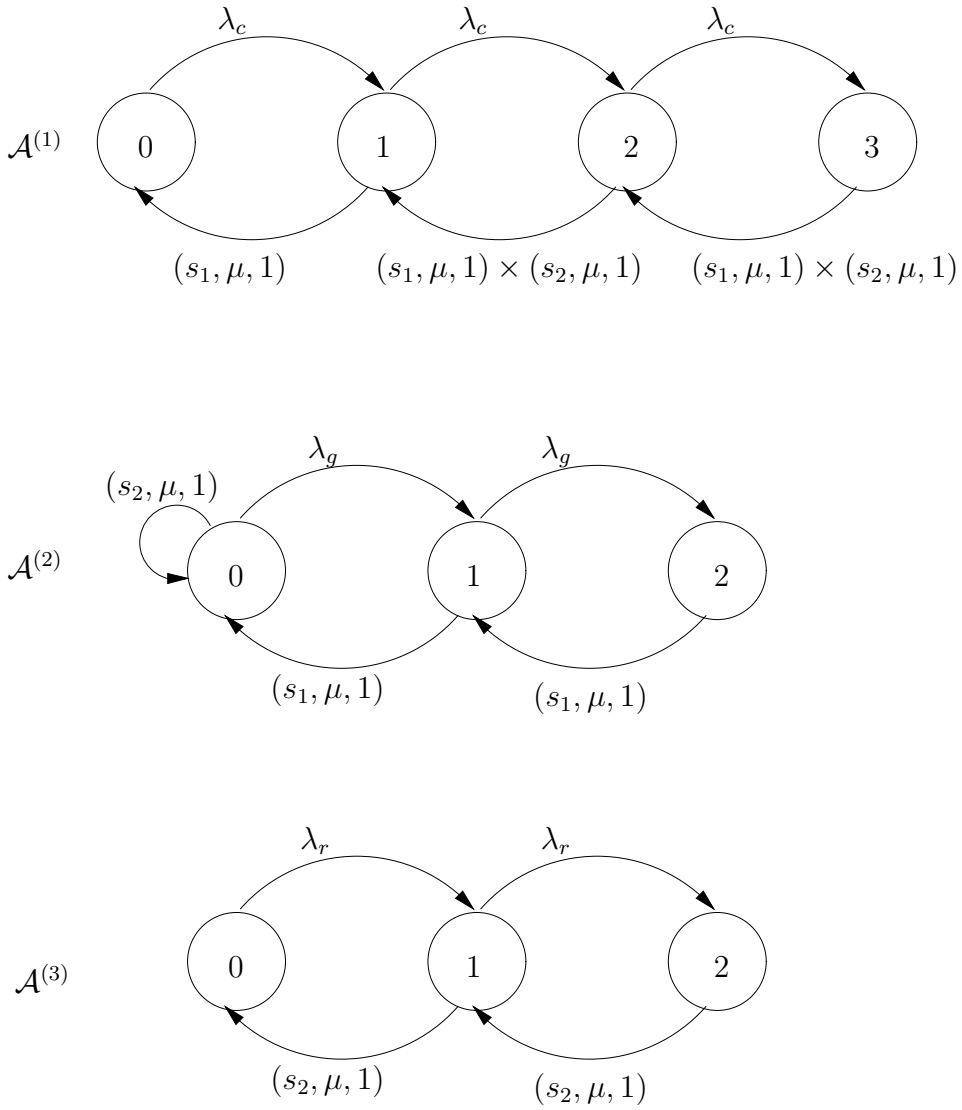

FIG. 4.2. *The automata transitions diagram of the SAN associated to the Virtual Leaky Bucket*

than S and red if not). The service rate is always μ and the occurrence probability (routing probability) is 1, only one choice being possible. The transitions in $\mathcal{A}^{(2)}$ and $\mathcal{A}^{(3)}$ are arrivals (with rates λ_g or λ_r) of green or red tokens or their departures. A token departure means that a cell has been served, therefore synchronization events s_1 and s_2 stand. These transitions have rate μ and also an occurrence probability equals to 1. The fact that a red token is usable only when B_g is empty is modeled by the loop around state 0 of automaton $\mathcal{A}^{(2)}$.

The SAN modeling the VLB is determined by the descriptor Q and the initial distribution $\Pi(0)$ as follows. Let be, for $k = 1, 2, 3$,

$Q^{(k)}$ the infinitesimal generator associated to automaton $\mathcal{A}^{(k)}$,
$E_1^{(k)}$ the positive event matrix of event s_1 over automaton $\mathcal{A}^{(k)}$, and
$E_2^{(k)}$ the positive event matrix of event s_2 over automaton $\mathcal{A}^{(k)}$.

The *descriptor* is given by

$$Q = Q^{(1)} \oplus Q^{(2)} \oplus Q^{(3)} + E_1^{(1)} \otimes E_1^{(2)} \otimes E_1^{(3)} + E_2^{(1)} \otimes E_2^{(2)} \otimes E_2^{(3)}.$$

If $\Pi^{(k)}(0)$, $k = 1,\ 2,\ 3$, denotes the initial distribution of the k^{th} automaton $\mathcal{A}^{(k)}$, we have $\Pi_1^{(k)}(0) = 1$ and for $i \geq 2$, $\Pi_i^{(k)}(0) = 0$. The global initial distribution (of the SAN) is $\Pi(0)$ such that $\Pi(0) = \otimes_{k=1}^{3} \Pi^{(k)}(0)$.

4.2. Performance analysis. In order to compute the vector $\Pi(t)$ for our SAN model, we consider $n_1 = 256$, $n_2 = 128$ and $n_3 = 32$. This situation means that buffers B_c, B_g and B_r have a limited capacity of 255, 127 et 31 respectively. The threshold S is fixed to 200. Thus, the descriptor Q has an order $M = 2^{20}$ and **the system has 1.048.576 states**. It is important to note that only the matrices $Q^{(k)}$ and $E_i^{(k)}$, $k = 1, 2, 3$ and $i = 1, 2$, are stored by using a compact storage scheme. The rates values are such that $\lambda_b = \lambda_c = \lambda = 0.5$ and $\mu = 1$. For evaluating the performance of our global algorithm, we execute it on a CRAY T3E, a distributed memory parallel machine. It possesses up to 256 processing elements, each running at 300 MHZ. The computations of the program (written in Fortran) are done in numerical arithmetic double precision.

We first focus on the CPU time of our algorithm as function of the mission time t and the number of processors P. We consider $t = 10^i$, $i = 0, \ldots, 5$ and $P = 32, 64, 128$. Next, we compute the speedup $\mathcal{S}_P$ and the efficiency $\mathcal{E}_P$ as function of P. For the SU method, the value of ε is fixed to 10^{-10} (cf. relation (2.7)).

TABLE 4.1
CPU time (s) for computing $\Pi(t)$ as function of t and P

t	1	10	10^2	10^3	10^4	10^5
P=32	61.72	222.17	1745.45	14599.34	137696.21 (e)	1350907.9 (e)
P=64	30.55	137.76	912	7628.15	71945 (e)	705842.6 (e)
P=128	1.6	7.96	48.16	427.93	3791.89	37201.16

Table 4.1 includes the CPU time values for computing $\Pi(t)$ as function of t and P. In this table, *the notation (e) means that corresponding CPU time values are estimated taking advantage of the SU method for estimating a priori the time complexity.* A lecture of this table shows that even when $t = 10^5$, the vector $\Pi(t)$ can be evaluated with 128 processors in about 10 CPU hours. This table also shows the feasibility limits of some problems as function of the state space size M, mission time t, and number of processors P. Speedup

TABLE 4.2
Speedup and efficiency as function of P

P	32	64	128
$\mathcal{S}_P$	29	54	100
$\mathcal{E}_P$	0.90	0.84	0.78

and efficiency, as function of P, are given in Table 4.2. The values of $\mathcal{S}_P$ increases with P; the CPU times is then inversely proportional to P. Table 4.2 shows that the value of $\mathcal{E}_P$ remains in average greater than 80%, meaning a good use of processors and a low communication time.

It is important to note that the given numerical results depend on the SAN model. The speedup $\mathcal{S}_P$ decreases when N increases [19]. It is then possible to aggregate some automata [20] in order to obtain the same value of N. The mean number α of non-zero terms in rows or columns of matrices used in the tensor sums and products increases without modifying significantly the efficiency $\mathcal{E}_P$. If we consider complex SANs for which N is large (N increases), it is difficult to predict the expected speedup starting from the given application. More complex applications constitute the goal of our future work.

5. Conclusion. In this study, we addressed the problem of transient solution of SAN. This modelisation methodology allows to treat complex parallel systems by avoiding the built of the entire infinitesimal generator. In the computation of the transient state probability vector, we faced the problem of computation time, especially, for large mission time values. We first adapted the SU method to compute this vector and developed an new sequential algorithm which computes a product of a vector by a tensor sum of matrices. Next, we implemented a parallel algorithm of the SU method in order to deal with the increasing of the time complexity with the mission time. The parallel version of this implementation uses an efficient algorithm computing a product of a vector by a tensor product of matrices and a parallel version of the presented algorithm. The interest of used algorithms is the minimization of communication time between processors. We presented numerical results of a SAN modeling an ATM network with 1048000 states and large mission time values. Some CPU time values are given as function of mission time and number of processors. We also given speedup and efficiency as function of number of processors. Even a decreasing of the efficiency when the number of processors increases, the value of this efficiency remains, in average, greater than 80%.

REFERENCES

[1] B. Plateau, *On the Stochastic Structure of Parallelism and Synchronisation Models for Distributed Algorithms,* Performance Evaluation, 13(5):142–154, 1985.

[2] W. J. Stewart, K. Atif, and B. Plateau, *The numerical solution of stochastic automata networks,* European Journal of Operation Research, 86:503–525, 1995.

[3] B. Plateau, *On the Stochastic Structure of Parallelism and Synchronisation Model for Distributed Systems,* ACM SIGMETRICS conference On Measurement and Modeling of Computer Systems, pages 147–153, May 1985.

[4] B. Plateau and K. Atif, *Stochastic Automata Networks for Modelling Parallel Systems,* IEEE Transactions On Software Engineering, 17(10):1093–1108, 1991.

[5] M. S. Bebbington, *Parallel implementation of an aggregation/disaggregation method for evaluating quasi-stationary behaviour in continuous-time Markov chains,* Parallel Computing, 23:1545–1559, 1997.

[6] J. K. Muppala M. Malhotra, K. S. Trivedi, *Stiffness-tolerant methods for transient analysis of stiff Markov chains,* Microelectronic Reliability, 34(11):1825–1841, 1994.

[7] K. S. Trivedi and A. L. Riebman, *Numerical Transient Analysis of Markov Models,* Computer and Operations Research, 15(1):19–36, 1988.

[8] H. Abdallah and R. Marie, *The Uniformized Power Method for Transient Solutions of Markov Processes,* Computers and Operations Research, 20(5):515–526, April 1993.

[9] C. Lindemann, M. Malhotra and K. S. Trivedi, *Numerical Methods for Reliability Evaluation of Markov Closed Fault-Tolerant Systems,* IEEE Transactions on Reliability, 44(4):694–704, 1995.

[10] H. Abdallah and M. Hamza, *Sensitivity analysis of instantaneous transient measures of highly reliable systems* 11^{th} European Simulation Symposium (ESS'99), Erlangen-Nuremberg, Germany, october 26-28, pages 652–656, 1999.

[11] C. Tadonki and B. Philippe, *Parallel Multiplication of a Vector by a Kronecker Product of Matrices (part I),* Parallel and Distributed Computing Practices, 2(4):413–427, December 1999.

[12] H. Abdallah and M. Hamza, *Sensitivity analysis of the expected accumulated reward using uniformization and IRK3 methods,* Second Conference on Numerical Analysis and Applications (NAA'2000), Rousse, Bulgaria, June 11-15, 2000.

[13] M. Davio, *Kronecker Products and Shuffle Algebra* IEEE Transactions on Computers, 30:116–125, 1981.

[14] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and Distributed Computing.* Prentice-Hall, Englewood Cliffs, N. J., 1988.

[15] S. M. Mller A. Bingert, A. Formella and W. J. Paul, *Isolating the Reasons for the Performance of Parallel Machines on Numerical Programs,* In International Workshop on Automatic Distributed Memory Parallelization, Automatic Data Distribution and Automatic Parallel Performace Prediction, pages 34–64, Austin, Texas, USA, 1993.

[16] B. Weiss *ATM.* Hermes, Paris, 1995.

[17] I. Cidon and I. S. Gopal, *An approach to Integrated High-Speed Private Network,* International Journal on Digital and Analogical Systems, 1:77–86, 1988.

[18] M. Hirano and N. Watanabe, *Characteristics of a cell multiplexer for bursty ATM traffic,* IEEE ICC'89, pages 1321–1325, 1989.

[19] C. Tadonki and B. Philippe, *Parallel Multiplication of a Vector by a Kronecker Product of Matrices (part II),* Parallel and Distributed Computing Practices, 3(3), September 2000.

[20] O. Gusak, T. Dayar, and J. M. Fourneau, *Lumpable continuous-time stochastic automata networks,* International Conference on Mathematical Modeling and Scientific Computing, Middle East Technical University and Selcuk University, Ankara and Konya, Turkey, April 2001.

Edited by: Roman Trobec
Received: January 15th, 2002
Accepted: June 26th, 2002