

Performance Study of Distributed Generation of State Spaces Using Colored Petri Nets

W.M. Zuberek

Department of Computer Science
Memorial University
St. John's, Canada A1B 3X5

Abstract

The performance of many distributed applications depends upon the ratio of computation to communication times. In the case of distributed generation of state spaces for timed Petri nets, this ratio is determined by the partitioning function which assigns generated states to classes associated with processors; if the partition classes correspond to clusters of states with only a few connections between clusters, the required communication is minimized, and the performance is maximized. The effects of state clustering are analyzed by simulating a colored timed Petri net modeling the distributed state space generation.

1. Introduction

Actual implementation of complex, real-world systems is usually preceded by thorough studies, performed on a formal model of the original system. For systems which exhibit concurrent activities, Petri nets are a popular choice of the modeling formalism, because of their ability to express concurrency, synchronization, precedence constraints and nondeterminism. Moreover, Petri nets “with time” (stochastic or timed) include the durations of modeled activities and this allows to study the performance aspects of the modeled system [1, 14, 21].

The basic analysis of net models, based on exhaustive generation of all reachable states, is known as reachability analysis. In reachability analysis, the states of the model and the transitions between states are organized in the so called reachability graph which is used for verifying the required qualitative properties (such as absence of deadlocks or liveness). For timed and stochastic Petri nets (with deterministic or exponentially distributed times), this graph is a Markov chain, so its stationary probabilities of states can be determined using known numerical methods [19]. These stationary probabilities are used to derive performance measures of the model [1, 6, 21].

For large net models, the state space can easily exceed the resources of a single computer system. The availability of clusters of (inexpensive) workstations and portable libraries for distributed computing make distributed generation of the state space quite an attractive alternative to the traditional sequential approach.

In distributed generation of the reachability graph, the (yet unknown) state space is partitioned into n disjoint regions, R_1, R_2, \dots, R_k , and these regions are constructed independently by k identical processes running concurrently on different machines. At the end, the regions can be integrated in one state graph if needed.

There are several approaches to the distributed generation of reachability graphs. Many results and implementation details for parallel reachability graph generation on shared-memory multiprocessors are given in [2, 3, 4, 5]. In shared-memory systems, however, the

processors are tightly connected, so inter-processor communication is quite efficient, which makes such systems significantly different from clusters of workstations or PCs. Distributed state space generation described in [12] is organized into a sequence of phases, and each phase contains processing of currently available states followed by communication in which non-local states are sent to their regions (i.e., processors). The next phase begins only after completion of all operations of the previous phase. Reasonable values of speedup were reported for small numbers of processors (2 to 4) and for large state spaces.

The purpose of this paper is to present a simple colored Petri net model of a generation of the reachability graphs for timed Petri nets using a cluster of processors connected by a switch, and to illustrate the effects of state space partitioning on the performance of distributed generation. In particular, it has been observed that for some types of timed Petri nets, the straightforward partitioning algorithm [16, 17] results in poor speedups, so a more sophisticated partitioning algorithm, taking into account structural properties of the model, may be required to improve the performance of distributed state space generation.

Section 2 provides a brief overview of basic concepts of timed Petri nets which are relevant to this paper. Section 3 outlines the distributed generation of the state space for timed Petri nets. A simple colored timed Petri net model of a cluster of processors connected by a switch is presented in Section 4, while Section 5 discusses the results obtained from the presented model. A simplified approach to performance evaluation is outlined in Section 6. Section 7 contains several concluding remarks.

2. Timed Petri Nets

A timed Petri net is a triple $\mathcal{T} = (\mathcal{M}, c, f)$ where \mathcal{M} is a marked (place/transition) net, $\mathcal{M} = (P, T, A, m_0)$, with P denoting the set of places, T – the set of transitions, A – the set of directed arcs, $A \subseteq P \times T \cup T \times P$, and m_0 – the initial marking function, $m_0 : P \rightarrow \{0, 1, 2, \dots\}$. \mathcal{M} can be extended in a number of ways if needed; for example, it can include a set of inhibitor arcs $B \subset P \times T$, $\mathcal{M} = (P, T, A, B, m_0)$, $A \cap B = \emptyset$, or multiple arcs described by a weight function $w : A \rightarrow \{1, 2, \dots\}$, $\mathcal{M} = (P, T, A, w, m_0)$, and so on. c is a conflict-resolution function which assigns probabilities to conflicting transitions, $c : T \rightarrow [0, 1]$, in such a way that for each free-choice class of transitions the sum of assigned probabilities is equal to 1. For more general conflicts, the probabilities assigned to transitions represent relative frequencies of transition firings, which are used to determine the probabilities of state transitions [10]. f is the firing time function which assigns the firing times to transitions, $f : T \rightarrow \mathbf{R}^+$, where \mathbf{R}^+ denotes the set of nonnegative real numbers; if firing times are constant, the nets are called D-timed; if they are random variables with (negative) exponential distributions (described by their average values), the nets are called M-timed (or Markovian); other distributions can also be used.

In timed nets, it is assumed that each transition which can fire, initiates its firing in the same instant of time in which it becomes enabled, and the firings of transitions occur in “real time”, i.e., the tokens are removed from the input places at the beginning of the firing interval, and they are deposited into the output places at the end of this interval. Consequently, the behavioral description of a timed net must take into account the places (i.e., the remaining tokens) as well as the firing transitions of a net [21]. For D-timed nets, a state of a net, s , is a triple, $s = (m, n, r)$, where m is a marking function, $m : P \rightarrow \{0, 1, 2, \dots\}$, n is a firing function, $n : T \rightarrow \{0, 1, 2, \dots\}$, which indicates, for each transition, the number of its firings occurring in this state, and r is a (partial) remaining-firing-time function, $r : T \rightarrow (\mathbf{R}^+)^*$, which indicates, for each firing of each transition, the remaining time of firing (D-timed nets do not have the memoryless property, so the r

components of state descriptions represents a “history” of firings; for M-timed nets, this last component of state descriptions is not used at all).

For example, let the firing times associated with the transitions of the D-timed net shown in Fig.2.1 be $f(t_1) = 1.0$, $f(t_2) = 0.9$, $f(t_3) = f(t_4) = 0.1$, and $f(t_5) = 0.8$.

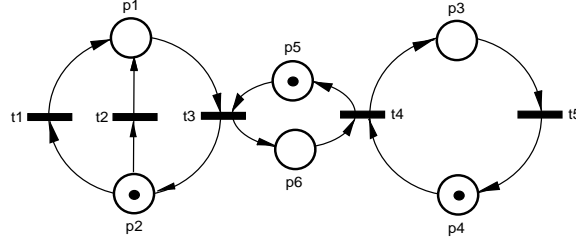


Fig.2.1. D-timed net.

For the initial marking shown in Fig.2.1, the net has two initial states, one corresponding to t_1 's firing and the second corresponding to t_2 's firing; these two initial states are as follows (the m , n and r components of states are separated by semicolons; undefined values of r are indicated by ‘-’):

$$s_1 = (0, 0, 0, 1, 1, 0; 1, 0, 0, 0, 0; 1.0, -, -, -, -),$$

$$s_2 = (0, 0, 0, 1, 1, 0; 0, 1, 0, 0, 0; -, 0.9, -, -, -).$$

The holding times (or durations) of s_1 and s_2 are equal to 1.0 and 0.9 (time units) respectively; at the end of the firing time, a token is deposited in p_1 , so t_3 can initiate its firing, which is represented by s_3 :

$$s_3 = (0, 0, 0, 1, 0, 0; 0, 0, 1, 0, 0; -, -, 0.1, -, -).$$

The holding time of s_3 is 0.1, at the end of which tokens are deposited in p_2 and p_6 , enabling t_1 , t_2 and t_4 , so there are two possible next states, s_4 with t_1 and t_4 firing concurrently, and s_5 with t_2 and t_4 firing concurrently:

$$s_4 = (0, 0, 0, 0, 0, 0; 1, 0, 0, 1, 0; 1.0, -, -, 0.1, -),$$

$$s_5 = (0, 0, 0, 0, 0, 0; 0, 1, 0, 1, 0; -, 0.9, -, 0.1, -).$$

Both these states have holding times equal to 0.1, at the end of which the firing of t_4 ends (while t_1 or t_2 continues its firing, with the remaining firing time equal to 0.9 and 0.8, respectively), and t_5 initiates its firing, creating states s_6 (from s_4) and s_7 (from s_5):

$$s_6 = (0, 0, 0, 0, 1, 0; 1, 0, 0, 0, 1; 0.9, -, -, -, 0.8),$$

$$s_7 = (0, 0, 0, 0, 1, 0; 0, 1, 0, 0, 1; -, 0.8, -, -, 0.8).$$

The holding times of s_6 and s_7 are both 0.8. At the end of s_6 , t_1 continues to fire (with the remaining firing time equal to 0.1) creating state s_8 :

$$s_8 = (0, 0, 0, 1, 1, 0; 1, 0, 0, 0, 0; 0.1, -, -, -, -),$$

while the end of s_7 corresponds to the termination of its both firings, tokens are deposited in p_1 and p_4 , and t_3 can initiate its firing; this is s_3 again.

The state graph (or the reachability graph) for this example is shown in Fig.2.2 with the state holding times shown in parentheses).

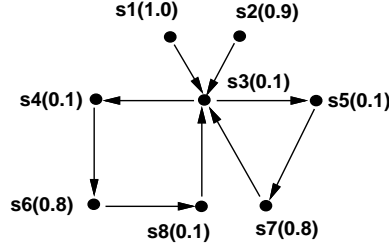


Fig.2.2. State graph for the net shown in Fig.2.1.

It should be noticed that for different values of firing times, the reachability graph for the net shown in Fig.2.1 may be significantly different from the one shown in Fig.2.2.

Timed colored Petri nets are extensions of place/transitions timed nets; a timed colored Petri net is also a triple $\mathcal{T} = (\mathcal{M}, c, f)$ where \mathcal{M} is a colored net, $\mathcal{M} = (P, T, C, A, e, m_0)$, with P , T and A as before, C denoting a set of attributes called “colors”, e assigning expressions (over colors and color variables) to arcs of the set A , the initial marking function extended to $m_0 : P \times C \rightarrow \{0, 1, 2, \dots\}$, and the choice and firing time functions extended to: $c : T \times B \rightarrow [0, 1]$ and $f : T \times B \rightarrow \mathbf{R}^+$, where B denotes the set of bindings, i.e., assignments of colors from the set C to variables used in arc expressions.

An outline of a “standard” approach to the (sequential) generation of state space (just the states, without arcs) can be as follows:

1. State space generation:
2. **var** *States* := \emptyset ; (* set of states *)
3. *unexplored* := \emptyset ; (* queue of states *)
4. **begin**
5. **for each** s **in** *IntitalStates*(m_0) **do**
6. *States* := *States* \cup $\{s\}$;
7. *insert*(*unexplored*, s)
8. **endfor**;
9. **while** *nonempty*(*unexplored*) **do**
10. *state* := *remove*(*unexplored*);
11. **for each** s **in** *NextStates*(*state*) **do**
12. **if** $s \notin$ *States* **then**
13. *States* := *States* \cup $\{s\}$;
14. *insert*(*unexplored*, s)
15. **endif**
16. **endfor**
17. **endwhile**
18. **end**.

where function *IntitalStates*(m) determines the set of initial states corresponding to the marking function m , and function *NextStates*(s) determines the set of successor states of s .

3. Distributed State Space Generation

In distributed generation of the state space, the partitioning function assigns each state to the region to which it belongs. The number of disjoint regions is usually equal to the number of available processors, k_p .

A straightforward partitioning function is based on the definition of the state, similarly to the one used in [12]. For timed nets, however, the partitioning function also takes into account the firing transitions:

$$region(s) = \left[\sum_{i=0}^{|P|} \alpha_i m(p_i) + \sum_{i=0}^{|T|} \beta_i n(t_i) \right] \bmod (k_p)$$

where $|P|$ is the cardinality of the set of places P , $|T|$ is the cardinality of the set of transitions T , the coefficients α_i and β_i are integer numbers, and m and n are marking and firing components of a state s [21].

Three kinds of (logical) processes are used in distributed generation of the state space [16], as shown in Fig.3.1: a process starting the distributed system and initiating the computations, called *Spawner*; several processes constructing the regions of the state space, called *Generators*, and a process collecting and integrating the results, called *Collector*.

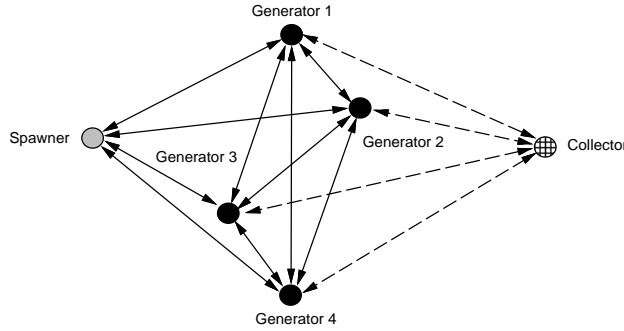


Fig.3.1. Distributed state space generation system.

The execution begins with the *Spawner* which creates the *Collector* and spawns k_p *Generators* on the hosts of the cluster; it also collects the identifiers of all created processes and broadcasts them to all processes so each process can send messages to any other one [16]. In its final phase, the *Spawner* sends the initial states to the appropriate *Generators*.

An outline of the *Spawner* process is as follows [16]:

1. Spawner:
2. **var** m_0 ; (* initial marking *)
3. k ; (* the number of hosts *)
4. $proc_table[]$; (* processor identifiers *)
5. **begin**
6. input virtual machine and model descriptions;
7. **spawn** *Collector* **on** *this_host*;
8. **for** $i := 1$ **to** k **do**
9. $proc_table[i] :=$ **spawn** *Generator* **on** *host*[i]
10. **endfor**;
11. $broadcast(proc_table)$;
12. **for each** s **in** $InitialStates(m_0)$ **do**
13. $send(proc_table[region(s)], s)$
14. **endfor**
15. **end**.

For each state belonging to region R_i , the *Generator_i* determines all successor states. A successor state can be in the same region (in which case it is called a *local* state) or in a different region (in which case it is called an *external* state). Each *Generator_i* sends all external states to the *Generators* determined by the partitioning function.

In order to perform state processing concurrently with communication, each *Generator* is composed of three processes: the *Analyzer*, responsible for processing the states, the

Sender, responsible for sending messages to other processes, and the *Receiver*, responsible for receiving messages from other processes and for the termination detection. When the *Spawner* creates the *Generators*, it actually creates *Analyzer* processes. As its first step, each *Analyzer* creates its *Receiver* and *Sender* processes [16].

The *Analyzer*, *Receiver*, and *Sender* processes of each *Generator* reside on the same processor. Their communication is based on shared variables, as shown in Fig.3.2.

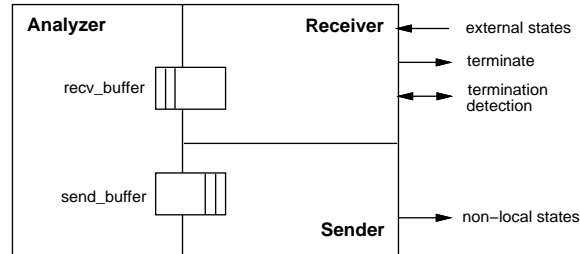


Fig.3.2. The structure of a *Generator*.

Each *Generator* processes the states from the internal queue *unexplored* (local states) and from *recv_buffer* (non-local states), with non-local states taking priority over local ones.

```

1. Analyzeri;
2. var Statesi := ∅;           (* set of states *)
3.   unexplored := ∅;         (* queue of states *)
4.   cont := true;          (* continuation flag *)
5. begin
6.   spawn Receiver, Sender on this_host;
7.   while cont do
8.     if empty(recv_buffer) ∧ nonempty(unexplored) then
9.       state := remove(unexplored);
10.      new := true
11.    else
12.      state := get(recv_buffer);
13.      if state = null then
14.        cont := false
15.      else
16.        new := state ∉ Statesi;
17.        if new then
18.          Statesi := Statesi ∪ {state}
19.        endif
20.      endif
21.    endif;
22.    if cont ∧ new then
23.      for each s in NextStates(state) do
24.        if region(s) = i then
25.          if not s ∉ Statesi then
26.            Statesi := Statesi ∪ {s};
27.            insert(unexplored, s)
28.          endif
29.        else
30.          put(send_buffer, s)
31.        endif
32.      endfor
33.    endif
34.  endwhile
35. end.

```

An important aspect of distributed applications is the termination condition which identifies the situation when no processor can continue its execution. The completion of all current tasks by any one of processors does not imply the overall termination, as some other processors may still continue and generate new states for processing. On the other hand, the processors cannot just wait for each other forever. A global termination detection algorithm [8] is interleaved with the computations, repeatedly checking if all processors have finished their tasks [16]. When global termination is detected (i.e., when all *Analyzer* processes are waiting on *get* operation – line 12), a special *null* state is sent to all *Analyzer* processes to terminate their operation (lines 13, 14).

Processes residing on different hosts constitute a “virtual machine”; they communicate by exchanging messages using the popular PVM (Parallel Virtual Machine) message passing library [9].

4. Petri Net Model

An outline of a cluster of (four) workstations connected to a switch is shown in Fig.4.1 (typically clusters contain more workstations, e.g., 16 or 32).

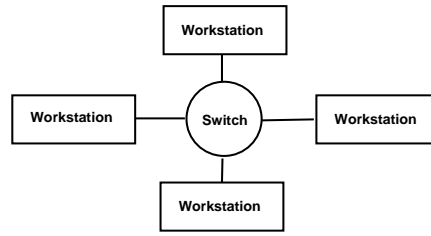


Fig.4.1. An outline of a cluster of 4 workstations.

A Petri net model of a single processor connected to a switch, with independent processes for state generation and for message passing to other processors, is shown in Fig.4.2.

The processing of states is represented by transition t_{ip} with the average time of processing a single state assigned to it as the firing time. The queue of states waiting for processing is represented by place p_{ir} , which, in Fig.3.2, contains 2 tokens (i.e., 2 states waiting for processing).

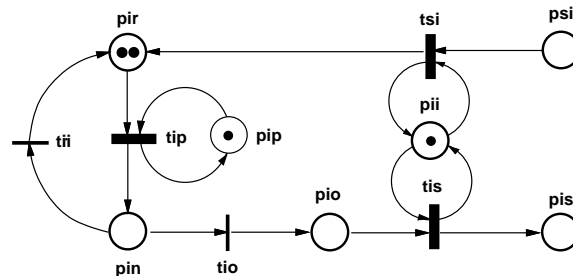


Fig.4.2. Petri net model of a processor and its link.

Transitions t_{is} and t_{si} represent message passing to and from the switch, respectively. Since the messages share the same link, place p_{ii} is a shared resource (the link) which can be used either for sending (t_{is}) or for receiving (t_{si}) a message.

For a 4-machine cluster (Fig.4.1), the switch connecting the processors is outlined in Fig.4.3; each of messages incoming from four directions (p_{1s} , p_{2s} , p_{3s} and p_{4s}) has a free-choice structure which forwards the message to one of the other processors connected to

of processors, and the switch is represented by a single transition t_{sw} with the number of occurrences equal to $k_p * (k_p - 1)$, grouped into k_p free-choice structures corresponding to the outline shown in Fig.4.3.

5. Performance Results

The performance of distributed applications depends upon several factors, which include the balancing of the workload among the processors, and also the amount of communication that is needed for the execution of distributed tasks.

The net model shown in Fig.4.4 contains two timing parameters, the firing time of t_{ip} , representing the (average) time required to process a single state, and the firing times of t_{is} and t_{si} , that represent the (average) time required for sending a single state description between a processor and the switch. Since the performance of the model depends upon the ratio of these two parameters rather than on their specific values, it is convenient to use just one parameter which characterizes model's temporal properties. The computation-to-communication ratio, $r_{comp/comm}$, is the ratio of the (average) state processing time to the (average) time needed for sending a single state description between two processors (which is assumed to be the same for all pairs of processors in a cluster).

In order to represent the behavior of the state generation process, some simplifying assumptions are made. It is assumed that a finite (but large) state space is generated in a “steady” manner, i.e., that the size of the *unexplored* queue in the scheme outlined at the end of Section 2, is approximately constant for the most part of the generation process. This assumption is not always valid, but is reasonable for many net models (in particular for models which contain buffers with large capacities, in which case the state space generation basically repeats similar groups of states for different “states” of the buffer(s)).

A practical consequence of the “steady” generation assumption is that each analyzed state generates one new state (i.e., a state can generate several next states, but, on average, only one of these states is new).

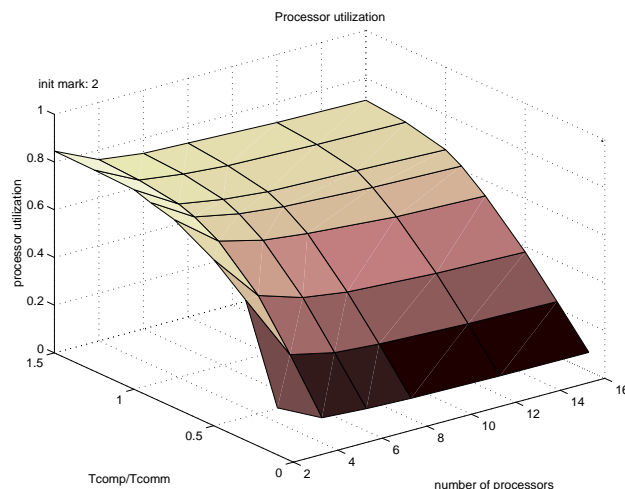


Fig.5.1. Utilization of processors for $m_0(p_{ir}) = 2$.

The workload of the system is controlled by the initial distribution of states among the processors (the initial marking of p_{ir} in Fig.4.2 and Fig.4.4). Two cases are analyzed: (i) when the same initial conditions are used for all processors, irrespective of their number (this is called “proportional load” as the total load is proportional to the number of processors), and (ii) the “shared load”, when the (total) initial marking is the same for different numbers

of processors, so as the number of processors increases, the load assigned to each processor decreases.

For proportional load, the utilization of processors, for different values of $r_{comp/comm}$ (from 0.1 to 1.5) and for several different numbers of processors (from 2 to 16), is shown in Fig.5.1 for small workload (the initial marking of p_{ir} equal to 2), and in Fig.5.2 for high workload (the initial marking of p_{ir} equal to 8).

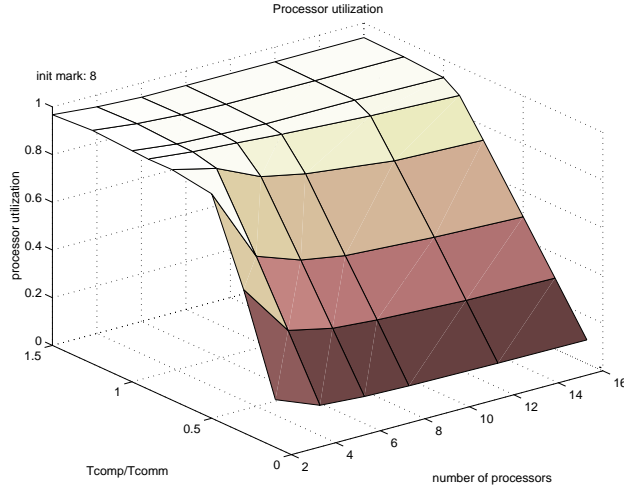


Fig.5.2. Utilization of processors for $m_0(p_{ir}) = 8$.

The results have similar character, and it can be observed that the utilization increases with the increase of the workload; Fig.5.2 shows some “saturation effects” for values of $r_{comp/comm}$ greater than 1. Moreover, for communication times significantly greater than the computation time (i.e., for $r_{comp/comm}$ less than 0.5), processor’s utilization is consistently poor, below 50%, and practically linearly tends to 0 with the value of $r_{comp/comm}$.

For the case of shared load, similar results are shown in Fig.5.3 for small workload (the initial marking of 8 is shared among all the processors), and in Fig.5.4 for high workload (the initial marking of 32 is shared among the processors, so the plots for $k_p = 16$ in Fig.5.4 and Fig.5.1 are the same; both correspond to the initial marking of p_{ir} equal to 2).

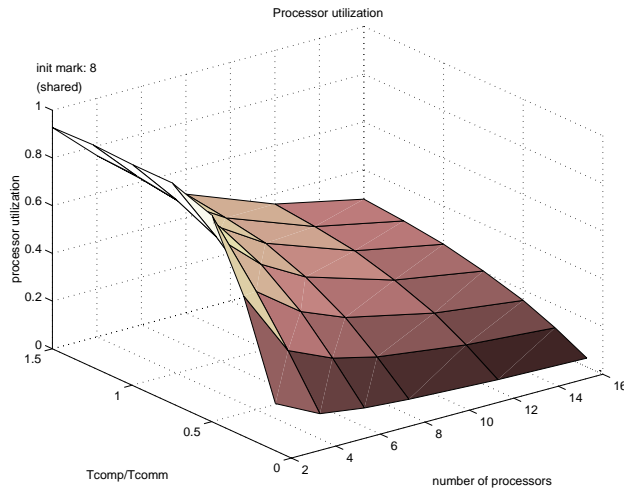


Fig.5.3. Utilization of processors, low shared load ($m_0 = 8$).

In the model shown in Fig.4.2, the (free-choice) probability associated with t_{ii} , $c(t_{ii})$,

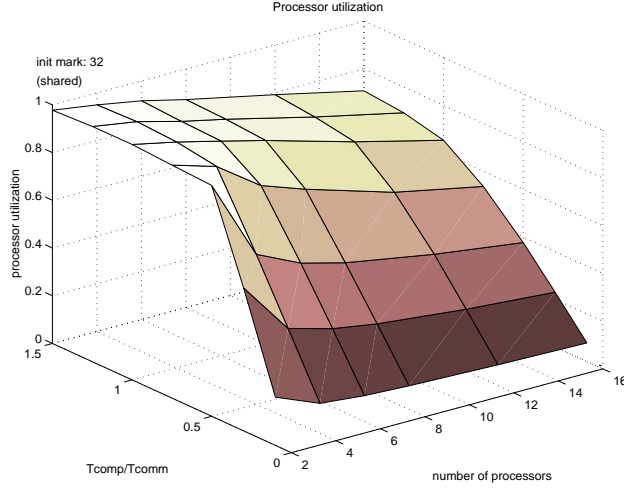


Fig.5.4. Utilization of processors, high shared load ($m_0 = 32$).

represents the probability that a new, generated state is a local one. Its default value, for uniform distribution of states over the nodes of the cluster of workstations, is $1/k_p$.

Fig.5.5 and Fig.5.6 correspond to Fig.5.1 and Fig.5.2 but assume that 50% of new states in each region are local states, while the remaining 50% are uniformly distributed over the other regions.

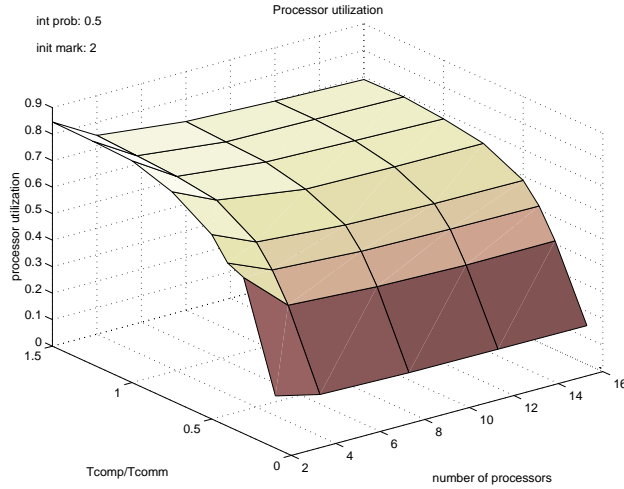


Fig.5.5. Utilization of processors for $m_0(p_{ir}) = 2$ and $c(t_{ii}) = 0.5$.

Increased values of $c(t_{ii})$ correspond to the clustering of states in the regions, i.e., the situations when many states generated by each processor are local states, so they do not require any communication, and this improves the utilization of processors. Increased utilization of processors results in increased speedup.

The speedup $S(k_p)$ of a k_p -processor system is usually defined as the ratio of execution time of an application on one processor, $T(1)$, to the application's execution time on k_p processors, $T(k_p)$ [20]:

$$S(k_p) = \frac{T(1)}{T(k_p)}.$$

For the ideal, uniform distribution of workload among the processors, the execution time on k_p processors can be expressed as:

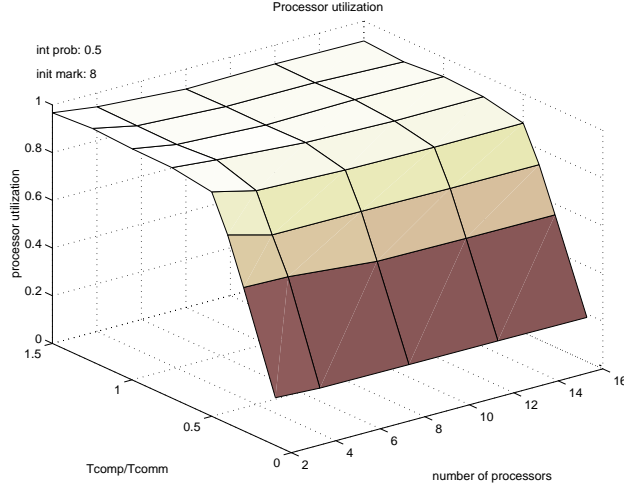


Fig.5.6. Utilization of processors for $m_0(p_{ir}) = 8$ and $c(t_{ii}) = 0.5$.

$$T(k_p) = \frac{T(1)}{k_p} \frac{1}{u_p(k_p)}$$

where $u_p(k_p)$ is the utilization of each processor in a k_p -processor system, and then the speedup is simply:

$$S(k_p) = k_p u_p(k_p).$$

Fig.5.7 and Fig.5.8 show the speedup as a function of the number of processors, k_p , for medium (proportional) workload ($m_0(p_{ir}) = 4$) and several values of $c(t_{ii})$, and for two different values of $r_{comp/comm}$; in Fig.5.7 $r_{comp/comm} = 1$, and in Fig.5.8 $r_{comp/comm} = 0.25$.

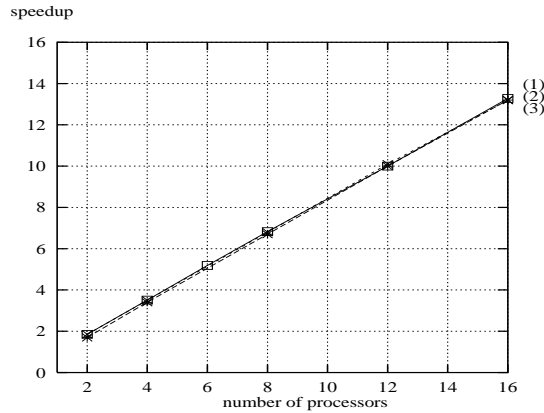


Fig.5.7. Speedup as a function of k_p ; $r_{comp/comm} = 1.0$,
 (1) $c(t_{ii}) = 1/k_p$, (2) $c(t_{ii}) = 0.5$, (3) $c(t_{ii}) = 0.75$.

It can be observed that, by increasing the value of $c(t_{ii})$, a significant improvement of the speedup can be obtained in cases when the communication is critical (i.e., $r_{comp/comm} < 0.5$), as shown in Fig.5.8; for $r_{comp/comm} \geq 1$ the effect of state clustering is quite negligible (Fig.5.7); in this case, the communication is not critical to the performance of the system. The value of $c(t_{ii})$, the probability that a new, generated state is local, depends primarily upon the partitioning function which assigns states to the regions of the distributed state

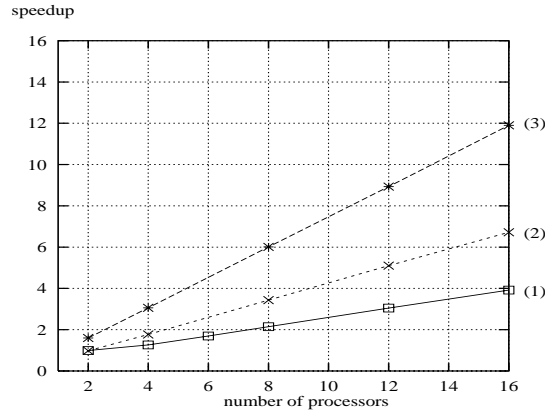


Fig.5.8. Speedup as a function of k_p ; $r_{comp/comm} = 0.25$,
 (1) $c(t_{ii}) = 1/k_p$, (2) $c(t_{ii}) = 0.5$, (3) $c(t_{ii}) = 0.75$.

space. Finding a partitioning function which takes the properties of the net model into account in a way that maximizes the value of $c(t_{ii})$, i.e., maximizes the clustering of states within regions, is an interesting area of research.

6. Simplified Performance Evaluation

In simulation-based performance evaluation, the simulation time depends more than linearly upon the size of the modeled cluster. A natural question is then if there is a need to model all the processors of the cluster to obtain the performance results.

For the shared workload, if the total token count of the initial marking is m_0 tokens, and the cluster is composed of k_p processors, the average load per processor is equal to m_0/k_p . The same average load per processor exists in a cluster with k'_p processors and with the initial marking containing $m_0 * k'_p/k_p$ tokens. Consequently, instead of simulating a 16-processor cluster with the total workload of 32 tokens, a 4-processor cluster can be simulated with the load of 8 tokens. Fig.6.1 and Fig.6.2 show the results obtained for a 4-processor cluster with the load adjusted to the case shown in Fig.5.3 and Fig.5.4.

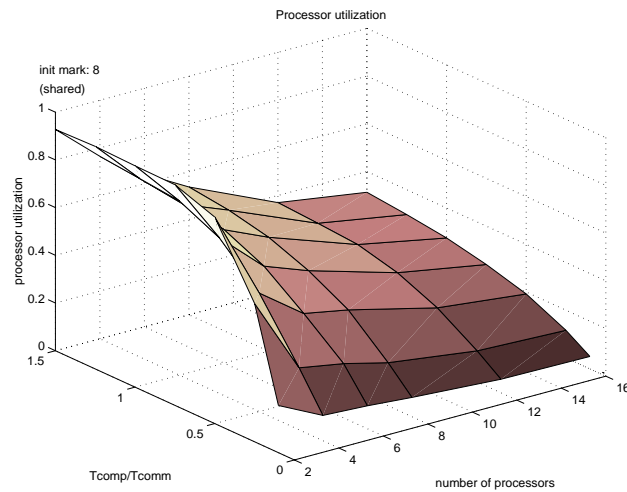


Fig.6.1. Utilization of processors, low shared load ($m_0 = 8$).

It can be observed that the results shown in Fig.6.1 and Fig.5.3 as well as Fig.6.2 and Fig.5.4 are practically the same.

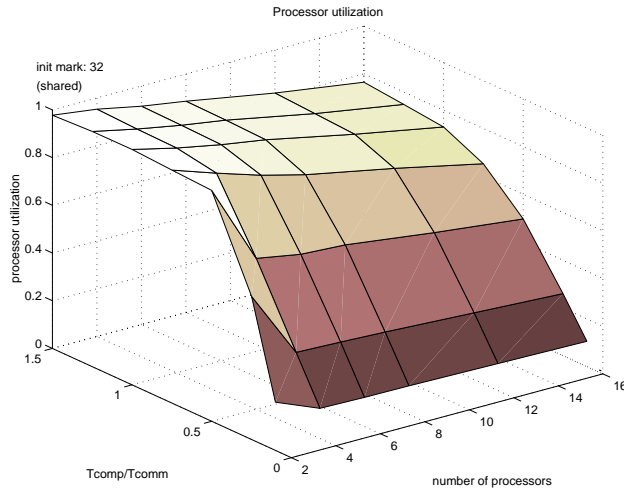


Fig.6.2. Utilization of processors, high shared load ($m_0 = 32$).

For proportional load, the performance is practically independent of the number of processors for k_p greater than 3, as can be observed in Fig.5.1 and Fig.5.2. This is due to the model of the switch which does not introduce any delays of forwarded messages (all delays are associated with the links).

7. Concluding Remarks

The paper shows that the performance of distributed state space generation can be improved, sometimes quite significantly, if the partitioning function assigns clusters of states to regions associated with independent processors. More specifically, if the communication is the bottleneck of a distributed application, i.e., if the computation to communication ratio is less than 1, any reduction of the amount of required communication improves the performance of the distributed application. However, designing a partitioning function with good locality properties is not an obvious task.

For example, the net model of the bounded buffer communication scheme, shown in Fig.7.1 (both producer and consumer are represented by simple free-choice structures), has a reachability graph with systematic structure, in which a section of the graph is repeated for each additional token in p_5 and/or p_6 (i.e., another unit of space in the buffer).

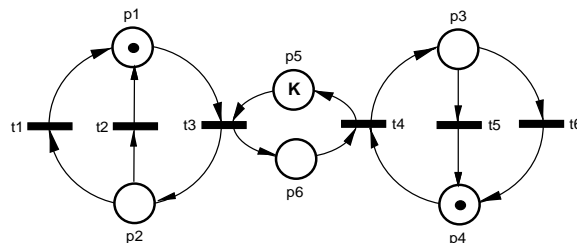


Fig.7.1. Net model of a bounded-buffer synchronization.

Fig.7.2 shows the initial part of the reachability graph for the model shown in Fig.7.1, with two different partitioning functions: (a) groups the nodes into regions with only a very few connections between the regions while (b) shows regions in which only a few nodes are connected within the same region, so there are very few local nodes. Case (a) has

significantly less communication than case (b), and the partitioning function in this case should take into account only the marking of place p_5 (or place p_6).

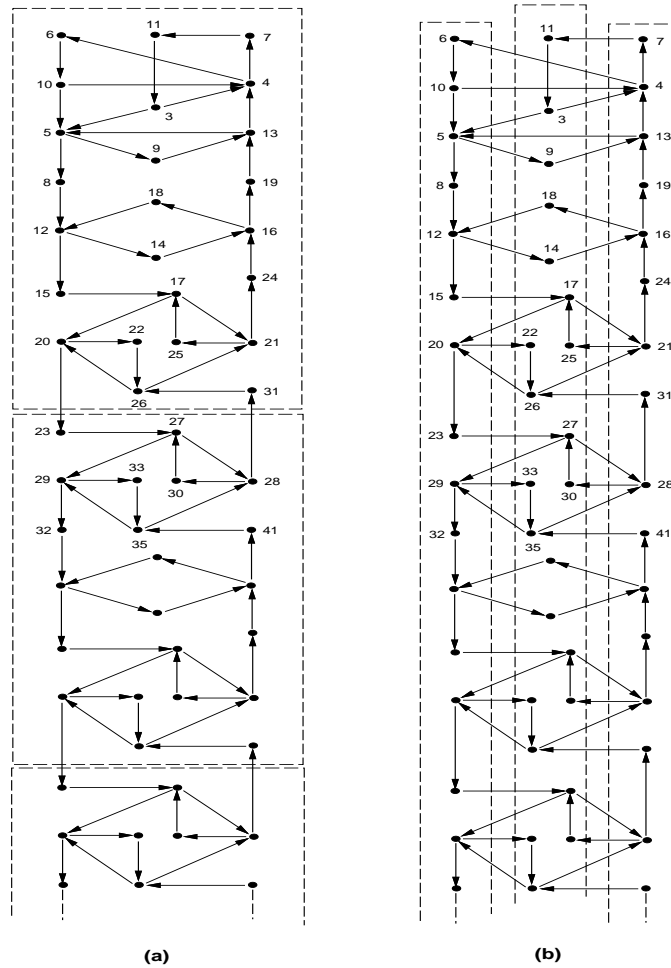


Fig.7.2. Part of the reachability graph for the net in Fig.7.1 with two different partitioning functions.

The very simple model of distributed state space generation, described in this paper, assumes that the temporal behavior can be described by constant times, i.e., that the analysis of each state requires the same amount of time, and that sending the state description from one processor to another also requires the same amount of time for each state. The first assumption (that the state processing time is constant) is not very realistic because the amount of (processing) time required for the generation of all “next states” depends upon the topology of the net (i.e., whether the current state is a conflict-free, free-choice or more general conflict state). However, it appears that the effects of variable state processing times are not very significant. Fig.7.3 shows the utilization of the processors for the case of shared load when the state processing time is exponentially distributed with the average value that is equal to the (constant) value used in Fig.5.3. It can be observed that the utilization shown in Fig.5.3 is slightly better than in Fig.7.3, but the differences are not significant.

Similarly, Fig.7.4 shows the utilization of processors for high shared load with random state processing times (also exponentially distributed), and the average value equal to that used for Fig.5.4. Again, the results shown in Fig.7.4 and Fig.5.4 are quite similar.

The assumption that the time required for sending a state description from one processor

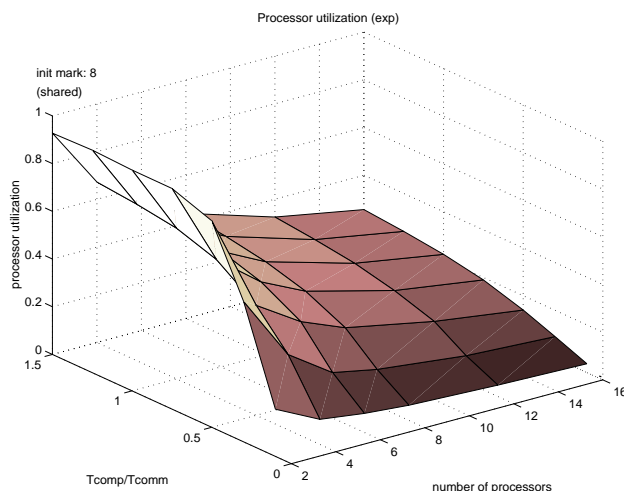


Fig.7.3. Utilization of processors, low shared load ($m_0 = 8$).

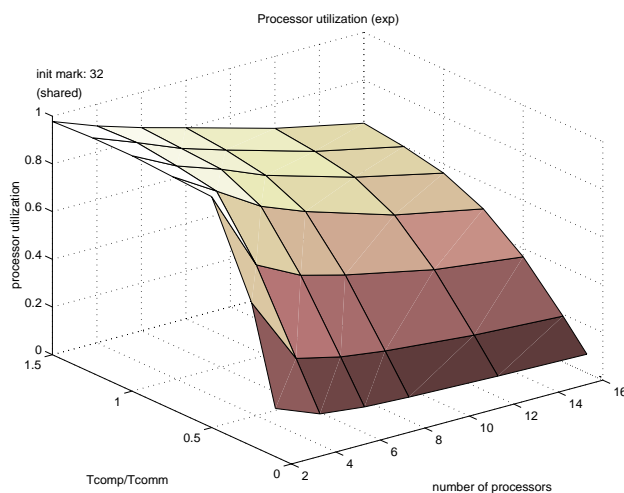


Fig.7.4. Utilization of processors, high shared load ($m_0 = 32$).

to another is the same for all states is based on typical characteristics of the message sending time as a function of message length which, for typical message-passing libraries (such as PVM or MPI) practically does not depend upon the message length for short (i.e., in the range of Kbytes) messages [18]; a representation of a state with 100 marked places and 100 firing transitions (which would typically correspond to a model with thousands of places and thousands of transitions) requires, without any “compaction”, about 1.6 Kbytes assuming the “standard” 4-byte representation of integers.

In order to capture the steady-state behavior of distributed generation of the state space, it was assumed that each analyzed state generates one new state. For many net models the state space is not generated in such a steady way, and the number of “unexplored” states changes in a very broad range during the generation. It is expected that the effects of such variations can be represented by an extension of the presented mode, and then the (average) performance can be studied in a similar way.

The presented approach is not restricted to the discussed application; it can be used to represent the behavior of many other applications which use reachability analysis, such as model checking [7] or automated verification of discrete-event systems [13].

All simulations of net models were performed using the TPN-tools package [22], [23].

The simulation runs corresponded to processing at least 100,000 states (for the shared load cases). Although the confidence intervals were not determined for these experiments, the results are presented without any “smoothing” operations, which indicates rather small variances of the obtained results.

Acknowledgements

The Natural Sciences and Engineering Research Council of Canada partially supported this research through grant RGPIN-8222.

Several constructive and insightful comments and remarks of four anonymous reviewers, an especially reviewer A, are gratefully acknowledged.

References

- [1] M. Ajmone Marsan, G. Balbo, and G. Conte, 1984. “A class of generalized stochastic Petri nets for the performance evaluation of systems”; *ACM Transactions on Computer Systems*, vol.2, no.2, pp.93–122.
- [2] S.C. Allmaier and G. Horton, 1997. “Parallel shared-memory state-space exploration in stochastic modeling”; *Solving Irregularly structured Problems in Parallel (IRREGULAR’97), Lecture Notes in Computer Science*, vol.1253, pp.207–218, Springer-Verlag.
- [3] S.C. Allmaier, S. Dalibor, and D. Kreische, 1997. “Parallel graph generation algorithms for shared and distributed memory machines”; *Parallel Computing: Fundamentals, Applications and New Directions (Proc. of the Conference ParCo’97), Advances in Parallel Computing*, vol.12, pp.581–588, Elsevier, North-Holland.
- [4] S.C. Allmaier, M. Kowarschik, and G. Horton, 1997. “State space construction and steady state solution of GPSN on a shared memory multiprocessor”; *Proc. IEEE Int. Workshop Petri Nets and Performance Models (PNPM ’97)*, pp.112–121.
- [5] S.C. Allmaier and D. Kreische, 1999. “Parallel approaches to the numerical transient analysis of stochastic reward nets”; in *Application and Theory of Petri Nets 1999 (Proc. 20th International Conference, IACTPN’99)*, (Lecture Notes in Computer Science 1639), pp.147–167, Springer-Verlag.
- [6] F. Bause and P. Krinzing, 1996. *Stochastic Petri Nets - An Introduction to the Theory*. Vieweg.
- [7] E.M. Clarke, O. Grumberg, D. Peled, 1999. *Model checking*. MIT Press.
- [8] E. Dijkstra, W. Feijen, and A. van Gasteren, 1983. “Derivation of a termination detection algorithm for distributed computations”; *Information Processing Letters*, vol.16, no.5, pp.217–219.
- [9] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, 1994. *PVM: Parallel Virtual Machine. A Users’ Guide and Tutorial*. MIT Press.
- [10] M.A. Holliday, M.K. Vernon, “Exact performance estimates for multiprocessor memory and bus interference”; *IEEE Trans. on Computers*, vol.36, no.1, pp.76–85, 1987.
- [11] K. Jensen, 1987. “Coloured Petri nets”; in *Advanced Course on Petri Nets 1986 (Lecture Notes in Computer Science 254)*, pp.248–299, Springer-Verlag.
- [12] P. Marenzoni, S. Caselli, and G. Conte, 1997. “Analysis of large GSPN models: a distributed solution tool”; *Proc. IEEE Int. Workshop on Petri Nets and Performance Models (PNPM’97)*, pp.122–131.

- [13] K.L. McMillan, 2000. "A methodology for hardware verification using compositional model checking", *Science of Computer Programming*, vol.37, no.1-3, pp.279-309.
- [14] T. Murata, 1989. "Petri nets: properties, analysis, and applications"; *Proceedings of the IEEE*, vol.77, no.4, pp.541-580.
- [15] J. Peterson, 1981. *Petri Net Theory and the Modeling of Systems*. Prentice Hall.
- [16] I. Rada, 2000. "Distributed generation of state space for timed Petri nets"; M.Sc. Thesis, Department of Computer Science, Memorial University of Newfoundland, St.John's, Canada. o
- [17] I. Rada, W.M. Zuberek, "Distributed generation of state space for timed Petri nets"; Proc. High Performance Computing Symposium 2001, Seattle, WA, pp.219-227, 2001.
- [18] M.R. Steed, M.J. Clement, 1996. "Performance prediction of PVM programs"; Proc. 10-th Int. Parallel Processing Symposium (IPPS-96), pp.803-807.
- [19] W. Stewart, 1994. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press.
- [20] B. Wilkinson, 1996. *Computer Architecture - Design and Performance* (2-nd ed.). Prentice Hall.
- [21] W.M. Zuberek, 1991. "Timed Petri nets, definitions, properties, and applications"; *Microelectronics and Reliability*, vol.31, no.4, pp.627-644.
- [22] W.M. Zuberek, 1996. "Modeling using timed Petri nets - model description and representation"; Technical Report #9601, Department of Computer Science, Memorial University, St.John's, Canada A1B 3X5.
- [23] W.M. Zuberek, 1996. "Modeling using timed Petri nets - event-driven simulation"; Technical Report #9602, Department of Computer Science, Memorial University, St.John's, Canada A1B 3X5,