Performance Analysis Issues for Parallel Implementations of Propagation Algorithm*

Leonardo Brenner Luiz Gustavo Fernandes Paulo Fernandes Afonso Sales
Faculdade de Informática, PUCRS, Av. Ipiranga, 6681 - 90619-900, Porto Alegre, Brazil
{lbrenner, gustavo, paulof, asales}@inf.pucrs.br

Abstract

*This paper presents a theoretical study to evaluate the performance of a family of parallel implementations of the propagation algorithm. The propagation algorithm is used to an image interpolation application. The theoretical performance analysis is based on the construction of generic models using Stochastic Automata Networks (*SAN*) formalism to describe each implementation scheme. The prediction results can be compared to the achieved performance in some real test cases to verify the accuracy of our modeling technique. The main contribution of this paper is to point out the advantages and problems of our approach to the development of generic models of parallel implementations.*

1. Introduction

The need of performance prediction of parallel implementations is uncontested. Many authors have presented generic studies offering options to the performance prediction of parallel implementations [9, 10]. To do so, the research community classifies the different approaches in three quite distinct groups:

- The monitoring approaches, which are mostly based on time costs comparisons of implementations execution; Those implementations can be as superficial as synthetic programs [9], or quite generic as well-known benchmarks, *e.g.*, *whetstones*, *dhrystones*, and *LINPACK*;
- The simulation approaches, which are based on the use of a computational tool to describe and simulate the behaviour of a given implementation; The simulation tools are commonly based on random generations of program choices, but also more sophisticated techniques as perfect simulation [11] can be used;
- The analytical modeling approaches, which are more rarely employed in the parallel programs prediction due to a frequently misconception; The most known formalisms to analytical modeling, *e.g.*, Markov chains [18] and queueing networks [7], are not very suitable to represent parallelism and synchronization.

Our work is located in the effort to popularize the use of analytical modeling to performance prediction of parallel implementations. In this paper we use the Stochastic Automata Networks formalism (SAN), a formalism with parallel and synchronizing primitives, to model a parallel implementation. The application chosen is the parallel implementation of the propagation algorithm [12], a *bag of tasks* parallelization proposed in [4].

The objective of this paper is to verify the difficulties to model a parallel implementation using an analytical modeling formalism. Similar previous works are rather generic [10], or too specific [19]. We try to contribute with a more systematic approach, based on describing the communication and processing tasks of each part of the algorithm. It is not the objective of this paper to perform a fine-tunning performance prediction of the specific implementation of the propagation algorithm, but to draw the main needs and advantages of the analytical modeling for this class of problem. In fact, the proposed model can be easily generalized to describe any bag of tasks problem.

The next section presents the propagation algorithm. Section 3 presents the parallel implementation defined in [4]. Section 4 presents the SAN formalism in order to fully understand the proposed model in Section 5. Section 6 presents the numerical definition of the model parameters to be used in the performance prediction of a practical case. Finally, the conclusion draws the next steps to continue this work and summarizes the lessons learned so far.

* Authors are partially supported by HP Brasil-PUCRS agreement CAP (T.A. 22) and CASCO (T.A. 24) projects.

0-7695-2046-4/03 $17.00 © 2003 IEEE
IEEE COMPUTER SOCIETY

2. Image Interpolation

Image-based interpolation is a method to create smooth and realistic virtual views between two original view points. The application studied in this paper [12] is based on a three-phase method. It starts by constructing a dense matching map using a growing/propagation scheme from a list of seed pairs which may contain bad matches. The matching is then corrected using local and global geometric constraints. In the following phase, a joint view triangulation algorithm is used to separate matched areas from unmatched ones and handle the partially occluded areas. Finally all in-between images are generated by interpolating the two original ones.

	Flower	House
Dimensions	368x384 pxs	768x512 pxs
Seed pairs	0.46s	2.24s
Propagation	**5.11s**	**19.75s**
Triangulation	1.64s	3.30s

Table 1. Images size and execution time for each phase of the algorithm.

Table 1 shows the execution times for each one of the three main phases of the application for two different image pairs (small size Flower and medium size House). The executions have been carried out on a Pentium III 733 MHz. These two examples indicate that the propagation is the largest time consuming phase of the algorithm.

2.1. Seed Pairs

In order to allow a better understanding of the propagation algorithm, the initial matching (seed selection) is presented here.

Points of interest [8, 17] are naturally good seed point candidates because they represent the points of the image which have the highest textureness. These points are detected in each separated image. Next, they are matched using the ZNCC (zero-mean normalized cross correlation) measure. At the end of this phase, a set of seed pairs is ready to be used to bootstrap a region growing type algorithm, which propagates the matches in the neighborhood of seed points from the most textured pixels to the less textured ones.

2.2. Propagation Algorithm

The propagation algorithm is based on a classic region growing method for image segmentation [15] which uses pixel homogeneity. But, instead of using pixel homogeneity property, a similar measure based on the matches correlation score is adopted [13]. This propagation strategy could also be justified because the seed pairs are composed by points of interest, which are the local maxima of the textureness. Thus, these matches neighbors are also strongly textured, what allows good propagation even though they are not local maxima.

The neighborhood $N_5(a, A)$ of pixels a and A is defined as being all pixels within the 5x5 window centered at these two points (one window per image). For each neighboring pixel in the first image, a list of match candidates is constructed. This list consists of all pixels of a 3x3 window in the corresponding neighborhood in the second image. The complete definition of the neighborhood $\mathcal{N}(a, A)$ of pixel match (a, A) is given by:

$$N(a, A) = \{(b, B), b \in \mathcal{N}_5(a), B \in \mathcal{N}_5(A), \\ (B - A) - (b - a) \in \{-1, 0, 1\}^2\}.$$

The input of the algorithm is the set *Seed* which contains the current seed pairs. This set is implemented by a heap data structure for a faster selection of the best pair. The output is an injective displacement mapping *Map* which contains all the good matches found by the propagation algorithm. Let $s(\mathbf{x})$ be an estimation of the luminance roughness for the pixel $\mathbf{x}$, which is used to stop propagation into insufficiently textured areas. And let t be a constant value that represents the lower luminance threshold accepted on a textured area.

Algorithm 1 Propagation Algorithm

```
1:  while Seed ≠ ∅ do
2:    pull the best match (a, A) from Seed
3:    Local ← ∅
4:    {store in Local new candidate matches}
5:    for all (x, y) ∈ N(a, A) do
6:      if ((x, ∗), (∗, y) ∉ Map) and (s(x) > t, s(y) > t)
        and (ZNCC(c, d) > 0.5) then
7:        Local ← (x, y)
8:      end if
9:    end for
10:   {store in Seed and Map good candidate matches}
11:   while Local ≠ ∅ do
12:     pull the best match (x, y) from Local
13:     if (x, ∗), (∗, y) ∉ Map then
14:       Map ← (x, y), Seed ← (x, y)
15:     end if
16:   end while
17: end while
```

Briefly, all initial seed pairs are starting points of concurrent propagations. At each step, a match (a, A) with the best ZNCC score is removed from the current set of seed pairs.

0-7695-2046-4/03 $17.00 © 2003 IEEE
IEEE COMPUTER SOCIETY

Then, the algorithm looks for new matches in its match neighborhood and, when it finds one, it is added to the current seed pairs set and also to the set of accepted matches which is under construction.

An example of the sequential propagation program execution can be observed on Figure 1. The squared regions in both images show the extension of the matched regions obtained from the seed matches.

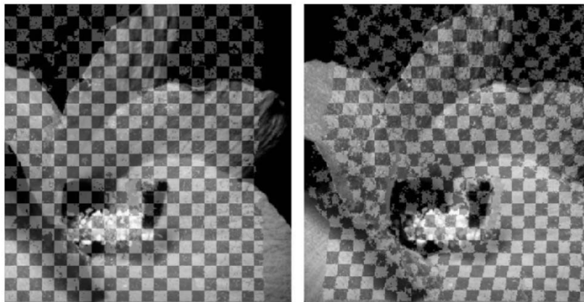

Figure 1. Propagation example using the Flower pair

3. Parallel Propagation Implementation

The parallel implementation for the propagation algorithm discussed on this section was developed in order to allow the use of this new algorithm on real situations. Thus, it was necessary to achieve better performances without using parallel programming models oriented to very expensive (but not frequently used) machines. Useful parallel versions for this algorithm should run distributed over several processors connected by a fast network. Therefore, the natural choice was a cluster with a message passing programming model.

As seen before, the propagation algorithm advances by comparing neighbors pixels through out the source images surface. From some seed pairs, it can form large matching regions on both images surface. In fact, a single seed pair can start a propagation that grows through a large region over the images surface. This freedom of evolution guarantees the algorithm to achieve good results in terms of matched surfaces. Another characteristic is that the algorithm is based on global best-first strategy to choose the next seed pair that will start a new propagation, which also have a direct effect on the final match quality. These two characteristics are hard to deal with if one wants to propose a parallel distributed version of the algorithm without loosing quality at the final match. The best-first strategy implementation is based on a global knowledge of the seed pairs set, which is not appropriated to a non-shared memory context. In addition, the freedom of evolution through out the images surface assumes that the algorithm knows the entire surface of the images, and this can create a situation where several processors are propagating over the same regions at the same time creating a redundancy of computation (Figure 2).

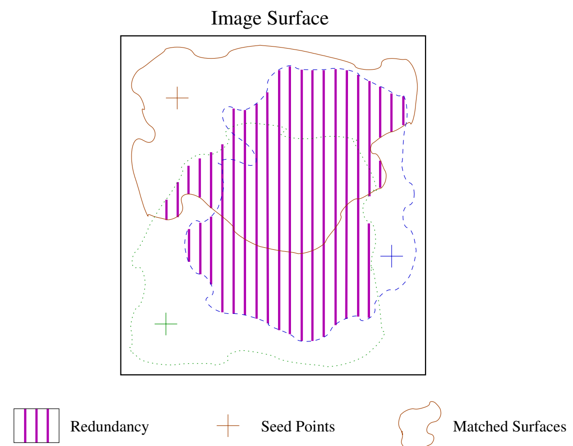

Figure 2. Redundancy problem.

Besides, it is not possible to know in advance how many new matches a seed pair will generate. Thus, from a parallel point of view, the propagation algorithm is an irregular and dynamic problem which exhibits unpredictable load fluctuations. Therefore, it requires the use of some load balancing scheme in order to achieve a more efficient parallel solution.

The solution proposed in [4] is based on a master-slave scheme. One processor will be responsible for distributing the work and centralizing the final results. The others will be running the propagation algorithm each one using a subset of the seed pairs and knowing a pair of corresponding slices over the images surface (coordinates of target slice). The master distributes the seed pairs over the nodes considering their location over the slices (see Figure 3). This procedure replaces the global best-first strategy by several local best-first ones. Each local seed pairs sub-set is still implemented as a heap which is ordered by the pair ZNCC score. This strategy minimizes the problem of loosing quality at the final match.

Once the problem with the global best-first strategy is solved, it still remains the problem with the algorithm limitation of evolution over the images surface. As said before,

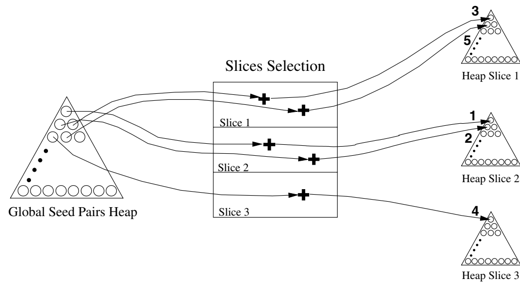

Figure 3. Seed pairs heap distribution over the slices.

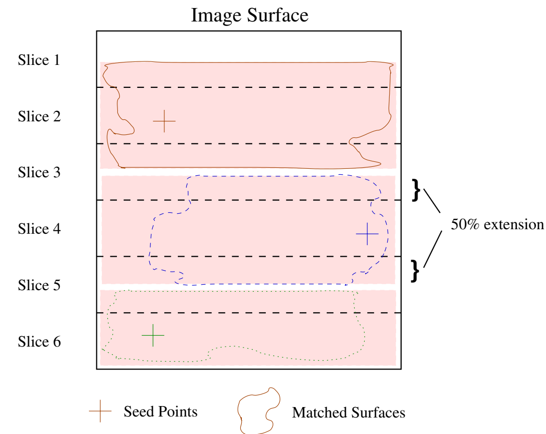

Figure 4. Flexible slices approach.

each node can propagate just over the surface of its associated slice in order to avoid computation redundancy. But, forbidding the evolution out of the associated slice generates two kinds of losses. First, some matches are not done because they are just at the border of one slice and one of its points is placed outside it. Second, some regions in one slice may not be reached by any propagation started by a seed pair located inside of its surface, but instead they could be reached by a propagation started at a neighbor slice.

Such a limitation is partially solved by a technique called flexible slices [4]. This technique allows the propagation algorithm to expand through the surface of its neighbor slices in a controlled way. As shown on Figure 4, each processor works over its own associated slice, but it also knows its neighbor slices and it has the permission to propagate over them. But still, it is not interesting to leave the propagation algorithm free to compute its neighbors entire surface. This may cause the computation of too many repeated matches. To avoid that, each processor has the permission to compute just over a percentage of its neighbors surface. This percentage, called *overlap*, is related to the number of slices. A large number of slices implies in thinner slices. In this case, it is acceptable to allow a processor to advance over a large percentage of its neighbors surfaces. On the other hand, a small number of slices implies in larger slices. Here, the algorithm must not propagate too much over the neighbors surface.

The last problem to deal with in the parallelization of the propagation algorithm is load balancing. If the source images are divided into more slices than the number of nodes available, the load balancing strategy adopted is:

1. the master divides the set of seed pairs into sub-sets based on their location over the slices;
2. each slave receives one slice with its associated sub-set;
3. each slave computes its own sub-set of seed pairs;
4. when there is no more seed pairs to compute, the slave sends a signal to the master;
5. if there is some available slices remaining, the master choose a new one and send it to the available slave;

In fact, the master has a queue of slices, organized by their position over the images surface. To choose which slice will be sent to an available slave, the master simply gets the first slice of this queue. In fact, the master sends a new seed pairs sub-set (coordinates of the slice) to the slave.

Finally, it is important to mention that the master must receive all matches generated by the slaves and it must filter the unavoidable duplicated ones. To send these final matches to the master, each slave has a communication buffer which is filled progressively as the propagation algorithm advances. When the buffer is full, it is sent to the master. After that, the slave immediately returns to its execution. All slaves do the same procedure, in a way that forces the master to have a receiving queue. This queue is dimensioned to avoid buffer losses by the master. When a slave reaches the end of its seed pairs sub-set, it sends an incomplete buffer to the master. When the master receives an incomplete buffer, it knows that the sender has finished its work and sends a new slice (seed pairs sub-set) back to it (if there is still sub-sets available).

4. SAN - Stochastic Automata Networks

The Stochastic Automata Networks formalism (SAN) was proposed by Plateau [16]. The reader interested in a formal description of the formalism can consult previous publications [5, 1].

The basic idea of SAN is to represent a whole system by a collection of subsystems with an independent behavior

(*local transitions*) and occasional interdependencies (*functional rates* and *synchronizing events*).

The SAN formalism describes a complete system as a collection of subsystems that interact with each other. Each subsystem is described as a stochastic automaton, *i.e.*, an automaton in which the transitions are labeled with probabilistic and timing information. Hence, one can build a continuous-time stochastic process[1] related to the SAN. *Global state* is the state of a SAN model defined by the combination of the *local states* of all automata. We adopt the following notation to this paper:

Let

- $\mathcal{A}^{(i)}$ the i-th automaton of a SAN model, numbering the first automaton as $\mathcal{A}^{(1)}$;
- $j^{(i)}$ the j-th state of the automaton $\mathcal{A}^{(i)}$, numbering the first state of the first automaton as $0^{(1)}$;
- e_k an event identifier[2];
- τ_k a constant rate of a given event;
- π_k a constant probability of a given event;
- f_k a functional rate of a given event;
- g_k a functional probability of a given event.

There are two types of events that change the global state of a SAN model: *local events* and *synchronizing events*.

Local events change the global state passing from a global state to another that differs only by one local state. On the other hand, synchronizing events can change simultaneously more than one local state, *i.e.*, two or more automata can change their local states simultaneously. In other words, the occurrence of a synchronizing event *forces* all concerned automata to fire a transition corresponding to this event. Thus, local events can be viewed as a particular case of synchronizing events that concerns only one automaton.

Each event is represented by an *identifier*[3] and a *routing probability* (the absence of probability is tolerated if only one transition can be fired by an event from a local state).

Figure 5 represents a SAN model with 2 automata, 1 synchronizing event, and 1 functional rate.

Table 2 describes the event firing rates used in the SAN model of the Figure 5.

1 In the context of this paper only continuous-time SAN will be considered, although discrete-time SAN can also be employed without loss of generality.

2 The index k has no particular semantic for the notations of this section.

3 In this paper, we use an indexed roman letter e as identifier, but to all purposes any identifier can be used.

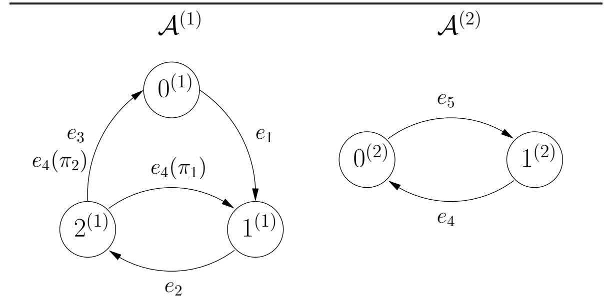

Figure 5. Example of a SAN model.

Event	Rate
e_1	τ_1
e_2	τ_2
e_3	τ_3
e_4	τ_4
e_5	f_1

Table 2. Event firing rates of the Figure 5

In the model of Figure 5, the rate of the event e_5 is not a constant rate, but a function rate called f_1 defined as:

$$f_1 = \begin{cases} \lambda_1 & \text{if automaton } \mathcal{A}^{(1)} \text{ is in the state } 0^{(1)} \\ 0 & \text{if automaton } \mathcal{A}^{(1)} \text{ is in the state } 1^{(1)} \\ \lambda_2 & \text{if automaton } \mathcal{A}^{(1)} \text{ is in the state } 2^{(1)} \end{cases}$$

The firing of the transition from state $0^{(2)}$ to $1^{(2)}$ occurs with rate λ_1 if automaton $\mathcal{A}^{(1)}$ is in state $0^{(1)}$, or λ_2 if automaton $\mathcal{A}^{(1)}$ is in state $2^{(1)}$. If automaton $\mathcal{A}^{(1)}$ is in state $1^{(1)}$, the transition from state $0^{(2)}$ to $1^{(2)}$ does not occur (rate equal to 0). Using the SAN notation employed by the software tool PEPS2003 [3], the expression of this function is:

$$f_1 = \left[\lambda_1 \left(st(\mathcal{A}^{(1)}) == 0^{(1)}\right)\right] + \left[\lambda_2 \left(st(\mathcal{A}^{(1)}) == 2^{(1)}\right)\right]$$

The interpretation of a function can be viewed as the evaluation of an expression of non-typed programming languages, *e.g.*, C language. Each comparison is evaluated to 1 for true and to 0 for false.

Note that the use of functional rates is not limited to local event rates. In fact, for synchronizing events not only the event rate, but also the probability of occurrence can be expressed as a function. The use of functional transitions is a powerful primitive of the SAN formalism, since it allows to describe very complex behaviors in a very compact format. The computational costs to handle functional rates has decreased significantly with the developments of numerical solutions for SAN models, *e.g.*, the algorithms for generalized tensor products [2, 5].

IEEE COMPUTER SOCIETY

Figure 6 represents the equivalent Markov chain to Figure 5 according the event firing rates described in the Table 2.

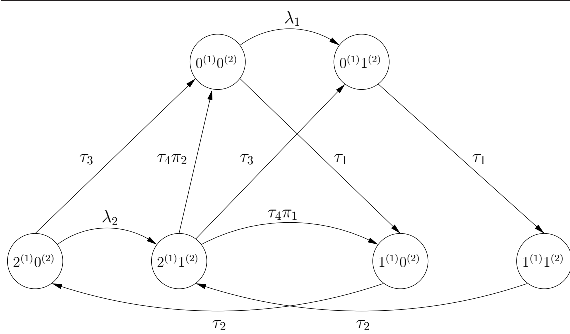

Figure 6. Equivalent Markov Chain to Figure 5.

5. Proposed Model

Figure 7 represents the SAN model which describes the parallel implementation for the propagation algorithm presented in the section 3. The SAN model contains three automata *Master*, *Buffer* and *Slices*, and P automata *Slave*$^{(i)}$ ($i = 1..P$).

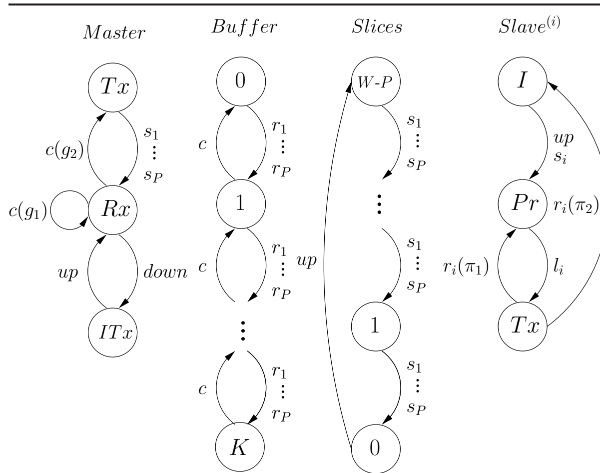

Figure 7. SAN **Model for the Propagation Algorithm.**

The automaton *Master* has three states ITx, Tx and Rx. It is responsible over the distribution of *slices* containing *seed pairs* to the slaves. The states ITx, Tx and Rx means, respectively, the initial transmission of all seed pairs to the slaves, the transmission of new slices to slaves, and the reception of *final matches* evaluated by the slaves. The occurrence of the synchronizing event up send the initial slices to all slaves. On the other hand, the occurrence of the event $down$ ends one execution of application. The event $down$ has a functional rate τ_{down}, such as:

$$\begin{aligned} \tau_{down} = &\text{ (st } Buffer \text{ == 0) \&\&} \\ &\text{ (st } Slices \text{ == 0) \&\&} \\ &\text{ (nb } I[Slave] \text{ == } N). \end{aligned}$$

The synchronizing event s_i represents the sending of a new slice to slave i. *Master* consumes final matches from the *Buffer* through the occurrence of the synchronizing event c. The event c has two functional probabilities g_1 and g_2, such as:

$$\begin{aligned} g_1 &= \text{(st } Slices \text{ == 0) || (nb } I[Slave] \text{ == 0)} \\ g_2 &= \text{(st } Slices \text{ != 0) \&\& (nb } I[Slave] \text{ != 0)} \end{aligned}$$

The automaton *Buffer* represents the final matches evaluated by the slaves. It has room enough to store all final matches received from the slaves.

The automaton *Slices* symbolizes the number of slices which will be sent to the slaves. There are $W - P$ slices remained, where W is the total number of slices and P is the total number of slaves. It is important to notice that the transition from state 0 to state $W - P$ represents the restart of new execution. Such transition is vital to a SAN description, since all SAN models must represent a problem that can be solved by a stationary solution, *i.e.*, they must be cyclic and have a steady-state.

Finally, the automaton *Slave*$^{(i)}$ represents the slave of index i, where $i = 1..P$. It has three states I (idle), Pr (processing) and Tx (transmission). Slave i finishes a final match through the occurrence of local event l_i. The synchronizing event r_i represents the reception of final matches from the slave i to *Buffer*. Slaves transmit a pack (final matches) to *Buffer* every time that it is full. The slave transmits a pack and returns to state Pr with a probability π_1 when there is more points to be evaluated. On the other hand, when there is no more points, the slave transmits a pack with probability π_2 and returns to state I.

6. Assigning Parameters

This section shows how to assign numeric values to the event rates and probabilities. Some parameters are given by the developer (input values of the model), whereas other are evaluated from those input values. We define BL as the buffer length, PS as percentage of slice extension over its neighbors (redundancy), NS as the number of slices, and

IEEE COMPUTER SOCIETY

FI as the number of final matches expected in the whole image. Some information is easily known before algorithm implementation, *e.g.*, NS, and transmission rates. Other information can only be estimated, *e.g.*, FI. Obviously, the quality of the prediction is quite dependant of the accuracy of such input parameters.

From those input values, we are able to evaluate FR (total number of final matches including redundancy due to slice extension) computing the expected average number of final matches in each slices ($\frac{FI}{NS}$) considering one *overlap* to the two edge slices ($2(1+PS)$), and two *overlap* to the inner slices ($(NS-2)(1+2PS)$):

$$FR = [2(1+PS) + (NS-2)(1+2PS)]\,\frac{FI}{NS} \quad (1)$$

Hence, it is easy to evaluate AF (average final matches for slices):

$$AF = \frac{FR}{NS} \quad (2)$$

The probabilities π_1 and π_2 of the event r_i (automaton *Slave*$^{(i)}$) is given by:

$$\pi_1 = 1 - \pi_2 \;\; and \;\; \pi_2 = min(\frac{1}{\frac{AF}{BL}}, 1) = min(\frac{BL}{AF}, 1) \quad (3)$$

However the event rate r_i is directly dependent by the transmission speed and inversely dependent by the buffer length. So the event rate r_i is given by:

$$\forall_i r_i = \frac{Tx\ Speed}{BL} \quad (4)$$

The event rate l_i is directly dependent by the buffer length and the node speed, and inversely dependent by the number of final matches unknown (non-matches). Therefore the event rate l_i is evaluated by:

$$\forall_i l_i = \frac{BL \times Node\ Speed}{Non\text{-}matches} \quad (5)$$

The event rates s_i, $down$ and up are insignificant time, therefore, they are very high rates.

Finally, the event rate c is directly dependent by the node speed and inversely dependent by the buffer length. The event rate c is given by:

$$c = \frac{Node\ Speed}{BL} \quad (6)$$

Using the software tool PEPS2003 [3], it is possible to obtain the stationary solution for the model proposed in Section 5. Many numerical results can be computed from the stationary solution of the proposed model. Some of them are quite easy to compute, *e.g.*, the average length of buffer queue, or the average number of busy slaves. Some other very important indexes can also be computed, but they require some additional effort. This is the case of the average time of one single execution.

Observing the automaton *Slices* we see a mono-cyclic behavior, *i.e.*, from any local state, there is only one possible transition. For mono-cyclic automata, the average residence time in each local state x is inversely proportional to the actual departure rate of state x, which is directly proportional to the local state probability found in the stationary solution. Therefore, it is possible to say that, if we know the average residence time of one local state of automaton *Slices* and the stationary solution, we can compute the residence time of each local state. Since the passage by all local states of automaton *Slices* represents one execution, it is possible to compute the average time one single execution in such way.

The actual rate of event *up* is independent of any contention of the others automata. In fact, it represents the rate corresponding the initial startup of the program, *i.e.*, the initial distribution of slices to the slave nodes. This information is called τ_{up} and, therefore, $t_0 = \frac{1}{\tau_{up}}$ is the average residence time in local state 0 of automaton *Slices*. Computing the stationary solution of the model we can find the probability of this local state (p_0), as well the probability of the other local states of this same automaton ($p_{\bar{0}} = 1 - p_0 = \sum_{k=1}^{W-P} p_k$). The average residence time of the other local states of automaton *Slices* ($t_{\bar{0}}$ can be computed by:

$$t_{\bar{0}} = t_0 \frac{p_{\bar{0}}}{p_0}$$

and the estimated average time of one single execution will be:

$$t_0 + t_{\bar{0}} = t_0 + t_0 \frac{p_{\bar{0}}}{p_0} = t_0 \left(1 + \frac{p_{\bar{0}}}{p_0}\right)$$

7. Conclusion

The development of the model for the parallel implementation of the propagation algorithm was a valid exercise. According to authors best knowledge, no other effort to predict performance was made so formally. The recent work of Gemund [19] exploits a formal definition used for simulations to automatically generate a formal description of a parallel implementation. More conservative works [14, 6] tend to use unappropriated formalisms (queueing networks) where the synchronization processes are not easy to describe.

The SAN formalism, instead was quite adequate to describe the quasi-independent behavior of the master and slave nodes, and also the independent representation of the buffer queue (automaton *Buffer*) and the workload, *i.e.*, slices to analyse (automaton *Slices*) was very intuitive. The definition of the events, their rates and probabilities was a little bit more difficult but still intuitive. In fact, the main difficult in the modeling process was to extract the prediction information from the stationary solution.

The most obvious performance index wanted is the one execution average time of the propagation algorithm. Unfortunately, the described SAN model must be defined by an ergodic SAN, *i.e.*, a SAN with a stationary behavior. For such model, is not easy to define the time need for one single execution. Luckily, the automaton *Slices* has a cyclic behavior, therefore the time for a complete cycle can be approached by the stationary probability distribution and the average residence time of one local state. This made possible to compute the time for one execution as presented in Section 6. Nevertheless, other performance indexes can be easily computed by the marginal probabilities of each automata local states, *e.g.*, the average number of slave nodes in use (from automata $Slave^{(i)}$), or the average length of buffer queue (from automaton *Buffer*).

The natural future work for this paper is to verify the accuracy of the proposed model by comparison with some real parallel implementations. A comparison to the numerical figures presented in [4] is not yet possible due to the absence of information concerning the machine in which the experiments were executed. Information like the interconnection speed and processing times were not defined. It is necessary to run the parallel algorithm in completely known environment to do the fine-tunning of the proposed model, *i.e.*, to verify the modeling choices made. Specially during the assigning parameters phase, some assumptions were made with a lot of guessing. This is not a particular flaw in the modeling procedure. In fact, the successive refinements in the model is a quite common technique in analytical modeling.

Finally, our experience developing a model for parallel implementation using SAN seems to be very promising. The definition of quasi-independent subsystems by automata, communication processes by synchronizing events, and independent changes by local events are very intuitive. The applicability and the usefulness of the proposed technique is yet to prove, but it is the authors opinion that the effort of exploring this possibility is worthwhile.

References

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

[2] A. Bennoit, L. Brenner, P. Fernandes, and B. Plateau. Aggregation of Stochastic Automata Networks with replicas. In *Fourth International Conference on the Numerical Solution of Markov Chains*, Urbana and Monticello, Illinois, USA, 2003.

[3] A. Benoit, L. Brenner, P. Fernandes, B. Plateau, and W. J. Stewart. The PEPS Software Tool. In *Proceedings of the* 13th *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, Urbana and Monticello, Illinois, USA, 2003. Springer-Verlag.

[4] L. G. Fernandes. *Parallélisation d'un Algorithme d'Appariement d'Images Quasi-dense*. PhD thesis, Institut National Polytechnique de Grenoble, France, 2002.

[5] 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.

[6] E. Gelenbe, R. Lent, A. Montuoria, and Z. Xu. Cognitive Packet Network: QoS and Performance. In *10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'02)*, Fort Worth, Texas, October 2002.

[7] E. Gelenbe and H. Shachnai. On G-Networks and Resource Allocation in Multimedia Systems. In *IEEE Research in Data Engineering*, pages 104–110, Orlando, Florida, February 1998.

[8] C. Harris and M. Stephens. A Combined Corner and Edge Detector. In *Proceedings of the 4th Alvey Vision Conference*, pages 147–151, Manchester, UK, 1988.

[9] L. Hu and I. Gorton. Performance Evaluation for Parallel Systems: A Survey. Technical Report 9707, University of NSW, Sydney, Australia, 1997.

[10] W. M. Jr. Modeling Performance of Parallel Programs. Technical Report 589, The University of Rochester, Rochester, New York, 1995.

[11] W. S. Kendall. Perfect simulation for the area-interaction point process. In L. Accardi and C. Heyde, editors, *Probability Towards 2000*, pages 218–234, Manchester, UK, 1988. Springer Verlag New York.

[12] M. Lhuillier and L. Quan. Image Interpolation by Joint View Triangulation. In *Proceedings of the International Conference on Computer Vision and Pattern Recognition*, pages 139–145, Fort Collins, Colorado, USA, 1999.

[13] M. Lhuillier and L. Quan. Robust Dense Matching using Local and Global Geometric Constraints. In *Proceedings of the 15th International Conference on Pattern Recognition*, pages 968–972, September 2000.

[14] D. A. Menascé and V. A. F. Almeida. *Capacity planning for web services: metrics, models, and methods*. Prentice Hall, 2002.

[15] O. Monga. An Optimal Region Growing Algorithm for Image Segmentation. *International Journal of Pattern Recognition and Artificial Intelligence*, 1(3):351–375, 1987.

[16] B. Plateau. *De l'Evaluation du Paralllisme et de la Synchronisation*. PhD thesis, Paris-Sud, Orsay, 1984.

[17] C. Schimid, R. Mohr, and C. Bauckhage. Comparing and Evaluating Interest Points. In *Proceedings of the 6th International Conference on Computer Vision*, pages 230–235, Bombay, India, 1998.

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

[19] A. J. C. van Gemund. Symbolic Performance Modeling of Parallel Systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(2):154–165, 2003.

0-7695-2046-4/03 $17.00 © 2003 **IEEE**

IEEE COMPUTER SOCIETY