Performance Issues for Parallel Implementations of Bootstrap Simulation Algorithm

Ricardo M. Czekster, Paulo Fernandes, Afonso Sales and Thais Webber
Pontifícia Universidade Católica do Rio Grande do Sul
Av. Ipiranga, 6681 – Prédio 32 – 90619-900 – Porto Alegre, Brazil
{ricardo.czekster, paulo.fernandes, afonso.sales, thais.webber}@pucrs.br

Abstract—The solution of state-based stochastic models is usually a demanding application, then it is a natural subject to high performance techniques. We are particularly interested in the speedup of Bootstrap Simulation of structured Markovian models. This approach is a quite recent development in the performance evaluation area, and it brings a considerable improvement in the results accuracy, despite the intrinsic effect of randomness in simulation experiments. Unfortunately, Bootstrap Simulation has higher computational cost than other alternatives. We present experiments with different options to optimize the parallel solution of Bootstrap Simulation applied to three practical examples described in Stochastic Automata Networks (SAN) formalism. This paper contribution resides in the discussion of theoretical implementation issues, the obtained speedup and the actual processing and communication times for all experiments. Additionally, we also suggest future works to improve even more the proposed solution and we discuss some interesting insights for parallelization of similar applications.

Keywords-Stochastic Modeling; Discrete-Event Simulation; Performance Evaluation; Parallel Applications;

I. INTRODUCTION

Simulation approaches are normally used when numerical solution reaches its computational limits or when analytical models are just too complex to be created. The literature presents several approaches to use simulation as a reliable tool for performance estimations and model verification. Many examples employ the generation of discrete events combining them with non-trivial data structures that save important information regarding the simulation execution [1], [2], [3], [4].

In Markovian simulation contexts, a fundamental objective is to generate independent samples for latter statistical analysis. Since every experiment does not depend on each other, they can be executed in parallel, distributing the burden of generating lots of samples to achieve a given accuracy. This simulation technique is known as parallel sampling and it consists of a master-worker architecture that uses clusters of computers. The master entity is responsible for generating and unifying the samples scattered in parallel and distributed environments whereas the workers calculate samples, as optimized as they possible can.

Traditional and Markovian simulation are slightly different, having different ways to generate and process the samples. In Markov Chains simulation, and for simulation of all state-based models [5], one can use a transition matrix to derive a possible sequence of states, also called a random walk, or a simulation trajectory [6]. However, due to the potentially huge size of the transition matrix and subsequent memory cost, simulation may be an impractical option for massively large models. To cope with this limitation, structured formalisms offer high-level descriptions of the underlying Markov chain, usually reducing the transition matrix to workable memory limits. That was the major concern of the Stochastic Automata Networks (SAN) formalism [7]: to efficiently store the transition matrix using tensor algebra [8].

In recent years we have witnessed the adoption of different simulation techniques such as Monte Carlo [9], Perfect sampling [10] and even in traditional simulation [11] there is room for improvements and broader discussions. In fact, there are several advances regarding the parallel sampling technique for structured Markov Chains. For instance, in [12], the authors were interested in the relation of the number of samples to be produced in comparison with traditional simulation and bootstrap simulation. Bootstrap simulation demonstrates remarkable results to improve results precision, however, its usage is impaired due to high computational costs to generate repeated batches of samples. The aim of this paper is to present a parallel implementation of bootstrap simulation technique and its performance issues, focusing on the solution of structured Markovian models. Our purpose is to enhance the overall bootstrap technique performance by presenting a method to generate larger amounts of samples in less time, improving results accuracy for further model inspections.

The remainder of this paper is presented as follows. We show basic modeling concepts based on Markov Chains and SAN, followed by principles of simulation in Section II. We present the Bootstrap simulation method applied in the context of Markovian simulation and discuss about its accuracy and computational cost in Section III. Section IV
II. MARKOVIAN MODELING AND SIMULATION

When performing advanced performance indices analysis, it is very common to recur to Markovian modeling. Such representations are equipped with simple entities composed by states and transitions among states. The transitions in Markov Chains (MC) [6] represent how the system states alternate and also how frequently this occurs. Despite its simplicity, MC usually requires large amounts of states to represent complex systems. This problem is referred in the literature as state space explosion and several efforts are directed to diminish its catastrophic implications, e.g., memory exhaustion. That is the main reason for the existence of various Markovian structured formalisms that have been proposed throughout the years.

The basic principles of structured formalisms rely on composition of subsystems, or modularization techniques, in order to describe a large system. The objective is to reduce the system complexity by dividing it in more manageable parts using high-level modeling primitives. It is worth mentioning that all Markovian structured formalisms are intrinsically based on MC, i.e., an underlying representation is always present [4].

A. SAN – Stochastic Automata Networks

There are several examples to cite such as Stochastic Petri Nets [3], Markovian Process Algebras such as Performance Evaluation Process Algebra (PEPA) [2] and Stochastic Automata Networks (SAN) [7], to name a few. We will be interested in the latter, although all this paper conclusion could be generalized to all other structured formalisms without any loss of generality.

The notion of automata is central when choosing SAN as a modeling tool for real world situations. An automaton is an entity that behaves more or less independently with other automata. Each entity possess both local and synchronizing transitions, respecting a set of rules in order to be fired. Events can be mapped to rates indicating the frequency of change among predefined states for a continuous time MC, and to probabilities for a discrete representation [6].

In SAN, rates can be of two types, constant or functional. The first type corresponds to scalar values whereas functional rates depend on other automata states to fire a given transition. The functional primitive is a powerful mechanism to depict variability in complex systems as changes normally occurs depending on external factor (the states of other automata).

Given that Markovian simulation can be used to approximate solution and that structured representations are associated with an underlying MC, it is possible to conclude that the combination of both can produce relevant research in the context of performance evaluation of complex systems, a fact already observed in other discussions outside the scope of this paper [12].

Figure 1 shows an example of a stochastic automata network model composed of two automata and five events. Note that it is possible to fire transitions locally in each automaton such as the events $e_4$ and $e_5$ in the first automaton ($A^{(1)}$), as well as $e_2$ and $e_3$ in the second automaton ($A^{(2)}$). The event $e_2$ is a local event to $A^{(2)}$, but it has a functional dependency related to automaton $A^{(1)}$. This model also defines a synchronizing event $e_1$, which corresponds to fire a transition from state 1 to state 0 in the automaton $A^{(1)}$ and from state 0 to state 1 in the automaton $A^{(2)}$.

![Figure 1. Example of a SAN model.](image)

The SAN example in Figure 1 can be represented by its underlying Markov chain (Figure 2) composed of six states and transitions correspondent to the firing of events in global states of the SAN. In fact, each state of this Markov chain corresponds to a combination of local states of the SAN model. Since there is a Markov chain representation for each SAN model, it is possible to derive a transition matrix to obtain the numerical solution of the system described by the SAN model. For example, Table I is the transition matrix for the model described in Figure 1.

B. Simulation of structured Markovian models

In this context we have distinct ways to solve any given model, e.g., using numerical methods or simulation. Numerical methods often rely on direct or iterative mathematical techniques such as the LU factorization [13], SOR [14],

---

**Table I. Transition matrix for the SAN model.**

<table>
<thead>
<tr>
<th>Type</th>
<th>Event</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>syn</td>
<td>$e_1$</td>
<td>$\mu_0$</td>
</tr>
<tr>
<td>loc</td>
<td>$e_2$</td>
<td>$f$</td>
</tr>
<tr>
<td>loc</td>
<td>$e_3$</td>
<td>$\mu_3$</td>
</tr>
<tr>
<td>loc</td>
<td>$e_4$</td>
<td>$\mu_4$</td>
</tr>
<tr>
<td>loc</td>
<td>$e_5$</td>
<td>$\mu_5$</td>
</tr>
</tbody>
</table>
Position probabilities $P$ for each state, where every step produces a set of trajectories and counting the amount of visits for models [11], [9]. Such methods are based on simulating the acceleration of iterative methods [8]. Another alternative is to use traditional simulation methods for Markovian models. The transition function $\phi$ is generically described by the following expression:

$$
\phi(s_t, U) = \begin{cases} 
   s_0 & \text{for } U \in [0, P_{00}) \\
   s_1 & \text{for } U \in [P_{00}, P_{00} + P_{10}) \\
   \vdots & \vdots \\
   s_j & \text{for } U \in \left\{ \sum_{l=0}^{j-1} P_{il}, \sum_{l=0}^{j} P_{il} \right\} \\
   \vdots & \vdots \\
   s_r & \text{for } U \in \left\{ \sum_{l=0}^{r-1} P_{ir}, 1 \right\}
\end{cases}
$$

This stochastic matrix is derived from the discretization of the continuous time transition matrix. A simulation method also needs a pseudorandom generation function $U$ uniformly distributed in $(0, 1)$ [9]. Every transition between two states occurs according to a transition function given by $\phi(s_t, U)$, dictating how the random walking takes place for the set of valid states defined by the SAN model. The transition function $\phi$ is a simulation technique.

III. Bootstrap simulation technique

Traditional simulation employs simple techniques to derive approximated measures of interest. In a Markovian context, the main idea is to perform a random walking procedure given the set of possible states that the system assumes and, from an initial state, jump to another state, if a transition is defined. Each simulation path is termed a simulation trajectory and the user must choose the way to compute the performance indices, accumulating the number of times it visits each state until a trajectory length is reached. There is a myriad of options to choose how the trajectory length is chosen, in this paper we will remain on the simplest form which is the arbitrary choice of a number of transitions. It is normal to run the simulation for long trajectories in order to improve the accuracy, since the precision is directly related to the number of samples that was produced. When the simulation run stops, a sample or the whole trajectory is stored for later statistical analysis.

Bootstrap is a well known statistical technique [17] applied to many fields to improve accuracy when performing sample estimations for complex distributions. The core method execution works with sets of unknown data comprising a potentially infinite sized population termed $\tilde{\Lambda}$. The
objective is to discover the distribution in random samplings \( \Lambda \) of size \( n \) directly extracted from the population, i.e., \( \Lambda \subset \bar{\Lambda} \). When observing \( \Lambda \), the \( n \) samples are drawn with probability \( \frac{1}{n} \) and that is proved to represent the unknown population as closely as possible. That helps improving the methods accuracy since the main feature of using bootstrap is to work with sample replacement, i.e., the obtained values could be repeated throughout the process. The technique bases its execution on more reliable estimations for the set of considered samples, improving the results for a number of examples. For more comprehensive examples and a more detailed view of the whole bootstrap simulation process, we suggest the reader to consult [12].

We are interested in extending the transition function to take into account the bootstraps. In one hand, one could benefit from being able to analyze huge sized Markovian models using simulation and on the other hand, one could improve the sampling procedure to return more reliable and precise results.

Blending Markovian simulation and bootstrapping can enable the determination of massive state spaces with reasonable accuracy and less computational efforts to produce the performance indices. After the consideration of Markovian simulation, we expect to translate the same gains to structured representations of Markov Chains, i.e., Stochastic Automata Networks. Using the compositional aspects and modularization already available when modeling in SAN, we clearly extend the amount of different realities that can be used to represent complex systems interactions.

Algorithm 1 presents the application of the Bootstrap in the context of Markovian simulation. In this algorithm, variables and vectors used in the method are initialized between lines 1 and 4. The samples counted in the bootstraps for all trajectories of length \( n \) are shown from line 6 to line 16. Note that the variable \( z \) defines the number of bootstraps considered. Between lines 17 and 25, it is calculated the permanence probabilities of states for each bootstrap. For the remaining part of the algorithm (lines 26-31), the average probabilities of every state in vector \( \pi \) are adjusted according the information contained in the bootstraps.

Thus, \( n \times z \) samplings take place for every trajectory step, comparing each pseudorandom value against an arbitrarily chosen constant value \( \alpha \) between 0 and \( n - 1 \). If the pseudorandom generated value is equal to \( \alpha \), the state is accounted in the correspondent bootstrap \( K \). This procedure is repeated for all bootstraps until a predefined trajectory length \( n \) is reached. However, we know that the random generation of \( n \) samples for every trajectory step is computationally too expensive. Previous work [12] demonstrates that it is possible to only perform \( \bar{n} \) trials, where \( \bar{n} \ll n \).

Bootstrap simulation is suitable to obtain more accurate results, however it has a high computational cost, firstly generating \( z \) bootstraps, and secondly, integrating them in the final probability vector. The simulation time is largely augmented when compared to the traditional simulation executing \( n \) steps, because bootstrap simulation includes the execution of \( n \times z \times \bar{n} \) trials. Due to this, one can devise parallel algorithms for the Bootstrap simulation, focusing on the distribution of bootstraps to enhance the execution time. Moreover, once the samples can be generated in a faster manner, the number of samples could also be augmented in order to improve results accuracy.

IV. PARALLEL BOOTSTRAP SIMULATION

In Markovian simulation, every sample corresponds to an independent draw, making it very ineffective for sequential computation. That is the main reason why parallel implementations are such advantageous in these context. In every simulation, it is a well known fact that one must produce as many samples as possible to enhance the overall simulation quality and, therefore, its precision. When simulating any given reality, the objective is to generate as many samples as possible, employing every available resource to pursue that specific objective.

```
Algorithm 1 Bootstrap simulation method

1: \( \alpha \leftarrow U(0,\bar{n} - 1) \) \{constant value \( \alpha \) initialization with a pseudorandom value chosen between 0 and \( \bar{n} - 1 \)
2: \( \pi \leftarrow 0 \) \{initialization of the probability vector \( \pi \}
3: \( K \leftarrow 0 \) \{initialization of all z bootstraps \( K \}
4: \( s_c \leftarrow s_0 \) \{set \( s_c \) as initial state \( s_0 \}
5: \{walk on a trajectory of length \( n \}
6: for \( t = 1 \) to \( n \) do
7: \( s_d \leftarrow \phi(s_c, U(0,1)) \) \{finds destination state \( s_d \) from \( s_c \) according to \( U(0,1) \}
8: for \( b = 1 \) to \( z \) do
9: for \( c = 1 \) to \( \bar{n} \) do
10: if \( U(0,\bar{n} - 1) == \alpha \) then
11: \( K_b[s_d] \leftarrow K_b[s_d] + 1 \) \{counts in \( K_b[s_d] \) every time the sample equals to \( \alpha \}
12: end if
13: end for
14: end for
15: \( s_c \leftarrow s_d \) \{current state \( s_c \) is updated to \( s_d \}
16: end for
17: for \( b = 1 \) to \( z \) do
18: \( \omega \leftarrow 0 \)
19: for \( i = 1 \) to \(|S|\) do
20: \( \omega \leftarrow \omega + K_b[i] \) \{calculates in \( \omega \) the total sum of accumulated values in \( K_b \}
21: end for
22: for \( i = 1 \) to \(|S|\) do
23: \( \bar{x}_b[i] \leftarrow \frac{K_b[i]}{\omega} \) \{calculates the probability of \( i \)-th state in \( K_b \}
24: end for
25: end for
26: for \( i = 1 \) to \(|S|\) do
27: for \( b = 1 \) to \( z \) do
28: \( \pi[i] \leftarrow \pi[i] + \bar{x}_b[i] \)
29: end for
30: \( \pi[i] \leftarrow \frac{n[i]}{z} \) \{calculates the average probability from the bootstraps\}
31: end for
```
The parallelization approach adopted in this paper was to split the bootstrap sampling tasks over the processing nodes. In this approach all nodes will perform the full trajectory simulation, but each node will produce a different set of samples. It is worthy to notice that this is not the only parallelization option, for example, another choice was to split the trajectory simulation among nodes. However, we believe that our parallelization choice is, at the same time, simple enough to code, and it preserves the logical sequence of states of the sequential implementation.

Our chosen parallel execution model follows a master-worker pattern where entities known as workers performs the sampling generation procedure whereas the master organizes all produced data and accumulating in its corresponding probability vector entries where it uniformizes results. In the workers side, when a sample is computed, it is stored locally and, when the amount of collected samples reaches a predefined required quantity set by the master entity, it is readily sent to the master.

From a practical point of view, the implementation of the parallel version of the bootstrap simulation was made using C++ language and MPI primitives. The sequential and parallel tests were executed in a cluster architecture containing eight homogeneous nodes Dell PowerEdge R610 connected in a Gigabit Ethernet network. Each node is composed of two processors Intel Xeon E5520 QuadCore 2.27 GHz, Hyper Threading technology, 16GB memory, under Linux O.S. The prototype was compiled using gcc version 4.2.4 and MPICH library version 1.2.7p1.

A. The tested SAN models

Considering the background presented for Markovian simulation, following we present three SAN models with different characteristics [18], [19], [20]: Alternate Service Patterns (ASP), First Available Server (FAS) and Resource Sharing (RS).

ASP model describes an Open Queueing Network [6] with servers that map different service patterns. The model has four queues represented by four automata, an additional automaton representing the service patterns. This model has $(K + 1)^4 \times P$ reachable states, where $K$ is the capacity of the queues and $P$ the number of service patterns.

FAS model indicates the availability of $N$ servers, where every server is composed of a two state automaton, representing the two possible server conditions: available or busy. In the model, requests are firstly assigned to the first server. If the server is busy, the task must be assigned to the second server and so on, i.e., the first available server is assigned to the request. This model has $2^N$ reachable states.

The classical RS model maps $R$ shared resources to $P$ processes. Each process is represented by an automaton with two states: idle or busy. The number of available resources is represented by a function that only grants access to the busy state if there is less than $R$ process in the busy state. This model represents $2^P$ states and, due to the nature of the model transitions, the number of reachable states is just $\sum_{i=0}^{R} \binom{i}{i}$, where $\binom{i}{i}$ is the number of $i$-combination of a $P$-sized set, i.e.: $\binom{i}{i} = \frac{P!}{i!(P-i)!}$

These models were used in this paper in two sets: one with very small models, and another with considerably large models.

The small models were parametrized as follows: ASP model - every queue with capacity two ($K = 2$) and two service patterns ($P = 2$); FAS model - with nine servers ($N = 9$); and RS model - with 10 processes ($P = 10$) and five resources ($R = 5$), giving respectively models with reachable state space sized, respectively, 162, 512 and 638 states.

The large models were parametrized as follows: ASP model - every queue with capacity fifty ($K = 50$) and four service patterns ($P = 4$); FAS model - with twenty five servers ($N = 25$); and RS model - with 25 processes ($P = 25$) and fifteen resources ($R = 15$), giving respectively models with reachable state space sized around, respectively, 27, 33 and 29 million states.

B. The conducted experiments

In order to verify the efficiency of the parallel implementations a set of experiments was tested over the eight nodes parallel machine described, running a bootstrap simulation of all examples over one (sequential implementation), 2, 3, 4, 5, 6, 7, and 8 nodes. The experiments were conducted considering 36 bootstraps ($z = 36$) and trajectory lengths ($n$) of $10^6$, $10^7$, $10^8$, and $10^9$. Table II describes how the 36 bootstraps were split among the nodes. It is important to notice that the number of bootstraps to each node was defined trying to obtain the more balanced distribution of tasks.

<table>
<thead>
<tr>
<th>configuration</th>
<th>number of bootstraps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>36</td>
</tr>
<tr>
<td>2</td>
<td>18 18</td>
</tr>
<tr>
<td>3</td>
<td>12 12 12</td>
</tr>
<tr>
<td>4</td>
<td>9 9 9 9</td>
</tr>
<tr>
<td>5</td>
<td>7 7 7 7 8</td>
</tr>
<tr>
<td>6</td>
<td>6 6 6 6 6 6 6 6 6 6 6</td>
</tr>
<tr>
<td>7</td>
<td>5 5 5 5 5 5 5 5 5 5 5</td>
</tr>
<tr>
<td>8</td>
<td>4 4 4 4 4 4 4 4 5 5 5</td>
</tr>
</tbody>
</table>

Figures 3 and 4 presents, respectively, the time spent in seconds for execution of the large and small version of models. There are one chart to each trajectory length (from $10^6$ to $10^9$). Inside each chart there are three sets of bars representing each model (ASP, FAS and RS). At each set of bars representing one model, there are eight bars representing the running over each node configuration.
Each of these bars are split in two colors section, one representing the time spent in processing the bootstrap simulation tasks (in red), and another representing the time spent in communication among nodes (in blue).

All results presented were computed considering the average of 30 trials taking a 95% confidence intervals into account. Therefore, the time in seconds presented in $Y$ axis is statistically validated.

Observing Figure 3, we notice a quite poor optimization for running bootstrap simulation with a $10^6$ trajectory length. And this bad result is explained by the considerable increase in the communication cost in configuration with a higher number of nodes.

However, increasing the trajectory length to $10^7$, the processing time demands increase, but the communication costs remains approximatively the same. It is important to observe that the $Y$-axis scale increases of an order of magnitude as the the trajectory length increases. The overall speedup improvement while the trajectory length increases is clearly noticeable observing the charts in Figure 3, since the increase of processing time demands turns the communication demands irrelevant.

Observing Figure 4, it is possible to confirm the perception of the major drives for the parallel implementation of bootstrap simulation. The small models have very small reachable state spaces, and therefore the communication costs are irrelevant. In fact, very small probability vectors (162, 512 and 638 for these models) need to be passed from the worker nodes to the master. In opposition to the large model were probability vectors of around 30 million position need to be transmitted.

Observing the processing time requirements for small and large models it is possible to observe that the small and large models have quite similar demands. That surprising fact indicates that the processing demand is bounded almost only by the trajectory length, and it is independent of the model size.

V. CONCLUSION

It is an established fact that the process of deriving samples independently are trivial in terms of parallelization efforts. When dividing an independent task among several processors, it is clear that the performance will be better than the pure sequential version. In respect to that matter, we observe that the computational complexity issues and related open problems to tackle such classes of problems are shifted towards two concerns: i) faster and more efficient ways to compute each sample in each work unit; and ii) adapting the parallel sampling techniques to orchestrate efforts with structured Markovian models such as SAN. The solution of such problems will directly impacts the time and produced accuracy and several works are presented to mitigate such simulation constraints.
The first contribution of this paper was an efficient parallel implementation of a novel simulation algorithm achieving a considerable speedup for very large models, i.e., tens of millions states, specially when quite long trajectories were necessary, i.e., billions of transitions. Additionally, the speedup was consistent with different SAN models, delivering for all models a nearly 5 times speedup for a 8 node configuration.

Another important contribution resides in the comprehension that the processing demands depends only on the simulation trajectory length, while the communication demands depends only on the reachable state space size of the model. Such insight allows a better choice of configuration according to the simulation to perform.

It was not the objective of this work to analyze the simulation accuracy improvement achieved by the bootstrap simulation. This work focus was on the parallelization of the bootstrap algorithm, assuming that the benefits of the sequential implementation would remain the same. However, this is not necessarily the case, since some precision issues may arise from the split computation of probability vectors. Also, the possibility to run longer trajectories with smaller time costs let us dream with an enhanced accuracy that was not possible to the sequential implementation. In fact, a further study of the impacts of the parallel implementation on the accuracy of the bootstrap simulation is a natural future work to this paper.

Another interesting subject of research is the further study of bootstrap distribution over non uniform memory architecture, since some levels of shared memory among processing nodes could be highly beneficial to cope with high communication costs found in short trajectory lengths for large models. Such future work may improve the application scope of bootstrap simulation to distributed platforms, and not only clusters of processors as the experiments in this paper.

Finally, it is also possible to foresee improvements in the bootstrap simulation itself in order to consider a broader global system dynamics rather than only inspecting the global state. Another aspect that we expect to direct our attention concerns the application of variance reduction techniques to produce even more accurate results. We envision also the combination of parallel sampling with more sophisticated simulation approaches, such as Perfect Simulation or Perfect Sampling.

Nevertheless, the results already achieved by the parallel version of bootstrap simulation allows quite important time savings in the solution of massively large systems described as SAN models that were impossible to solve with the numerical software tools available. That fact alone justifies the research effort made.
REFERENCES


