Parallel and Distributed Simulation of Free Choice Petri Nets

François Baccelli, Nathalie Furmento and Bruno Gaujal
INRIA Sophia-Antipolis,
BP 93, 06902 Sophia-Antipolis, France
e-mail : {baccelli}{nfurment}{gaujal}@sophia.inria.fr

Abstract

In this paper, we present two new methods to simulate Petri Nets : a data parallel simulation and a distributed simulation. Both simulations use an equational representation of the net in the so called (min, +) algebra. The data parallel simulation is based on the use of matrix representation of these equations, and the distributed simulation on the decomposition of a Petri Net into marked graph components.

1 Introduction

Petri Nets provide a powerful formalism to model various classes of discrete event systems. They can be used for qualitative and simulation purposes because they provide a better understanding of the dynamics of the system.

Parallel simulation of large discrete event networks has been introduced in the 80's. We can cite the seminal work of Chandy and Misra [4] and the optimistic method introduced by Jefferson [10]. More recently, new techniques, like time parallelism have emerged and provide substantial improvements [9, 7]. As for Petri Nets, one can exploit the net structure to improve the distribution of the simulation, as in [5]. However, all these techniques are often *ad hoc* and use an event-driven approach of the simulation.

A novel technique to simulate large Petri Nets have been presented in [1]. The simulation was no longer based on the events occuring in the system but on evolution equations describing the dynamics of the system and written in the (max,+) algebra. This method provided substantial acceleration of the simulation. However, it was restricted to a small class of systems: the marked graphs. Here, we present a similar approach that provides evolution equations using the (min,+) formalism and which works for Free Choice Nets, a more general class of nets with a considerably greater power of description than marked graphs.

This paper is organized as follows. Section 2 presents the class of Petri Nets that we will simulate. After that, we present in Sections 3 and 4 respectively, two algorithms of simulation. The first one is a data parallel algorithm that uses exclusively the evolution equation under its matrix multiplication form. The second uses the equations as well as a structural decomposition of the network. Section 5 presents possible improvements for these two programs, and finally Section 6 presents performance measures made respectively on the CM2 and on a farm of workstation communicating via PVM.

2 Definition of Petri Net

A Petri Net is a t-uple $S = (P, T, A, M_0)$ where P is the set of places, T is the set of transitions, A is the set of arcs between places and transitions and between transitions and places (A is a subset of $P \times T \cup T \times P$), and $M_0 : P \to \mathbb{N}$ is the initial marking in the places[1].

We define the following sets :

- $^\bullet t = \{p \in P : (p,t) \in A\}$ is the set of all the input places of t,
- $t^\bullet = \{p \in P : (t,p) \in A\}$ is the set of all the output places of t,
- $^\bullet p = \{t \in T : (t,p) \in A\}$ is the set of all the input transitions of p,
- $p^\bullet = \{t \in T : (p,t) \in A\}$ is the set of all the output transitions of p.

Definition 1 (serial place, routing place) *A serial place is a place p such that $|^\bullet p| = |p^\bullet| = 1$, and for a routing place p, $|^\bullet p| + |p^\bullet| > 2$.*

2.1 Timed Petri Nets

A timed Petri Net is a Petri Net with timings associated to transition firings. For each transition t, we define a firing time process $\{\phi_t(n), n \in \mathbb{N}\}$ which gives the duration of the n-th firing of transition t. If one token becomes available at time k in each input place of t, then t initiates its n-th firing. If the tokens are still available at time $k+\phi_t(n)$, the tokens are removed from the input places and one token is placed in each

[1] For more details on Petri Nets, see [12], for example.

output place of t. This completes the n-th firing of transition t.

In the following, we will restrict the firing times to be positive numbers all multiple of a common time step, which can be taken equal to 1 without loss of generality. Furthermore, we assume that they are all bounded by K: $\forall t \in T, \forall n \in \mathbb{N}, \phi_t(n) \in \{1, \cdots, K\}$.

2.2 Free Choice Nets

Free Choice Nets are Petri Nets verifying the following condition :

$$\forall p \in P, |p^\bullet| > 1 \Rightarrow {}^\bullet(p^\bullet) = \{p\},$$

Free Choice Nets have been extensively studied in the 70's and have regained interest recently because they constitute a nice compromise between power of modelization and tractability of problems.

Routing in Free Choice Nets There exists several policies to solve the problem of conflicts when two transitions share a common input place. In free choice nets, most of them can be implemented using a *routing policy.*

In the routing policy, each routing place p is given a routing sequence $\rho^p : \mathbb{N} \to p^\bullet$, where $\rho^p(n)$ is the transition $t \in p^\bullet$ which receives the n-th token to enter place p. The routing sequences can be periodic or random. If a token is the k-th to be routed to transition $t \in p^\bullet$, then this token is immediately consumed by the transition where it experiences a firing time of $\phi_t(k)$. It can be shown that Free Choice Nets with routing policies have the power of description of Turing machines.

Definition 2 (routing function) *From the routing sequence ρ^p defined for each routing place p, we define for every transition $t \in p^\bullet$, the function :*

$$\Pi^t(n) = \sum_{i=1}^{n} 1_{(\rho^{\bullet t}(i)=t)} \, .$$

$\Pi^t(n)$ gives the number of tokens that are routed to transition t after n tokens have entered place ${}^\bullet t$.

2.3 Evolution Equations for Counters

The choice of state variables used to describe the evolution of a Petri Net is essential to exhibit its underlying dynamical structure. Unlike in the classical framework that uses the markings as state variables, we rather use the *counters* of the system. For every transition, we define $X_t(k)$ to be the number of times that the transition t has begun to fire by time k. The counters describe completely the state of the system at time k, in particular, the marking at time k can be easily retrieved from the counters. If we denote by $M_p(k)$ the marking of place p at time k,

$$M_p(k) = \sum_{t \in {}^\bullet p} X_t(k) - \sum_{t \in p^\bullet} X_t(k).$$

Now, we construct *the evolution equations for counters* that will be used to simulate the system.

The transitions can be separated into two subsets :

- the set of transitions which belong to the output set of serial places,
- the set of transitions which belong to the output set of a routing place.

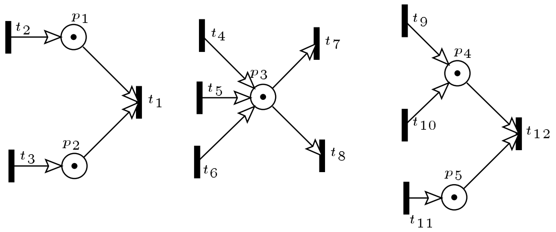

Figure 1: Transitions in a Petri Net.

For the first set of transitions, (like transition t_1 in Figure 1) the min operator describes the evolution of their counters. Indeed, the number of firing of these transitions is equal to the minimum of the number of firings of its upstream transitions. For example, in Figure 1, for the transition t_1, we get :

$$X_1(k) = \min((X_2(k - \phi_2) + 1), (X_3(k - \phi_3) + 1)).$$

For the second set of transitions, (t_7 and t_8 in Figure 1) the usual addition is used, and we also have to use a routing function (see definition 2). To compute the number of firings, we must add the number of completed firings of upstream transitions and the initial marking of the input place p, denoted by $N_p(k)$, then apply the routing function. For example, in Figure 1, we have :

$$\begin{aligned} N_3(k) &= X_4(k - \phi_4) + X_5(k - \phi_5) + X_6(k - \phi_6) + 1 \\ X_7(k) &= \Pi^7(N_3(k)) \\ X_8(k) &= \Pi^8(N_3(k)). \end{aligned}$$

However, we can also find a third kind of transitions : the transitions which have in their input set several places which are not all serial (t_{12} in Figure 1). For these transitions, the usual algebra and the min operator would be necessary to calculate their evolution. To avoid this situation, we apply a canonical transformation to the Free Choice Net.

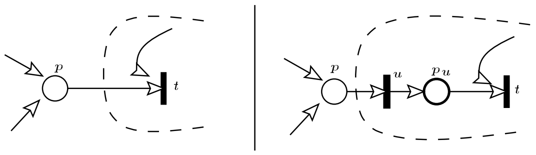

Figure 2: Canonical transformation.

2.4 Canonical Form of a Petri Net

Let $F = (P, T, A, M_0)$ a general Petri Net. Each transition $t \in T$ verifies the following conditions :

1. ${}^\bullet t \cap t^\bullet = \{p\}$ (*i.e.* each transition is recycled),
2. $\forall n \in \mathbb{N}, \phi_t(n) > 0$.

To obtain its canonical form $F_c = (P_c, T_c, A_c, M_{0_c})$, we make the following transformation. For every routing place p of F, we add a transition u and a empty place p_u between p and each output transition of p, such that $p^\bullet = \{u\}$, $u^\bullet = \{p_u\}$ and $p_u^\bullet = \{p\}$ (see Figure 2).

In the new Petri Net, we only have two sets of transitions, $T = T_1 \cup T_2$:

- The added transitions $u \in T_1$ which verify ${}^\bullet u$ is a routing place,
- and the original transitions $t \in T_2$ which verify ${}^\bullet t$ contains only serial places.

Lemma 1 *For a transition $i \in T_1$, we have :*

$$X_i(k) = \Pi^i \left(N_{\bullet i}(k) \right), \tag{1}$$

where :

$$N_{\bullet i}(k) = \sum_{j \in {}^{\bullet\bullet} i} \left(X_j(k - \phi_j) + M_0(j,i) \right).$$

And for a transition $i \in T_2$:

$$X_i(k) = \min_{j \in {}^{\bullet\bullet} i} \left(X_j(k - \phi_j) + M_0(j,i) \right). \tag{2}$$

3 Data Parallel Simulation

We recall that $X_i(k)$ is the number of times that transition i has begun to fire at time k. We build the vector $X(k) = \{X_i(k), i \in T\}$. This vector is decomposed into two other vectors : $S(k) = \{X_i(k), i \in T_2\}$ and $Z(k) = \{X_i(k), i \in T_1\}$.

Assumption 1 *In the following, we suppose that firing times are constant i.e. $\phi_i(n) = \phi_i, \forall n \in \mathbb{N}$.*

This introduces no loss of generality since a system with bounded stochastic firing times with integer values can be transformed into a system with constant firing times by replacing every timed transition with a random firing time by a subnetwork as in Figure 3.

Indeed, let $\phi(n)$ be an integer-valued random process bounded by K. Then, the law of $\phi(n)$ is determined by the sequences $P_1(n), \cdots, P_K(n)$, with $P_i(n) = \mathbf{P}(\phi(n) = i)$.

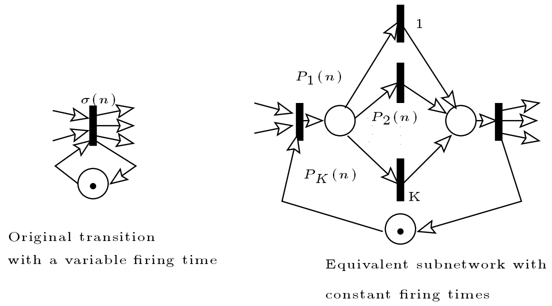

Figure 3: Transformation of a random firing time transition into a subnetwork composed with constant firing time transitions.

Definition 3 (algebra (min, +)) *The algebra (min, +) is the semi-field $\mathbb{R}_{\min} = (\mathbb{R}^\star, \wedge, \odot)$ where $\mathbb{R}^\star = \mathbb{R} \cup \{+\infty\}$, the law $\wedge$ is the minimization, and the law $\odot$ the addition.*

In this new algebra, the product matrix-vector is defined of the following way :

$$\forall A \in \mathbb{R}_{\min}^{n \times m}, \forall v \in \mathbb{R}_{\min}^{m} \quad [A \odot v]_i = \bigwedge_j (A_{ij} \odot v_j).$$

The use of the (min, +) algebra gives a vectorial version of the evolution equations.

Lemma 2 *Under the above assumptions,*

$$S(k) = \bigwedge_{l=0}^{K} (A_l \odot S(k-l) \wedge B_l \odot Z(k-l)),$$

$$Z(k) = \Pi \left(\sum_{l=0}^{K} (P_l Z(k-l) + Q_l S(k-l)) + R \right)$$

where :

- *$A_l = [A_l(t,t')]_{T_2 \times T_2}$ is defined by $A_l(t,t') = m$ if $\phi_{t'} = l$ and $M_0(t',t) = m$. If there is no place between transitions t and t', $A_l(t,t') = \infty$. If there is more than one place between t and t', we take $A_l(t,t')$ equal to the minimum of their initial marking.*

- $B_l = [B_l(t,t')]_{T_2 \times T_1}$ *is defined by* $B_l(t,t') = m$ *if* $\phi_{t'} = l$ *and* $M_0(t',t) = m$. *If there is no place between transitions* t *and* t', $B_l(t,t') = \infty$.

- $P_l = [P_l(t,t')]_{T_1 \times T_1}$ *is defined by* $P_l(t,t') = 1$ *if* $\phi_{t'} = l$ *and there is one place between* t *and* t'. *Otherwise* $P_l(t,t') = 0$.

- $Q_l = [Q_l(t,t')]_{T_1 \times T_2}$ *is defined by* $Q_l(t,t') = 1$ *if* $\phi_{t'} = l$ *and there is one place between* t *and* t'. *Otherwise* $Q_l(t,t') = 0$.

- $R = [R(i)]_{T_1}$ *is the matrix of initial marking of places* ${}^\bullet t$ *(*$R(t) = M_0({}^\bullet t)$*).*

Proof. These equations are simply a rewriting of Equations (1) and (2). ■

This vectorial form of the evolution equations carries with it the power of the linear formalism and has already been useful in the past [2]. Here it shows that the dynamics of the Free Choice Nets is split into a (min, +) linear part and a nearly linear part in a classical sense. This nice property will be used to design a data parallel algorithm in this paper. This kind of equation has already been used to derive qualitative and theoretical results for Free Choice Nets [3].

3.1 Rewriting of the Equations

In order to obtain an equation which is easy to use in a data parallel program, we extend the function Π and the vector R to transitions in T_2 in the following way :

$$\Pi^t(n) = \begin{cases} \sum_{i=1}^{n} 1_{\rho^{\bullet t}}(i) & \text{if } t \in T_1, \\ n & \text{otherwise.} \end{cases}$$

$$R(t) = \begin{cases} M_0({}^\bullet t) & \text{if } t \in T_1, \\ 0 & \text{otherwise.} \end{cases}$$

Then, we can write :

$$X(k) = \Pi\left[\left(\bigoplus_{l=0}^{K} \begin{pmatrix} A(l) & B(l) \\ Q(l) & P(l) \end{pmatrix} \star X(k-l)\right) + R\right].$$

Now, we extend the state space to transform a recurrence of order K into a recurrence of order 1. We set :

$$\tilde{X}(k) = \begin{pmatrix} X(k) \\ X(k-1) \\ \vdots \\ X(k-K+1) \end{pmatrix},$$

$$M(i) = \begin{pmatrix} A(i) & B(i) \\ Q(i) & P(i) \end{pmatrix},$$

$$M = \begin{pmatrix} M(1) & M(2) & \cdots & M(K) \\ I & 0 & 0 & 0 \\ \vdots & I & \vdots & \vdots \\ 0 & 0 & I & 0 \end{pmatrix}.$$

Finally, we obtain :

$$\tilde{X}(k) = \Pi\left(M \star \tilde{X}(k-1) + R\right) \tag{3}$$

In these equations the operator $\oplus$ is at the same time the addition of the usual algebra (for transitions $\in T_1$) and the minimization of the algebra (min, +) (for transitions $\in T_2$). Similarly, the operator $\star$ is at the same time the multiplication of the usual algebra (for transitions $\in T_1$) and the addition of the algebra (min, +) (for transitions $\in T_2$).

3.2 A Data Parallel Simulation Method

Equation (3) can be efficiently parallelized using a data parallel algorithm similar to a classical parallel matrix-product. Here, we just use the "exotic" operators ($\oplus$, $\star$) instead of classical operators.

This approach has the advantage to use efficient method developed for linear algebra. In particular, we do not need to address the difficult problem of mapping onto the processors.

According to Equation (3), we see that the computation of $X_t(k), \forall t \in T$ is carried through the computation of vector $\tilde{X}(k)$. The steps of the computation are the following :

1. Computation of $X(k)$:
   (a) product matrix-vector,
   (b) addition of R,
   (c) computation of Π.

   We get

   $$X(k) = \Pi\left((M(1)\cdots M(K)) \star \tilde{X}(k-1) + R\right)$$

2. Computation of $\tilde{X}(k)$ from $X(k)$ and $\tilde{X}(k-1)$:
   (a) shift down $\tilde{X}(k)$ of $|T|$,
   (b) append $X(k)$.

The first step isn't completely parallel because of the simultaneous use of two algebras : we must first activate the processors which correspond to transitions in T_1 with the operator $\otimes$, and next activate the other processors with the operator $+$. After that, we can add the vector R, and then calculate the routing function. The second step is divided in two sub-steps which are completely parallel: shift the vector $\tilde{X}(k)$ and append the new vector $X(k)$.

Complexity Since our algorithm resemble a classical matrix vector multiplication, we can compute its PRAM complexity.

The time of execution for steps (1b), (1c), (2a) and (2b) is O(1) with $|T|^2K$ processors. So we can say that the time complexity of the computation of $X(k)$ is equal to the time of execution of a product matrix-vector. Since the size of the matrix is $|T| \times (K \times |T|)$, the complexity with $|T|^2K$ processors is $O(\log(|T|K))$.

The PRAM complexity for computation of $X(k)$ is $\mathrm{CREW}(|T|^2K,\ k\log(|T|K))$.

Implementation on the Connection Machine In the Connection Machine, there are many small processors, each with an associated memory, under the direction of a serial computer called the *front end*[2].

- *The number of processors* used is $p = \{2^n/2^n \geq |T|^2K \geq 2^{n-1}\}$. If p is greater than the number P of physical processors of the Connection Machine, each physical processor emulates sequentially $\lceil p/P \rceil$ virtual processors.
- Since the architecture of the Connection Machine is a hypercube, *the shift down of vector* $\tilde{X}(k)$ has a complexity of $O(\log(K))$, and not O(1).

4 Distributed Simulation

For this part, we use the scalar form of equations instead of their vectorial form. We assume that assumption 1 still holds and we use Equations (1) and (2).

4.1 Decomposition Into Components

The decomposition is based on the Marked Graph Components which are introduced in [6].

First, we define a relation $\mathcal{L}$ by: $t, t' \in T$, $t\mathcal{L}t'$ if there is a serial place p verifying $\{{}^\bullet p, p^\bullet\} = \{t, t'\}$. Let $\mathcal{K}$ be the transitive closure of $\mathcal{L}$. We partition the set of transitions T into its maximal $\mathcal{K}$-classes, $T_1, \cdots, T_n$. We denote $P_i = \{p \in P \mid p \text{ serial and } {}^\bullet p, p^\bullet \in T_i\}$, for all $1 \leq i \leq n$.

Definition 4 (Marked Graph Component) *A Marked Graph Component (MGC) G_i of a Petri Net $F = (P, T, A)$ is the sub-Petri Net $G_i = (P_i, T_i, A_i)$ with $A_i = A \cap (P_i \times T_i \cup T_i \times P_i)$.*

[2]see [11] for more details.

The places which do not belong to any component G_i are the routing places of the Petri Net. Figure 4 shows a example of decomposition.

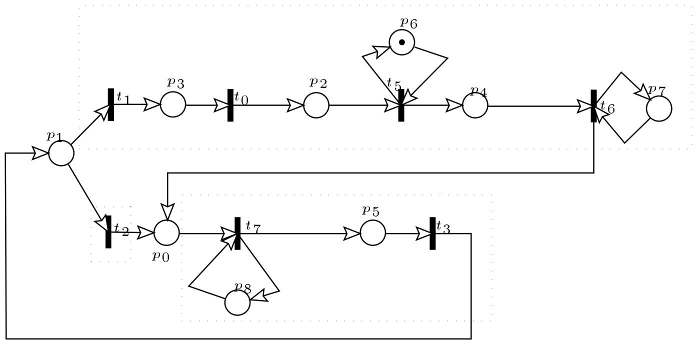

Figure 4: Example of decomposition into MGC.

4.2 Principles of The Algorithm

For the data parallel version, the program reads the structure of a Petri Net, and executes a simulation from this data. But here, we have another strategy: we have a program which reads the structure of a Petri Net, and which generates a distributed program of simulation (using PVM). So the simulation is divided in four steps:

1. Read the structure of the Petri Net.
2. Decompose the Petri Net into components: a component is either a MGC or a routing place with its output transitions.
3. Generate the code for every component (i.e. implement the equations (1) or (2)).
4. Execute the obtained program of simulation.

Generation of code The strategy is the following: a component receives data from its input components, executes calculation for its transitions, and sends the results to its output components. In order to minimize the number of messages, a transition t sends his results by packets of size ϕ_t.

For a MGC, the generated code for the computation of X_t at time k is of the following form:

1. If t is an input transition and the necessary data are not available, then receive data from all $t' \in {}^{\bullet\bullet}t$ (which are in its input components).
2. Compute $X_t(k)$.
3. If t is an output transition and (k mod $\phi_t = 0$), then send the ϕ_t-th last results to all $p \in t^\bullet$ (which are in output components).

For a process attached to a routing place p, the code at time k is of the following form :

1. If the data necessary are not available, then get the data from the inputs components.
2. Compute $N_p(k)$
3. For each $t \in p^{\bullet}$:
   (a) Compute $\Pi^t(N_p(k))$.
   (b) If (k mod $\phi_t = 0$), then send the ϕ_t-th last results to $t \in p^{\bullet}$.

Implementation PVM is a software package that allows a heterogeneous network of parallel and serial computers to appear as a single concurrent computational resource, which is called a *virtual machine*. The main routines permit to initiate processes or *tasks* on other machines (also called *hosts*), to communicate between processes and to change the configuration of machines (see [8] for more details).

For our program of simulation, we have one *task* per component in the above decomposition. The communications between tasks are always done with blocking messages.

5 Some Improvements

We provide several generalizations and improvement of the two algorithms.

5.1 Data Parallel Simulation

Block Matrix Algorithm The Data parallel algorithm that we presented need to be implemented on a massive parallel machine to be efficient. However if we want to implement it on a moderate size machine with high speed communications, we can modify the matrix product parallelization and design instead a block matrix multiplication, where one process is responsible for a block product and then we retrieve the global product by sums (or mins) on the rows. If the number of processors is lower than on the Connection Machine, but if, on the other hand, communications and computations are very fast, this method could be very efficient.

Sparse Matrix Method The matrix M used for the data parallel simulation (see Section 3) does not contain many significant values. Indeed, for a transition t, the number of significant values is $d_t = |^{\bullet\bullet}t|$. Let D be the maximum of d_t, $\forall t \in T$. Using sparse matrix reduction, we can use a new matrix M of size $|T| \star D$. If we de note by α, the maximum between D and K, the PRAM complexity of this new algorithm is CREW($|T|\alpha$, $k\log(D)$).

5.2 Distributed Simulation

Reduction of MGC Each MGC G with constant timings can be transformed into an equivalent MGC of minimal size G', in the sense that for any input sequences, the output sequences of G and of G' are the same. This transformation is presented in [2] and uses the (min,+) representation of the MGC. It can be made automatically.

This transformation substantially reduces the size of the system that a given task would have to simulate, thus accelerating the whole simulation process. However, this transformation can be done only if we are interested in statistics of the systems which are independent of the inner part of the MGC's.

Stochastic Timing We now consider the case where firing times are still integer-valued and bounded, but variable with time. For the distributed algorithm, we want to avoid the transformation presented in Figure 3 because this would increase the number of components and the performances would degenerate.

Let $\zeta_t(k)$ be the minimum of K and the time which has elapsed since the last time t has started firing before time k. If we consider the variables to be left-continuous, we have :

Lemma 3

$$S(k) = \bigwedge_{l=1}^{K} (A_l(k) \odot S(k-l) \wedge B_l(k) \odot Z(k-l)),$$

$$Z(k) = \Pi\left(\sum_{l=1}^{K} (P_l(k)Z(k-l) + Q_l(k)S(k-l)) + R\right)$$

with $A_l(k)(t,t') = c$, the number of tokens in the initial marking of the place between t' and t if $l = \zeta_t(k) \wedge \phi_{t'}(X_{t'}(k-1))$, ∞ otherwise (with a similar definition for B) and $P_l(k)(t,t') = 1$, if there is a place between t' and t and $l = \zeta_t(k) \wedge \phi_{t'}(X_{t'}(k-1))$, 0 otherwise (with a similar definition for Q).

This new version of the equations allows us to adapt the constant firing time algorithm to take into account the random firing time. However this adds heavier computations on each process. Indeed, at each step of the computation, we need to compute the value of $\zeta_t(k)$.

6 Performances

6.1 Data Parallel Simulation

Figure 5 shows some results of simulation with the Connection Machine. The time that we measured is not the total time of the simulation, but the busy time of the Connection Machine. Indeed, it is not really interesting to measure the total running time of simulations because they can be different between two executions for the same Petri Net due to time sharing protocols.

For the full matrix algorithm (first curve), the steep increases of simulation times correspond to creation of virtual processors ($|T| = 42, 62, 82$). The time of simulation using a sparse matrix technique (second curve) roughly coincides with the full matrix method for small systems but when the size of systems increases, we can see that for the sparse matrix algorithm, the results remain practically constant. Indeed, if we denote by p' the number of processors used by the Connection Machine, we have: $p' = \{2^n/2^n \geq |T|\alpha \geq 2^{n-1}\}$. We can note that p' is significantly smaller than p (p = number of used processors for full matrix), and so, even if the Petri Net has a large number of transitions, the sparse matrix method does not need virtual processors as in the full matrix version.

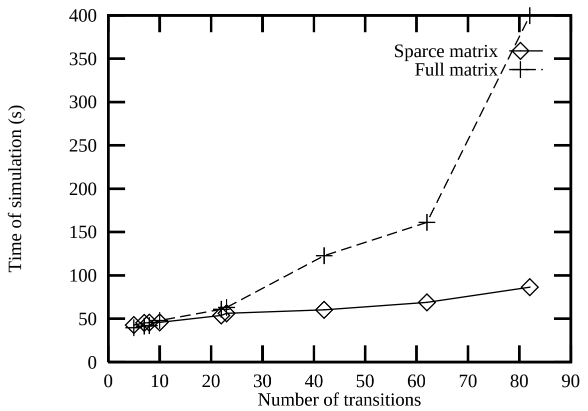

Figure 5: Comparison Full Matrix vs. Sparse Matrix.

We have also run comparisons between the parallel simulation on the Connection Machine and a simulation using a standard industrial simulation package (QNAP2) running on SS10 Sun workstation. The Petri Net used in these simulations consists of five jobshops interconnected by a routing network. The system contains around 150 places and 80 transitions.

The curves exhibit a quasi linear behavior for the data parallel algorithm while the conventional sequential simulation has a super-linear behavior. To simulate 1000 units of time, the ratio of simulation times between the two systems is around 45.

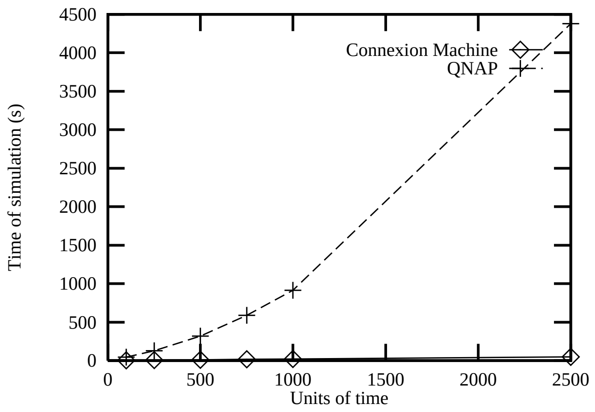

Figure 6: Comparison QNAP vs. data parallel algorithm

6.2 Distributed Simulation

Figure 7 shows some results of simulation. We can note that when the number of workstations increases, the time of simulation generally decreases. But we also have some disturbance phenomenon which is due to the message passing system (on an Ethernet Network in our case) as well as time sharing with other processes.

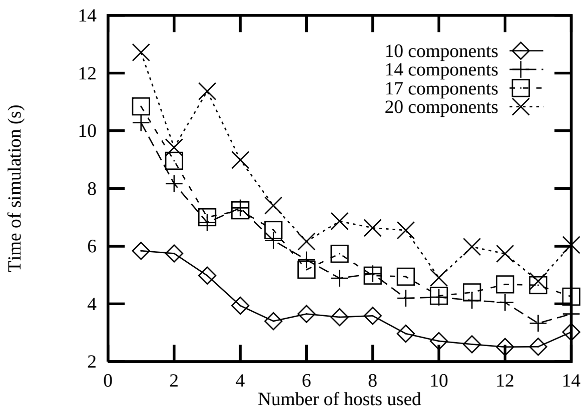

Figure 7: Performances using PVM.

References

[1] F. Baccelli and M. Canales. Parallel simulation of stochastic petri nets using recurrence equations. *ACM Transactions On Modeling And Computer Simulation*, 3(1):20–41, 1993.

[2] F. Baccelli, G. Cohen, G.J. Olsder, and J.P. Quadrat. *Synchronization and Linearity*. Wiley, 1992.

[3] F. Baccelli and B. Gaujal. Stationnary regime and stability of free-choice petri nets. In G. Cohen and J.P. Quadrat, editors, *11th Int. Conf. on Analysis and Optimization of Systems.*, volume 199 of *LNCIS*, pages 505–515. Springer Verlag, June 1994.

[4] K.M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(4):198–205, 1981.

[5] G. Chiola and A. Ferscha. Distributed simulation of timed Petri nets: Exploiting the net structure to obtain efficiency. In M. Ajmone Marsan, editor, *14th Int. Conf. on Application and Theory of Petri Nets*, volume 691 of *LNCS*, pages 146–165. Springer Verlag, 1993.

[6] B. Gaujal. *Parallélisme et simulation des systèmes à événements discrets*. PhD thesis, Université de Nice - Sophia Antipolis, 1994.

[7] B. Gaujal, A.G. Grenberg, and D. Nicol. A sweep algorithm for massively parallel simulation of circuit-switched networks. *Journal of Parallel and Distributed Computing*, 18(4):484–500, 1993.

[8] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3 user's guide and reference manual*. Engineering Physics and Mathematics Division.

[9] A.G. Greenberg, B.D. Lubachevsky, and I. Mitrani. Algorithms for unboundedly parallel simulation. *ACM Transaction on Computer Systems*, 9(3):201–221, 1991.

[10] D.R. Jefferson. Virtual time. *ACM Transaction on programming languages and systems*, 7(3):404–425, 1985.

[11] The Connection Machine© System. *C* Programming Guide*. Thinking Machines Corporation, 1990.

[12] T. Murata. Petri nets: properties, analysis and applications. In *IEEE*, volume 77-4, pages 541–580, April 1989.