Performance models for *master/slave* parallel programs [1]

Lucas Baldo [2] Leonardo Brenner [3] Luiz Gustavo Fernandes [4]
Paulo Fernandes [5] Afonso Sales [6]

Faculdade de Informática
Pontifícia Universidade Católica do Rio Grande do Sul
Porto Alegre, Brazil

Abstract

This paper proposes the use of Stochastic Automata Networks (SAN) to develop models that can be efficiently applied to a large class of parallel implementations: *master/slave* (m/s) programs. We focus our technique in the description of the communication between master and slave nodes considering two standard behaviors: *synchronous* and *asynchronous* interactions. Although the SAN models may help the pre-analysis of implementations, the main contribution of this paper is to point out advantages and problems of the proposed modeling technique.

Key words: performance evaluation, analytical models, stochastic automata networks, parallel programming, propagation algorithm.

1 Introduction

Performance evaluation is, or at least should be, a primary concern of parallel program developers. Unfortunately, most of this concern does not appear early enough in the development process. Execution trace techniques and visualization tools [23,13,22] offer important information about performance of an existing parallel implementation, but if you have a trace of execution much of the implementation effort was already done. Benchmarks [2,10] contribute to generic, and yet useful, information about an execution platform. However,

[1] This work was developed in collaboration with HP Brazil R&D.
[2] Email: lbaldo@inf.pucrs.br
[3] Email: lbrenner@inf.pucrs.br
[4] Email: gustavo@inf.pucrs.br
[5] Email: paulof@inf.pucrs.br (corresponding author - the order of authors is merely alphabetical)
[6] Email: asales@inf.pucrs.br

in the absence of a formal approach to model the interaction of a designed parallel program, it can be very hard to correctly estimate the behavior of its implementation before starting the programming phase.

To cope with this problem, we propose the use of modeling formalisms to describe and analyse the designed implementation and its platform. We believe that such modeling may furnish good estimation on the implementation behavior in quite early stages of the parallel program development. The main difficulty in doing so is the (usually great) complexity in developing performance models. In fact, the development of an accurate model demands as much understanding of parallel programs as performance issues. Thus, it would be very interesting to relieve the parallel programmers from this performance analysis skills burden.

Although far from an automated generator of performance models [21], we propose some modeling techniques that can be efficiently applied to a large class of parallel implementations: the *master/slave* (m/s) programs, *e.g.*, *bag of tasks* solutions [1]. We focus our technique in the description of the communication between master and slave nodes considering two standard behaviors: *synchronous* and *asynchronous* interaction. Our experience in the matter suggests that the adequate modeling of the communication is the key issue to develop models for such parallel programs. This paper contribution is based on the real case study of a classical parallel application to, hopefully, describe in the same manner any m/s parallel program implementation.

The next section informally presents the *Stochastic Automata Networks* (SAN) formalism [17] with a minimum of details. Section 3 briefly describes the concept of m/s programs and the possible kinds of interaction between master and slave nodes, as well as it presents the generic m/s models using the SAN formalism. To illustrate our approach, Section 4 presents a practical case example: the implementation choices for a parallel version of the Propagation Algorithm. Section 5 shows the SAN models describing the possible implementations of the Propagation Algorithm. Section 6 presents the analysis of performance indices for the presented models. Finally, the conclusion summarizes our contribution and suggests future works.

2 Stochastic Automata Networks

The SAN formalism was proposed by Plateau [16] and its basic idea 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 framework proposed by Plateau defines a modular way to describe continuous and discrete-time Markovian models [17]. However, only continuous-time SAN will be considered in this paper, although discrete-time SAN can also be employed without any loss of generality.

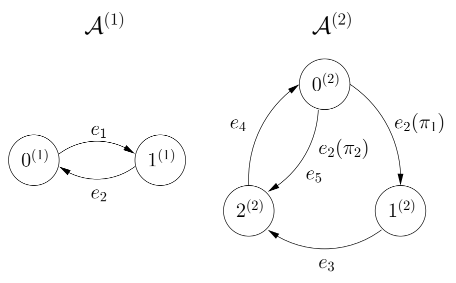
The SAN formalism describes a complete system as a collection of subsystems that interact with each other. Each subsystem is described as a stochastic automaton, *i.e.*, an automaton in which the transitions are labeled with probabilistic and timing information. Hence, one can build a continuous-time stochastic process related to SAN, *i.e.*, the SAN formalism has exactly the same application scope as Markov Chain (MC) formalism [20,6]. The state of a SAN model, called *global state*, it is defined by the cartesian product of the *local states* of all automata.

There are two types of events that change the global state of a model: *local events* and *synchronizing events*. Local events change the SAN 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. Actually, local events can be viewed as a particular case of synchronizing events that concerns only one automaton.

Each event is represented by an *identifier* and a *rate* of occurrence, which describes how often a given event will occur. Each transition may be fired as result of the occurrence of any number of events. In general, non-determinism among possible different events is dealt according to Markovian behavior, *i.e.*, any of the events may occur and their occurrence rates define how often each one of them will occur. However, from a given local state, if the occurrence of a given event can lead to more than one state, then an additional *routing probability* must be informed. The absence of routing probability is tolerated if only one transition can be fired by an event from a given local state.

The other possibility of interaction among automata is the use of functional rates. Any event occurrence rate may be expressed by a constant value (a positive real number) or a function of the state of other automata. In opposition to synchronizing events, functional rates are one-way interaction among automata, since it affects only the automaton where it appears.

Fig. 1 presents a SAN model with two automata, four local events, one synchronizing event, and one functional rate. In the context of this paper, we will use roman letters to identify the name of events and functions, and greek letters to describe constant values of rates and probabilities.



Type	Event	Rate
loc	e_1	f
syn	e_2	μ
loc	e_3	σ
loc	e_4	δ
loc	e_5	τ

$$f = [(st\ \mathcal{A}^{(2)} == 0^{(2)}) * \lambda] + [(st\ \mathcal{A}^{(2)} == 2^{(2)}) * \gamma]$$

Fig. 1. Example of a SAN model

In the model of Fig. 1, the rate of the event e_1 is not a constant rate, but a functional rate f described by the SAN notation [7] employed by the PEPS tools [4]. The functional rate f is defined as:

$$f = \begin{cases} \lambda \text{ if automaton } \mathcal{A}^{(2)} \text{ is in the state } 0^{(2)} \\ 0 \text{ if automaton } \mathcal{A}^{(2)} \text{ is in the state } 1^{(2)} \\ \gamma \text{ if automaton } \mathcal{A}^{(2)} \text{ is in the state } 2^{(2)} \end{cases}$$

The firing of the transition from $0^{(1)}$ to $1^{(1)}$ state occurs with rate λ if automaton $\mathcal{A}^{(2)}$ is in state $0^{(2)}$, or γ if automaton $\mathcal{A}^{(2)}$ is in state $2^{(2)}$. If automaton $\mathcal{A}^{(2)}$ is in state $1^{(2)}$, the transition from $0^{(1)}$ to $1^{(1)}$ state does not occur (rate equal to 0).

The use of functional expressions is not limited to event rates. In fact, routing probabilities also may be expressed as functions. The use of functions 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 the SAN models, *e.g.*, the algorithms for generalized tensor products [4]. Fig. 2 presents the equivalent MC model to the SAN model in Fig. 1.

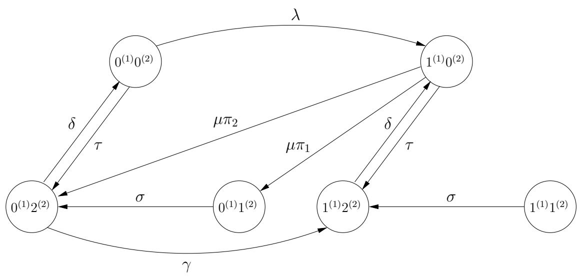


Fig. 2. Equivalent Markov Chain to the SAN model in Fig. 1

It is important to notice that only 5 of the 6 states in this MC model are reachable. In order to express the reachable global states of a SAN model, it is necessary to define a (*reachability*) function. For the model in Fig. 1, the reachability function must exclude the global state $1^{(1)}1^{(2)}$, thus:

$$\textit{Reachability} = !\left[\left(st\ \mathcal{A}^{(1)} == 1^{(1)}\right) \&\& \left(st\ \mathcal{A}^{(2)} == 1^{(2)}\right)\right]$$

3 Master/Slave paradigm for parallel programs

In the m/s paradigm, one master node generates and allocates work, usually called *tasks*, to N slave nodes. Each slave node receives tasks, computes them,

[7] 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 value 1 (*true*) and value 0 (*false*).

and sends the result back. Thus, master node is responsible for receiving and analyzing the computed results by slave nodes (*summation*).

This paradigm is generally suitable for shared-memory or message-passing platforms, since the interaction is naturally two-way. The main problem with m/s paradigm is that the master may become a *bottleneck*. This may happen if the tasks are too small or if there are too many slaves connected to one master. The choice of the amount of work in each task is called *granularity*. One of the main performance issues in parallel programming is the choice between many small tasks (*fine grain*) or few large tasks (*coarse grain*).

According to how the master distributes tasks and collects results from the slaves, we present two different kinds of m/s implementations: programs with *synchronous* or *asynchronous* interaction[8]. A program is said to have synchronous interaction when the tasks need to be performed in phases, *i.e.*, all tasks in each phase must finish before the distribution of next phase tasks. In opposition, the interaction between the master and the slaves is called *asynchronous* when the master distributes new tasks every time a slave finishes its computation. Such interaction may reduce waiting time considering a non-deterministic distribution of slave requests (tasks have an unknown computational cost). In this context, the master node usually implements a waiting buffer in order to deal with simultaneous arrivals of results from slaves.

3.1 Generic model for m/s parallel programs

Regardless of interaction type, the SAN generic models to m/s parallel programs are composed of an automaton to each node. *Master* automaton (see Fig. 3) is composed of three states:

- ITx : representing the initial transmission of tasks to slave nodes;
- Tx : representing the transmission of new tasks to slave nodes; and
- Rx : representing the reception of computed results from slave nodes.

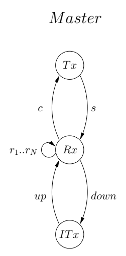


Fig. 3. Master automaton

Each automaton $Slave^{(i)}$ (see Fig. 4), where $i = 1 \dots N$, is also composed of three states:

- I : representing the slave node without tasks to process (idle);

[8] Readers must pay attention that we are *not* talking about synchronicity in the programming primitives level.

- Pr : representing the slave node processing a task; and
- Tx : representing the slave node transmitting results back to the master node.

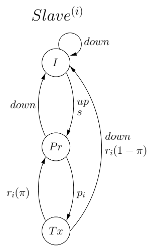


Fig. 4. Slaves automata

There are six distinct events in the generic model composed of those $1+N$ automata: *up*, s, p_i, r_i, c, and *down*. The program execution begins with event *up*. Event s corresponds to the sending of tasks to the slaves. Event p_i represents the processing stop of a task by the i^{th} slave. After this stop, event r_i indicates the sending of results from the i^{th} slave back to the master. If the i^{th} slave needs to continue the task execution, events p_i and $r_i(\pi)$ will occur successively until the ending of the task execution. On the other hand, when there is no more points to be evaluated, the slave transmits a pack through the occurrence of event $r_i(1-\pi)$ and returns to idle state. The numerical value of π must be estimated according to information known by the parallel programmer. After receiving a result from one of the slave nodes, the master node performs the summation represented by event c. When all needed work were done, event *down* represents the ending of the parallel application and the returning to the initial situation. Note that event *down* may happen even if the slaves are in Pr or Tx states.

According to specific characteristics of each modeled application, some additional features must be added to this generic model. For example, in asynchronous models, event s must be split in N events s_i ($i = 1..N$), each one representing the sending of a task to the i^{th} slave. Synchronous models have only one event s, because all slaves receive tasks at the same time. Also, the passage of results from slaves to master is usually performed using a buffer. In that case an additional automaton may be included and the events r_i and c will not directly synchronize master with slaves, but slaves with buffer, and buffer with master respectively (see asynchronous model in Section 5.1).

The numerical value for the rate of each event will depend on a variety of characteristics, *e.g.*, problem size, granularity, nodes processing power, communication speed, *etc.* This mapping of real case information to numerical values, called *parameterization*, is a crucial point in the model development. Unfortunately, such mapping seems quite dependent on the particularity of each implementation. In Section 5, we provide two practical cases of parameterization, one considering a synchronous implementation, and the other considering the analogous asynchronous version of the same application.

4 Case study: The Propagation Algorithm

Image-based interpolation is a method to create smooth and realistic virtual views between two original view points. One of the major computational cost phases of this method is the one related to the construction of a dense matching map between the original scenes [5]. In this section, we briefly discuss some relevant details about the implementation of a region growing algorithm called Propagation [14]. Later, a parallel version for this algorithm is also briefly presented.

4.1 Basic Propagation Algorithm

The Propagation Algorithm is based on a classic region growing method for image segmentation [15] which uses pixel homogeneity. However instead of using pixel homogeneity property, a similar measure based on the matches correlation score is adopted [14]. 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, which allows good propagation even though they are not local maxima.

Points of interest [12,19] 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.

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* that 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* that contains all 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.

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

Algorithm 1. The Propagation Algorithm

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

current set of seed pairs. Then, the algorithm looks for new matches in its match neighborhood. 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 Fig. 5. The squared regions in both images show the extension of the matched regions obtained from the seed matches.

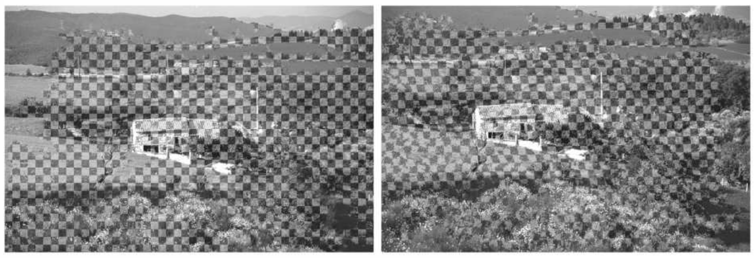


Fig. 5. Propagation example using the House pair

4.2 Parallel implementation

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 presented in the previous section, the Propagation Algorithm advances by comparing neighbors pixels through out the source images. From some seed pairs, it can form large matching regions on both images. In fact, a single seed pair can start a propagation that grows through a large region over the images. 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 assumes that the algorithm knows the entire surface of the images. This can create a situation in which several processors are propagating over the same regions at the same time creating a redundancy of computation (Fig. 6).

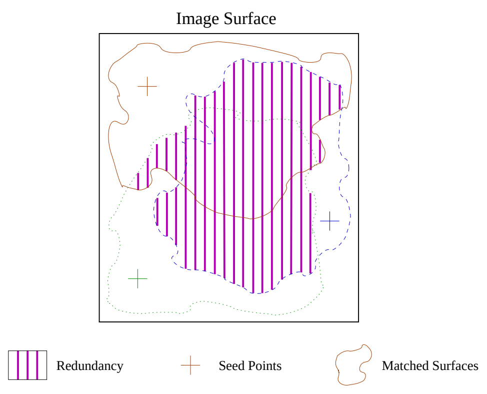


Fig. 6. Redundancy problem

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 policy in order to achieve a more efficient parallel solution.

The solution proposed in [9] is based on a master-slave paradigm. One processor (master) will be responsible for distributing the work and centralizing the final results. The others (slaves) 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 (coordinates of target slice). The master distributes the seed pairs over the nodes considering their location over the slices (see Fig. 7). This procedure replaces the global best-first strategy by several local best-first ones. Each local seed pairs subset 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.

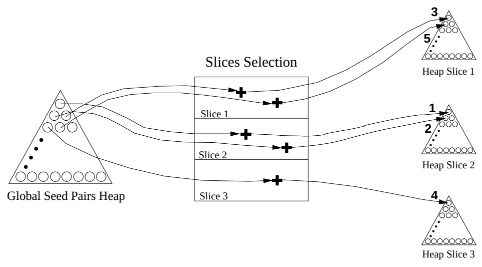


Fig. 7. Seed pairs heap distribution over the slices

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. As said before, 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. 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 limitation is partially solved by a technique called flexible slices [9]. This technique allows the Propagation Algorithm to expand through the surface of its neighbor slices in a controlled way. As shown on Fig. 8, 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.

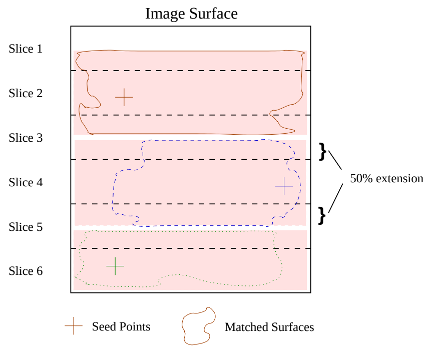


Fig. 8. Flexible slices approach

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

(i) the master divides the set of seed pairs into subsets based on their location over the slices;

(ii) each slave receives one slice with its associated subset;

(iii) each slave computes its own subset of seed pairs;

(iv) when there is no more seed pairs to compute, the slave sends a signal to the master;

(v) if there are available slices remaining, the master send it to an available slave;

Actually, the master has a queue of slices, organized by their position over the images. 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 subset (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 unavoidable duplicated ones. To send these final matches to master, each slave has a communication buffer which is filled progressively as the Propagation Algorithm advances. A buffer, when it is full, is sent to master and 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. When a slave reaches the end of its seed pairs subset, 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 subset) back to it (if there are available subsets).

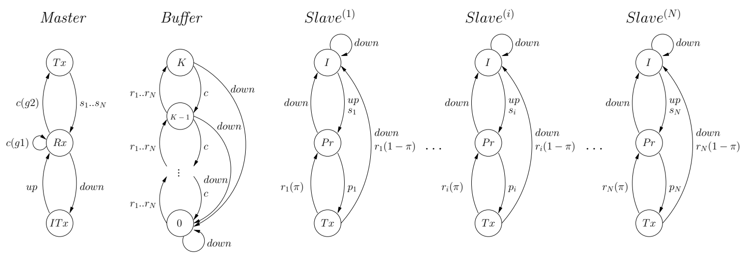
5 Modeling examples

This section proposes two distinct models to two quite different approaches to parallelisation of the Propagation Algorithm presented previously.

Our analysis of the sequential Propagation Algorithm indicates that a m/s implementation would be more efficient using asynchronous interaction between the master and the slaves, *i.e.*, the master does not distribute and collect results from the slaves in phases. In this section, we use SAN to model both synchronous and asynchronous m/s implementations in order to confirm (in Section 6) if our pre-implementation analysis of the Propagation Algorithm is corrected.

5.1 Asynchronous implementation model

Fig. 9 presents the SAN asynchronous model which contains one *Master* automaton, one *Buffer* automaton, and N $Slave^{(i)}$ ($i = 1..N$) automata.



Type	Event	Rate
syn	*up*	λ
syn	*down*	μ
syn	c	σ
syn	s_i	δ
syn	r_i	α
loc	p_i	γ

$g_1 = (nb[Slave^{(1)}..Slave^{(N)}]I == 0)$ $g_2 = (nb[Slave^{(1)}..Slave^{(N)}]I > 0)$

Fig. 9. SAN asynchronous model for the Propagation Algorithm

Master automaton is responsible for the distribution of slices (*tasks*) to slaves and analysis of final matches (*results*) evaluated by them. This automaton has three states (ITx, Tx and Rx) which mean respectively: the initial transmission of all seed pairs to the slaves; the transmission of new slices to the slaves; and the reception of final matches evaluated by the slaves. Synchronizing event *up* sends the initial slices to all slaves, starting a new program execution. On the other hand, synchronizing event *down* ends one execution of an application. The occurrence of this event indicates that all automata must change their actual state for the initial one.

Synchronizing event s_i represents the sending of a new slice to the i^{th} slave. Automaton *Master* consumes final matches evaluated by the slaves through synchronizing event c, consuming results from a repository (automaton *Buffer*). Since this SAN model has asynchronous characteristics, the results generated by the slave nodes are stored on this repository. Therefore, master node does not need to synchronize the reception of results sent by slaves.

When synchronizing event c occurs, the master node may have two behaviors. In the first one, indicated by functional probability [9] g_1, master stays on receiving and analyzing results. In the second one, functional probability g_2 indicates that a slave node is in idle state, so master must send him a new slice.

Finally, automaton $Slave^{(i)}$ represents the i^{th} slave and it has three states: I (*idle*), Pr (*processing*) and Tx (*transmission*). $Slave^{(i)}$ finishes a final match through the occurrence of local event p_i. Synchronizing event r_i represents the reception of final matches from $Slave^{(i)}$ by *Buffer*. The slave transmits a pack (final matches) and returns to Pr state with a probability π, when there still exists points to be evaluated. On the other hand, when there is no more points to be evaluated, the slave transmits a pack with probability $1 - \pi$ and returns to idle state. The numerical values of π and the rates of all events must be estimated according to information known by the parallel programmer. In the next section, we present how to map this information into model parameters.

[9] Using the SAN notation employed by the PEPS tools [4], the command $nb[Slave^{(1)}..Slave^{(N)}]I$ computes how many automata $Slave^{(i)}$ ($i = 1..N$) are in I (idle) state.

5.1.1 Assigning parameters

This section shows how to assign numerical values to the event rates and probabilities. Some parameters are given by the developer (input values of the model), whereas other are evaluated using those input values. We will describe the choice of those values considering the processing of the image in Fig. 5, and the characteristics of an homogeneous COW (Cluster of Workstations) with Pentium III 1.0 GHz and Myrinet network. Obviously, other environments or other workloads would have different input parameters.

Some variables were created to help the parameter definition. Let BL be the slaves buffer length, which was fixed with corresponding value of $64KB$ size ($\simeq$ 5.500 points). The percentage of slices extension of one slave over its neighborhoods (PS) was also fixed with value 0.5. Another fixed value is the number of points (NP) on the input image: 230.000 points[10]. Obviously, the quality of the prediction is quite dependant on the accuracy of such input parameters. Finally, number of slices (NS) describes the number of tasks in which the problem will be split (granularity).

From those input values, we can calculate the real number of points (RNP) to be evaluated (NP plus redundancy):

$$RNP = \left[2 * (1 + PS) + (NS - 2) * (1 + 2PS)\right] * \frac{NP}{NS} \tag{1}$$

Hence, it was easy to estimate the average number of buffers (NB) for each slice:

$$NB = \left\lceil \frac{RNP}{BL * NS} \right\rceil \tag{2}$$

Probability π of the event r_i (automaton $Slave^{(i)}$) is given by:

$$\pi = \frac{NB - 1}{NB} \tag{3}$$

Rates for events r_i, p_i, s_i and c were obtained with some statistical measurements performed through some sample programs over the target architecture. Rate of synchronizing events s_i (δ) and r_i (α) correspond, respectively, to send a new task (TT) to a slave and send back a result pack (TP) containing final matches, given by:

$$\delta = \frac{1}{TT} \quad and \quad \alpha = \frac{1}{TP} \tag{4}$$

In a similar way, rate of events p_i (γ) and c (σ) correspond to the time spent on computing final matches (CM) and evaluating results (ER) respectively. Those event rates are defined by:

$$\gamma = \frac{1}{CM} \quad and \quad \sigma = \frac{1}{ER} \tag{5}$$

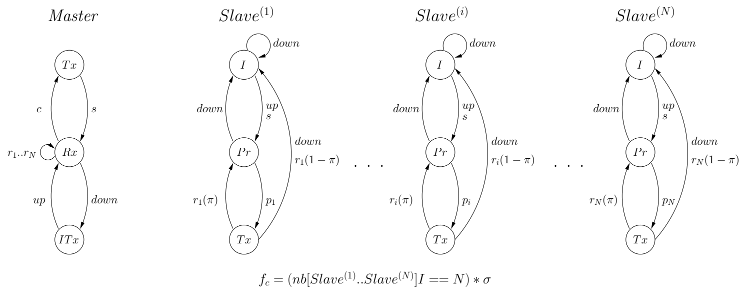
[10] This number of points was taken from the real image presented in Fig. 5.

Synchronizing event *down* (μ) has an insignificant time, so an arbitrary high rate ($\mu = 1000$) was adopted. Finally, synchronizing event *up* is associated with the time spent on sending tasks to slaves, similar as event s_i. Thus, its rate (λ) can be obtained by:

$$\lambda = \frac{1}{TT * N} \text{ , } where\ N = number\ of\ slaves \tag{6}$$

5.2 *Synchronous implementation model*

Fig. 10 presents the SAN synchronous model for the parallel implementation of the Propagation Algorithm. SAN synchronous model is quite similar to SAN asynchronous version. However, it is not necessary to consider a repository buffer (automaton *Buffer*), since the master node forces all slaves to finish their tasks before it starts a new task distribution.



Type	Event	Rate
syn	*up*	λ
syn	*down*	μ
loc	c	f_c
syn	s	δ
syn	r_i	α
loc	p_i	γ

Fig. 10. SAN synchronous model for the Propagation Algorithm

Since there is no buffer to store the slave results, which characterizes asynchronous model, master node is now responsible for reception of all packs evaluated by the slaves. Hence, events $r_1..r_N$ (automaton *Master*) describe this new behavior.

Additionally, synchronous model has all tasks in each phase finishing before the distribution of next phase tasks. This behavior is indicated by local event c, which has a functional rate (f_c) indicating that a new task distribution only must start when all slaves are in idle state. Synchronizing event s describes a new task distribution, *i.e.*, the starting of a new phase.

5.2.1 *Assigning parameters*

All events, which do not have any change from asynchronous to synchronous model, preserve their rates (*up*, *down*, s, and p_i). For the other ones, minimal changes have been done. Event c is now a local event and it has a functional rate (function f_c indicated in Fig. 10), expressing the waiting for synchronized distribution of new phase tasks.

Events $s_1..s_N$ were replaced for just one synchronizing event s. Since event s is related to a *broadcast* of new tasks, and inasmuch the time for a broadcast is similar to an *unicast*, its rate was preserved.

As the master node centralizes the receiving of all pack results, two or more slaves can send their results to master at the same time. Hence, the sending time of a slave declines drastically. Thus, rate of events $r_1..r_N$ (α) are now defined by:

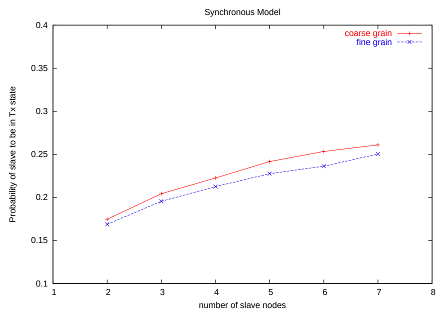
$$\alpha = \frac{1}{TP * N} \tag{7}$$

6 Performance indices

In order to validate the strategy of the implementation chosen and modeled using SAN, some models were made altering some parameters, *e.g.*, automaton *Buffer* size, number of slices (tasks), and number of slaves. With the results of those models, three kind of analysis could be done: synchronous *vs.* asynchronous; small buffer *vs.* large buffer; and fine grain *vs.* coarse grain. Despite of having no doubt about which approach (synchronous or asynchronous) fits better, the two models were solved and their theoretical results were compared. Finally, the size of automaton *Buffer* and the granularity were analyzed.

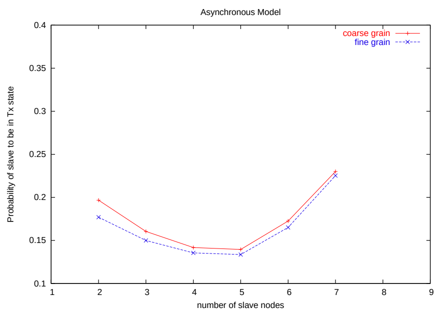
6.1 Synchronous vs. Asynchronous

Fig. 11 presents a comparison between synchronous and asynchronous models, observing the probability of a slave to be in Tx state. First analysis about synchronous model shows the probability of a slave to be (or trying to be) transmitting gets higher while the number of slaves grows. This analysis indicates that slaves stay more time transmitting, which is a coherent result, whereas master node centralizes all the receives. The behavior in asynchronous model is quite different. Up to five slaves, increasing the number of slaves do not result in an increasing Tx probability. This indicates an advantage for asynchronous implementation, since slaves can effectively work more time in parallel. We believe that it happens due to a smaller loss in transmission of result packs.



Number of Slaves	2	3	4	5	6	7
Coarse Grain	0.1746	0.2042	0.2225	0.2416	0.2533	0.2609
Fine Grain	0.1687	0.1954	0.2125	0.2275	0.2362	0.2503

(a)

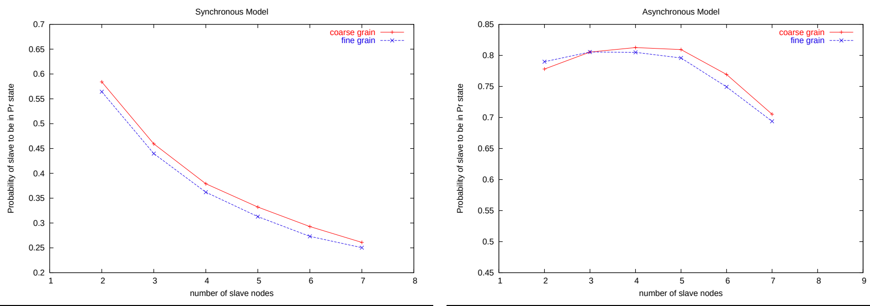


Number of Slaves	2	3	4	5	6	7
Coarse Grain	0.1967	0.1604	0.1417	0.1395	0.1725	0.2300
Fine Grain	0.1769	0.1499	0.1355	0.1336	0.1651	0.2253

(b)

Fig. 11. Probability of slaves to be in Tx state

This analysis can be confirmed by looking at the probability of a slave to be in *Pr* state. As it can be seen in Fig. 12 (b), the probability of *Pr* state (computing tasks) decreases as the number of slaves grows in asynchronous implementation.



Number of Slaves	2	3	4	5	6	7
Coarse Grain	0.5840	0.4594	0.3790	0.3322	0.2929	0.2609
Fine Grain	0.5643	0.4398	0.3620	0.3128	0.2731	0.2503

(a)

Number of Slaves	2	3	4	5	6	7
Coarse Grain	0.7780	0.8053	0.8126	0.8093	0.7692	0.7052
Fine Grain	0.7898	0.8054	0.8049	0.7958	0.7492	0.6938

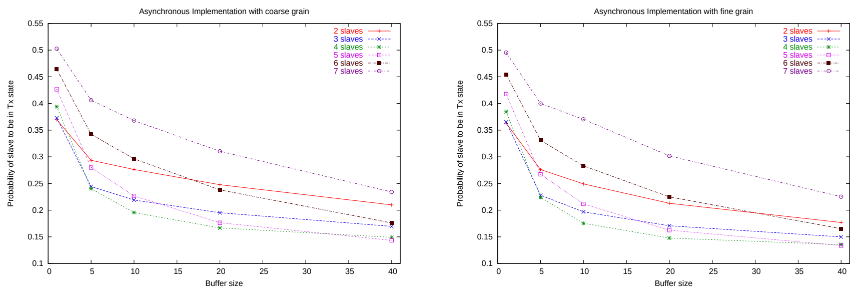
(b)

Fig. 12. Probability of slaves to be in *Pr* state

Additionally, performance gains once again keep increasing up to five slaves. This analysis do not indicate that we have a worse execution time with more than five slaves, but at this point the speed up starts to decline. The performance evaluation based on the SAN models indicates better results for asynchronous implementation of the Propagation Algorithm. Therefore, the next experiments will only consider asynchronous implementations.

6.2 Small buffer vs. Large buffer

For asynchronous model, an important parameter is the size of automaton *Buffer* (master node waiting queue). Fig. 13 shows curves for asynchronous model with fine and coarse grain.



Buffer size	2 Slaves	3 Slaves	4 Slaves	5 Slaves	6 Slaves	7 Slaves
1	0.3696	0.3729	0.3940	0.4264	0.4643	0.5025
5	0.2935	0.2443	0.2404	0.2799	0.3423	0.4061
10	0.2763	0.2190	0.1956	0.2266	0.2963	0.3680
20	0.2477	0.1953	0.1668	0.1765	0.2380	0.3103
40	0.2099	0.1696	0.1492	0.1434	0.1761	0.2341

(a)

Buffer size	2 Slaves	3 Slaves	4 Slaves	5 Slaves	6 Slaves	7 Slaves
1	0.3630	0.3652	0.3846	0.4176	0.4542	0.4954
5	0.2760	0.2280	0.2237	0.2672	0.3308	0.3999
10	0.2493	0.1969	0.1754	0.2116	0.2830	0.3703
20	0.2129	0.1708	0.1478	0.1626	0.2249	0.3017
40	0.1769	0.1499	0.1355	0.1336	0.1651	0.2253

(b)

Fig. 13. Buffer size for asynchronous model

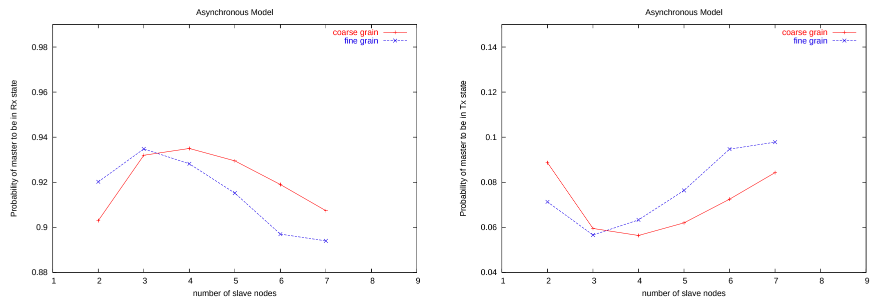
In both cases, a small buffer (less than 10 positions) shows slaves remaining too much time in Tx state (waiting for buffer space) and, consequently, less time calculating. On the other hand, increasing the buffer size indefinitely does not result in a better performance. The best buffer size depends directly on the number of slaves used. A 20-sized buffer seems to be enough using 2, 3, and 4 slaves. However, using more slaves, a larger buffer may be required, *e.g.*, a 40-sized buffer seems the most appropriated for 7 slaves. Therefore, for the next experiments we will consider a 40-sized buffer.

As can be seen in Fig. 13, probabilities presented for 1-sized buffer are almost the same as for synchronous model. This is a coherent result since synchronous model could be approximated by an asynchronous model with 1-sized buffer.

6.3 Fine grain vs. Coarse grain

In parallel applications, the granularity is one of the most important features. In our study, granularity is represented by the number of slices (NS) in which the input image is partitioned. We use two different grains defined through the number of slices allowed per node: coarse grain (2 slices per node) and fine grain (3 slices per node).

Looking at Fig. 11 (b) and 12 (b), it is not possible to point out which granularity is better. Using fine grain, slaves spent less time calculating (Pr state) and less time transmitting (Tx state). On the other hand, using coarse grain, the situation is the inverse.



Number of Slaves	2	3	4	5	6	7
Coarse Grain	0.9030	0.9320	0.9350	0.9295	0.9190	0.9074
Fine Grain	0.9202	0.9348	0.9282	0.9152	0.8970	0.8940

(a)

Number of Slaves	2	3	4	5	6	7
Coarse Grain	0.0887	0.0595	0.0564	0.0620	0.0725	0.0843
Fine Grain	0.0713	0.0566	0.0633	0.0764	0.0947	0.0978

(b)

Fig. 14. Granularity for asynchronous model

In fact, the main aspect to identify the best granularity is related to automaton *Master*. For parallel implementation of the Propagation Algorithm described in Section 4.2, it is clear that, if master spends too much time transmitting, it quickly becomes the bottleneck of the application. Based on this assertion, the best granularity would be the one where master remains most of its time in Rx state. Increasing the number of slave nodes (see Fig. 14 (a)), master node remains more time receiving information with coarse grain. Such analysis can be confirmed in Fig. 14 (b) in which the probability of master to be in Tx state is presented. Therefore, from our asynchronous model, we can assume that using coarse grain will result in a better speed up for the parallel implementation.

7 Conclusion

The modeling of such parallel programs is greatly facilitated by SAN formalism primitives. However, any other structured formalism, *e.g.*, *Stochastic Petri Nets* [7], Stochastic Activity Networks [18], and *Process Algebra* [11] could be employed. Thus, according to the programmer previous experience, the use of another formalism could be considered without any loss of generality. Balbo, Donatelli and Franceschinis, for example, had use a quite similar approach using SPN formalism [3]. In fact, the choice of SAN as modeling formalism was somewhat driven by our own experience with previous modeling examples [8]. Our main point is to show that the modeling of parallel programs can be generalized according to a given structured formalism, which was *Stochastic Automata Networks* for this paper.

The use of an automaton to describe each processing node (being a master or slave) and the buffer automaton (for asynchronous models) or a single synchronizing event (for synchronous models) seems quite intuitive. Such simplicity is the greater advantage of our modeling technique.

The major drawback of this approach is the importance left to the parameterization phase, *i.e.*, to the choice of rates for the model events. Much of the accuracy of the model rely on the mapping of user known characteristics to the rate numerical values of the events. In fact, the modeling technique may only help the programmer by posing the questions of what event rates should be informed. It remains on the programmer shoulders the burden of taking all important parameters into account.

The great advantages in our technique are the normal benefits of having a formal model for the application. The indication of possible *bottlenecks* may help the programmer to pay more attention to those specific points during the implementation. Results as those presented in the previous section may clearly point out the better number of tasks according to the number of processing nodes. Such information about the best granularity may be useful to a scheduler, or a processing node provider, to automatically choose how many nodes in a cluster or grid should be allocated to a given application.

The obtained results for the models of the Propagation Algorithm were not yet confirmed by a direct comparison with an actual implementation. Nevertheless, our results match the expectations of the algorithm programmers. Thus, the comparison with actual implementations, and other modeling experiences, are the natural future works to pursue. Equally, we may foresee the development of an user friendly tool to facilitate the construction of models to programmers with few, or none, performance evaluation skills. Despite the lack of more extensive practical experiences, we believe that the use of SAN analytical models for parallel implementations seems an useful and economic viable helping theoretical tool to programmers.

References

[1] G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Wesley, University of Arizona, USA, 2000.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[3] G. Balbo, S. Donatelli, and G. Franceschinis. Understanding parallel programs behavior through Petri nets models. *Journal of Parallel and Distributed Computing*, 15(3):171–187, 1992.

[4] A. Benoit, L. Brenner, P. Fernandes, B. Plateau, and W. J. Stewart. The PEPS Software Tool. In Peter Kemper and William H. Sanders, editors, *Computer Performance Evaluation / TOOLS 2003*, volume 2794 of *Lecture Notes in Computer Science*, pages 98–115, Urbana, IL, USA, 2003. Springer-Verlag Heidelberg.

[5] L. Brenner, L. G. Fernandes, P. Fernandes, and A. Sales. Performance Analysis Issues for Parallel Implementations of Propagation Algorithm. In *Proceedings of the* 15th *Symposium on Computer Architecture and High Performance Computing*, pages 183–190, São Paulo, November 2003.

[6] L. Brenner, P. Fernandes, and A. Sales. The Need for and the Advantages of Generalized Tensor Algebra for Kronecker Structured Representations. *International Journal of Simulation: Systems, Science & Technology*, 5(5-6):1–13, December 2004.

[7] G. Ciardo and K. S. Trivedi. A Decomposition Approach for Stochastic Petri Nets Models. In *Proceedings of the* 4th *International Workshop Petri Nets and Performance Models*, pages 74–83, Melbourne, Australia, December 1991. IEEE Computer Society.

[8] A. G. Farina, P. Fernandes, and F. M. Oliveira. Representing software usage models with Stochastic Automata Networks. In *Proceedings of the* 14th *International Conference on Software Engineering and Knowledge Engineering*, pages 401–407. ACM Press, 2002.

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

[10] V. Getov, A. Hey, R. Hockney, and I. Wolton. The Genesis Benchmark Suite: Current State and Results. In *Proceedings of Workshop on Performance*

Evaluation of Parallel Systems - PEPS'93, University of Warwick, Coventry, november 1993.

[11] S. Gilmore, J. Hillston, L. Kloul, and M. Ribaudo. PEPA nets: a structured performance modelling formalism. *Performance Evaluation*, 54(2):79–104, 2003.

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

[13] E. Kraemer and J. T. Stasko. The Visualization of Parallel Systems: An Overview. *Journal of Parallel and Distributed Computing*, 18(2):105–117, 1993.

[14] M. Lhuillier. *Modlisation pour la synthse d'images partir d'images.* Thse de doctorat en informatique, Institut National Polytechnique de Grenoble, France, 2000.

[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. On the stochastic structure of parallelism and synchronization models for distributed algorithms. In *Proceedings of the 1985 ACM SIGMETRICS conference on Measurements and Modeling of Computer Systems*, pages 147–154, Austin, Texas, USA, 1985. ACM Press.

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

[18] W. H. Sanders and J. F. Meyer. Reduced Base Model Construction Methods for Stochastic Activity Networks. *IEEE Journal on Selected Areas in Communications*, 9(1):25–36, 1991.

[19] 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.

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

[21] A. J. C. van Gemund. Compiling performance models from parallel programs. In *Proceedings of the 8th international conference on Supercomputing*, pages 303–312. ACM Press, 1994.

[22] A. Waheed and D. T. Rover. Performance Visualization of Parallel Programs. In *IEEE Conference on Visualization*, pages 174–181, San Jose, USA, October 1993.

[23] J. C. Yan, S. Sarukkai, and P. Mehra. Performance measurement, visualization and modeling of parallel and distributed programs using the AIMS toolkit. *Software Practice and Experience*, 25(4):429–461, 1995.