Perfect Sampling of Load Sharing Policies in Large Scale Distributed Systems

Gaël Gorgo
LIG - INRIA-Grenoble
MESCAL Project
51, avenue Jean Kuntzmann
38330 Montbonnot, FRANCE
Gael.Gorgo@imag.fr

Jean-Marc Vincent
LIG - INRIA-Grenoble
MESCAL Project
51, avenue Jean Kuntzmann
38330 Montbonnot, FRANCE
Jean-Marc.Vincent@imag.fr

***Abstract*—In high performance computers, load sharing is essential to improve the efficiency of distributed computations. This paper provides a performance evaluation of load sharing policies, based on a queueing network model. Such models usually lead to Markov chains with a very large state space and an exact solution is extremely difficult to obtain. An efficient simulation method, derived from Propp & Wilson, perfect sampling, enables to sample states of the Markov chain according to the stationary distribution. Thus, the performances of load sharing systems, are estimated by an unbiaised sampling of their steady-states.**

When the events of the system are monotone, the monotone perfect sampling algorithm is used to reduce drastically the sampling time. A general index based model for the definition of load sharing policies is proposed and proved to be monotone. Then, large scale distributed systems, i.e. with thousands of processors, could be analyzed keeping a reasonable sampling time.

***Index Terms*—Load sharing systems, load sharing policies, performance evaluation, markovian systems, perfect sampling, monotone models**

I. INTRODUCTION

Load sharing algorithms are used to improve the performances of distributed systems, dividing up the workload among several processors. The load sharing policy should ensure that, the average idle time of ressources is minimized.

With the *Work Sharing* paradigm, load transfers are initiated when a node is overloaded and try to send some of its tasks to an under-loaded node. With the *Work Stealing* paradigm, the under-loaded nodes take the initiative: they attempt to steal tasks from overloaded nodes. *Work stealing* is nowadays the most used paradigm because of the intuition that transfers occurs only when necessary (when every nodes are busy, no transfer is done).

For many years, the proliferation of experimental distributed infrastructures have given rise to implementations of load sharing algorithms on real systems [3], [2], [7]. Experiments on these infrastructures give concrete results on the performances of distributed computations. However, the difficulty to measure the system state during its execution and the overhead due to a lots of phenomenons make that the analysis is difficult.

Performance evaluation of load sharing systems with theoretical tools is a hard problem since the state space of such systems is very large. Usually, analytical methods have a very high complexity even for relatively small systems. Previous works have predominantly used queueing theory, representing the evolution of the system by a Markov chain, and focus on the performances of the system when it has reached its steady-state.

Assuming some regular properties on the load sharing model, Maryse Beguin [8] obtains the exact response time and saturation probability of a large system with analytical methods. However, when the system is more complex, the solution can be obtained only for very small models, such as a two processors system.

To solve complex systems in large scale, approximation techniques are used by [9], [12], [6], [5]. Based on the decomposition of Markov chains and matrix geometric solutions, these methods enable to model various behaviours : delays induced by transfers of tasks ([9]), cache affinity in shared memory multiprocessors systems ([12]), hierarchical load sharing ([4]).

Mean field heuristic capture the limiting behaviour of dynamic markovian models as the number of objects grows to infinity, representing their behaviour by differential equations. A general approach of this method can be found in [1]. Maryse Beguin [8] uses the mean field method to study an extended spins model derived from physic which represents a load balancing model. Mitzenmacher [10] analyses load stealing models with a mean field approach. The advantage is the ability to model a large variety of behaviours, such as delays of transfer, heterogeneous systems, threshold decision. The drawback is the difficulty to prove that differential equations really converge to a fixed point. Moreover, the method doesn't provide accurate results when the number of objects is small.

In all of those works, simulations are used to validate the results. Simulations estimate the steady-state distribution based on long run of the Markov process. The drawback of simulation is the control of the warm up period, which depends on the size of the state space and the initial state.

In this paper, we use a perfect sampling algorithm, derived from Propp & Wilson [11] and implemented in Ψ^2 [15], a software dedicated to the simulation of finite size queueing networks. A simulation with

Ψ^2 provides a sample in which states are generated according to the stationary distribution of the markov process. We model a distributed architecture by a finite size queueing network on which task creations, task completions and transfers of tasks between queues are events.

The aim of this paper is to show the efficiency of discrete events simulation based on perfect sampling techniques for dimensioning load sharing policies in large scale distributed systems.

In particular, the objective is to define a scope of monotone events such as monotonicity techniques could be used to reduce the sampling time of perfect sampling.

The difficulty is to provide a method which is as efficient on small systems (about 10 processors) as on large scale systems (thousands of processors).

II. LOAD SHARING SYSTEM MODELLING

A. Model Architecture

In this section, we describe the architecture of the load sharing systems that we model in the following. Such systems are composed of several nodes which run at, the same time, the same algorithm.

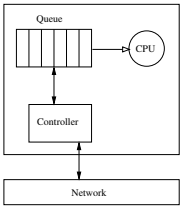

Fig. 1. Architecture of a node

A representation of a node is given in figure 1. A node has basically one activity which is the computation of tasks. The tasks that arrive at the node (tasks created by the node itself or tasks coming from other nodes) are stored in a queue. While the queue is not empty, tasks are carried out in sequence using a local sheduling policy.

To share the global workload among nodes, a control is performed on each node. A load sharing operation is typically the transfer of a task from a node to another node and is function of the decision of the local controller. The decision process is based on the local observation of the workload of the node. The controller then decides if the node is under-loaded, overloaded or in normal load. In the case where the load is not normal, the controller should find a target node for stealing a task or transfering a local task. When the operation is the stealing of a remote task, the system uses the *work stealing* paradigm, also called *pull* in this paper. When the controller transfers of a local task, the system uses the *work sharing* paradigm, also called *push* in this paper.

Decisions times of the controller could be driven by events (begining or completion of a task, arrival of a new task) or defined by internal timers which could be periodic for example.

In the execution, we suppose the control times negligeable, comparing with computation times of tasks. We also suppose that the transfer times are neglected (high speed network).

B. Events Modelling

We model a load sharing infrastructure by a queueing network with K queues, where a queue is the abstraction of a node. The state space of each queue Q_i is the set of integers $\mathcal{X}_i = \{0, \ldots, C_i\}$, where C_i is the capacity of queue Q_i. The state X^i of a queue Q_i, $X^i \in \mathcal{X}_i$, correspond to the number of tasks waiting in the queue and is called the load of Q_i. The state space $\mathcal{X}$ of the system is the Cartesian product of all $\mathcal{X}_i$; $\mathcal{X} = \mathcal{X}_1 \times \cdots \times \mathcal{X}_K$.

All the queues are in parallel, as represented in Figure 2. This implies that a task will never be computed more than one time. However, a task could be moved several times between queues until begining its computation.

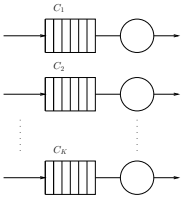

Fig. 2. Parallel Queueing Network

The system is characterized by a set of events $\mathcal{E}$ and a transition function Φ, defined on $\mathcal{X} \times \mathcal{E}$, which associates to each state x in $\mathcal{X}$ and each event e in $\mathcal{E}$, a new state $\Phi(x, e)$. In our distributed load sharing system, each queue Q_i is typically characterized by a set of 3 events $\{a_i, d_i, c_i\}$. The event a_i is a task *arrival* at Q_i, increasing x^i by one. The event d_i corresponds to a task *completion* at Q_i, meaning that the server has carried out a task and made decreased the load x^i by one. The event c_i is a *control* at queue Q_i, modelling an attempt of the controler to transfer a task. The application of the event c_i on a state x corresponds to the transfer of a task from a queue Q_k to another queue $Q_{k'}$ $(k, k' \in \{1, \ldots, K\})$ [1]. Then the load x^k decreases by one while the load $x^{k'}$ increases by one. The event c_i model the entire load sharing policy of the system and could be very complex. Details will be given in section II-C.

One could note that when we model a system wherein *control* times are driven by events (task arrival, task completion), *control* events should be applied at each time an *arrival* or *completion* event happens. To model those systems, the happening event (arrival or completion) and the control event are merged in only one event. This event performs the operations of both the happening event and the control event.

Notice that the triggering of an event depends on the state of the system. For example, the event *completion*

[1] Usually the controller evaluate the local load and, depending on the policy, transfer a local task ($k = i$) or export a remote task ($k' = i$)

can be executed only if the number of tasks in the queue is greater or equal to one. We consider that applying a *completion* event to an empty queue leaves the global state unchanged and more generally if an event cannot be applied, the corresponding transition is just a skip operation.

Example: We consider a small work sharing system. We model this system by two queues Q_1 and Q_2 both of capacity $C = 2$. The state space $\mathcal{X}$ of the system is the cartesian product $\{0, 1, 2\} \times \{0, 1, 2\}$ and the state of the system will be noted (x^1, x^2) or shortly x^1x^2. A representation of the system is given in figure 3. The dynamic of the system works as follow :

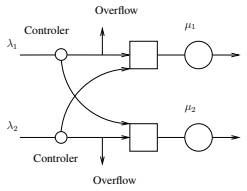

Fig. 3. Simple *push* queueing system

If the number of tasks in a queue Q is 0, then we say that Q is under-loaded. If there is one task, Q is normally loaded and finally, Q is said to be overloaded if its load is 2.

On a task arrival at Q_1, the Q_1 controller decide where the task should be allocated in function of the system state :

- if Q_1 is under-loaded, the task is allocated to Q_1.
- if Q_1 is normally loaded or overloaded while Q_2 is under-loaded, the task is transfered to Q_2.
- if Q_1 is overloaded and Q_2 is not under-loaded, the system get an overflow and the task is lost.
- In any other cases, the task is allocated to Q_1.

On a task arrival at Q_2, the Q_2 controler have the symetric behaviour of Q_1.

We model those mechanisms by the set of events $\mathcal{E} = \{a_1, a_2, d_1, d_2\}$. The event a_1 (resp. a_2) corresponds to a task arrival at Q_1 (resp. Q_2) and the event d_1 (resp. d_2) corresponds to a task completion at Q_1 (resp. Q_2), meaning that the task has been carried out. One could note that, in this example, *control* events are merged with *arrival* events. Let Φ be the transition function of the system, we give in Figure 4, for each event e, the definition of $\Phi(x, e)$ for all $x \in \mathcal{X}$ according to the dynamics defined above.

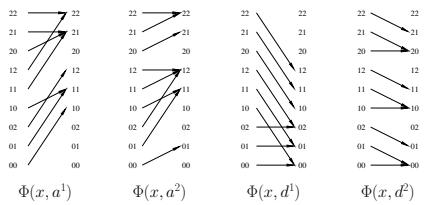

Fig. 4. Transition function of the system

C. Modelling load transfer events

In this section, we give a general scope for the definition of load sharing policies. It deals with the definition of transfers of load between queues. A transfer consists in moving a task from an origin queue to a target queue. The load sharing policy determines the choice of the origin queue and the target queue. A transfer is triggered, on a queue Q_i, by a local control event c_i. When the task is transfered from Q_i (the queue where the control event c_i occured), the policy uses the paradigm *push*. When the task is imported on Q_i, the policy is called *pull* or *work stealing*.

Consider a *control* event e, the choice of origin and target is performed by priority functions called index. An introduction to index functions can be found in [14]. We consider origin index functions that associates to each queue Q_i an index $I_i^{e,o}(x^i)$, then the origin queue is $o_e = \text{argmax}_i(I_i^{e,o}(x^i))$. In the same manner, we consider target index functions that associate to each queue Q_i an index $I_i^{e,t}(x^i)$, then the target queue is $t_e = \text{argmin}_i(I_i^{e,t}(x^i))$. The transition function associated to the event e is then defined by :

$$\Phi(x, e) = \begin{cases} x & \text{if } \ o_e = t_e \\ x - \mathbb{1}_{o_e} + \mathbb{1}_{t_e} & \text{else} \end{cases}$$

Modelling a distributed load sharing system, we associate to each queue Q_i a *control* event c_i and the corresponding index functions.

A first Join the Shortest Queue load sharing

The classical *Join the Shortest Queue* policy is extended to a load sharing policy :

when a control event c_i happens on Q_i, a task is transfered from Q_i to the less loaded queue of the system. For this policy, index are defined by :

$$I_j^{c_i,o}(x^j) = \begin{cases} +\infty & \text{if } \ j = i \\ -\infty & \text{else} \end{cases}$$

$$I_j^{c_i,t}(x^j) = x^j \text{ for all } j \in \{1, \ldots, K\}$$

Notice that the origin index functions are made such that we will always have $o = i$, meaning that the type of policy is *push*. Note that if Q_i is the less loaded queue, we have $t = i$, meaning that the target of the transfer is Q_i itself. In this case, the state is unchanged.

Threshold load sharing

The same JSQ load sharing policy is applied, but adding some threshold conditions.

If a potential target is under-loaded, i.e if its load is lower or equal to a low threshold θ_l, a transfer is possible on this queue. Then the less loaded queue is choosen among the targets that assert this condition. The Origin index family is the same as before. Target index are defined by :

$$I_j^{c_i,t}(x^j) = \begin{cases} x^j & \text{if } x^j \leqslant \theta_l \\ +\infty & \text{else} \end{cases} \quad \forall j \in \{1,\dots,K\},\ j \neq i$$

$$I_i^{c_i,t}(x^i) = x^i$$

The transfer of a task could be also conditionned by the state of the origin queue Q_i. We want a transfer to be done only if Q_i is overloaded, i.e if its load is greater or equal to a high threshold θ_h. Then, we have :

$$I_i^{c_i,t}(x^i) = \begin{cases} -\infty & \text{if } x^i \geqslant \theta_h \\ +\infty & \text{else} \end{cases}$$

In this last example, the index function $I_i^{c_i,t}(x^i)$ model the decision process of the queue Q_i which tell if an attempt of transfer must be done or not. In fact, when $I_i^{c_i,t}(x^i) = -\infty$, we have $t_{c_i} = \text{argmin}_k(I_k^{c_i,t}(x^k)) = i$ and no transfer is done. When $I_i^{c_i,t}(x^i) = +\infty$, a task is transfered to a target queue Q_j if there exists at least one $j \in \{1,\dots,K\}, j \neq i$ such as $I_j^{c_i,t}(x^j) \neq +\infty$.

A work stealing system

With the *work stealing* paradigm, when a queue reachs an under-loaded state, it steals a task on another queue selected according to a priority rule among other queues. If this queue is not able to provide some work, another attempt is made on another queue in the same manner and so on, until a suitable queue is found.

Definition 1. *A priority function γ is an application which associates to each queue Q_i, i in $\{1,\dots,K\}$, a priority level $\gamma(i)$ which takes values in $\mathbb{N}$. We consider that a queue Q_k have the priority on an other queue $Q_{k'}$ if $\gamma(k) < \gamma(k')$.*

To avoid ambiguity, we suppose that priorities have different values, that is $\forall i, j\ \gamma(i) \neq \gamma(j)$ (so the argmin is uniquely defined).

Given a threshold θ, a queue is under-loaded if its load is lower than θ, overloaded if its load is greater than θ and in normal load if its load is equal to θ. Given a priority function γ and considering the control event c_i on Q_i, we model a work stealing policy by the index functions :

$$I_j^{c_i,t}(x^j) = \begin{cases} -\infty & \text{if } j = i \\ +\infty & \text{else} \end{cases}$$

$$I_i^{c_i,o}(x^i) = \begin{cases} +\infty & \text{if } x^i \geqslant \theta \\ -\infty & \text{else} \end{cases}$$

$$I_j^{c_i,o}(x^j) = \begin{cases} -\infty & \text{if } x^j \leqslant \theta \\ \gamma(j) & \text{else} \end{cases} \quad \forall j \in \{1,\dots,K\}, j \neq i$$

At the opposite of the first example where the origin index functions are made such as the origin is always Q_i, we describe here a *pull* policy with target index functions that made Q_i always be the target.

D. System Dynamics

To explain the evolution of the system over the time, markovian assumptions are made in the modelling of task arrivals, task computations and load sharing controls. A Poisson process with rate λ_i is associated to each event e_i of the system. Those Poisson processes are supposed to be independent.

To simulate the Markov chain, we build a discrete time stochastic recurrence equation. To transform the continuous time model in a discrete model, we uniformized the process, which is then driven by the Poisson process with rate $\Lambda = \sum_{i=1}^{p} \lambda_i$ and generates at each time an event $e \in \mathcal{E}$ according to the probability distribution $(\frac{\lambda_1}{\Lambda},\dots,\frac{\lambda_p}{\Lambda})$. The uniformized process is proved to be equivalent to the initial queueing network Markov process in [13].

Let X_n be the n^{th} observed state of the system and $\{E_n\}_{n\in\mathbb{Z}}$ a random sequence of events, the system evolution is described by the equation :

$$X_{n+1} = \Phi(X_n, E_{n+1}) \tag{1}$$

Definition 2. *An **execution** of the system is defined by an initial state $x_0 \in \mathcal{X}$ and a sequence of events $\{e_n\}_{n\in\mathbb{N}}$. The sequence of states $\{x_n\}_{n\in\mathbb{N}}$ defined by the recurrence $x_{n+1} = \Phi(x_n, e_{n+1})$ for $n \leqslant 0$ is called a **trajectory**.*

The aim of discrete events modelling is to find an efficient method for the estimation of the stationary distribution of the Markov process. Section III shows that perfect sampling is based on this representation of the queueing network markovian model.

III. PERFECT SAMPLING

Simulation of Markov chains is a method to estimate the stationary distribution, when it is hard to solve it with analytical methods (state space explosion). According to Markov theory, under mild assumptions, an execution of the Markov chain starting from any initial state converges to the stationary distribution. The forward simulation then consists in running the chain from an initial state and stops after a sufficiently long defined time, called warm up period or burn-in time, such that the outputted state follows the stationary distribution. Sampling several states with this method, the stationary distribution characteristics are computed by a statistical estimation. However, there are sampling errors if the the burn-in time is not sufficiently long. Moreover, the forward simulation method doesn't enable to control this error.

The algorithm developped by by Propp & Wilson [11] allows to sample a state which follows exactly the stationary distribution. The idea is to use a backward coupling scheme, also called coupling from the past. Details are given in the following. Then the same methodology of large sampling and statistical estimation is employed to compute the result.

A. Backward coupling

Algorithm 1 Backward set-simulation of a Markov chain

Require:
Φ a transition function
$\{e_n\}_{n\leqslant 0}$ a backward independent events process

1: $n \leftarrow 0$
2: **repeat**
3: {Start from the past at time $-n$}
4: $\mathcal{Z} \leftarrow \mathcal{X}$
5: **for** $i = -n+1$ **downto** 0 **do**
6: $\mathcal{Z} \leftarrow \Phi(\mathcal{Z}, e_i)$;
7: **end for**
8: $\{\mathcal{Z} = \Phi^n(\mathcal{X}, e_{-n+1\to 0})$ the set of all possible states at time 0 knowing the innovation process downto $-n\}$
9: $n = n+1$
10: **until** $|\mathcal{Z}| = 1$
11: **return** $\mathcal{Z}$ $\{\mathcal{Z}$ is reduced to 1 state$\}$

Definition 3 (Global coupling time)**.**
The stochastic recursive system is **globally coupling** *if* $\tau \triangleq \text{argmin}_n\{|\Phi^n(\mathcal{X}, e_{1\to n})| = 1\}$; *is almost surely finite.* τ *is the* **global coupling time** *of the SRS.*

When the set of events is discrete, then the global coupling time property is directly concerned with synchronizing patterns :

Proposition 1 (Global coupling property)**.**
The system is globally coupling if and only if there is a finite sequence of events y (a synchronizing pattern) $e_1^s, \cdots, e_l^s$ *such that*

$$|\Phi^n(\mathcal{X}, e^s_{1\to n})| = 1.$$

Theorem 1 (Steady-state sampling)**.**
Provided the stochastic recursive system is globally coupling, the state generated by Algorithm 1 is stationary distributed.

To illustrate the backward coupling method, we give an example. We consider the small system introduced in section II (Figure 3). The arrivals rates and the completions rates are fixed such as $\lambda_1 = \lambda_2 = 0.6$ and $\mu_1 = \mu_2 = 1$. Then, the uniformized system is driven by the Poisson process with rate $\Lambda = \lambda_1 + \lambda_2 + \mu_1 + \mu_2 = 3.2$ and generates at each time of the process en event $e \in \{a_1, a_2, d_1, d_2\}$ according to the probability distribution $(\frac{\lambda_1}{\Lambda}, \frac{\lambda_2}{\Lambda}, \frac{\mu_1}{\Lambda}, \frac{\mu_2}{\Lambda}) = (0.1875, 0.1875, 0.3125, 0.3125) = p$.

In figure 5, we give an example of a backward coupling, generating at each time step an event which follows the probability distribution p. The principle is to go back at a time $-\tau$ sufficiently far in the past in such a way that the trajectories issued from all states at time $-\tau$ couple in one state before time 0. If this last assertion is verified, the state at time 0 follows the stationary distribution of the system. We can see that, in our example, taking $\tau = 14$ is sufficient because all the trajectories couple in state 01.

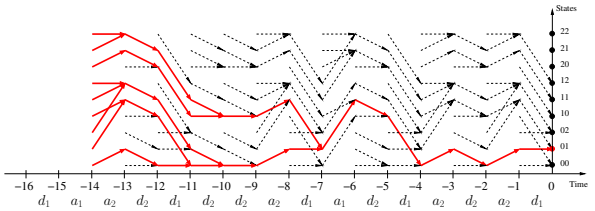

Fig. 5. Backward coupling starting from -14

This simple example shows how the backward coupling method works. At each step going back in the past from a time $-n$ to time $-(n+1)$, some trajectories are "lost". A trajectory is "lost" if it is issued from a state at time $-n$ which is not reached by any transition from time $-(n+1)$. As a consequence, when all the trajectories starting at time $-n$ and ending in state x at time 0 are lost, the state x become unreachable and the size of the coupling set, i.e the set containing the final coupling state, decreases. On our example, the states $\{22, 21, 20\}$ are unreachable since the time -1, the states $\{10, 00\}$ since time -2, and so on until the state 01 become the only reachable state.

Let's discuss now about how many steps are needed to obtain this scheme. Denote by τ^* the minimum number of time steps that enable coupling, called the coupling time. On this example, we have $\tau^* = 11$ and we show the obtained scheme, starting from time $-\tau^*$ in figure 6.

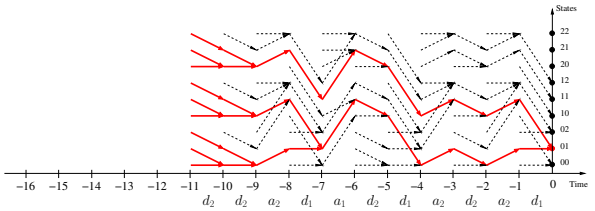

Fig. 6. Backward coupling starting from $-\tau^*$

We can see that, in the backward scheme starting at -14 (Figure 5), all the trajectories coupled since time -7, while in the one starting at -11 (Figure 6), the coupling isn't done before 0. Starting at any further time in the past may probably shift this coupling moment in the past again. However, running the process from any time $-\tau$ with $\tau > \tau^*$ will not change the output of this backward coupling scheme, because all the trajectories starting from time $-\tau^*$ end in the coupling state 01 at time 0.

B. Monotone perfect sampling

When a system is monotone, an interesting phenomenon happens to the backward coupling scheme : trajectories issued from a maximum and a minimum state of the system surround the trajectories issued from all others states. This property enables to get reasonable computation time for Perfect Simulation algorithms, drawing only two trajectories rather than as many as the size of the state space. This technique, which has been introduced in [11], is explained precisely in algorithm 2.

Algorithm 2 Backward monotone set-simulation of a Markov chain

Require:
Φ a monotone transition function
$\{\xi_n\}_{n\leqslant 0}$ a backward innovation process
$\mathcal{M}$ the set of extremal elements of $\mathcal{X}$

$n \leftarrow 0$
repeat
  {Start from the past at time -2^n+1}
  $\mathcal{Z} \leftarrow \mathcal{M}$
  for $i = -2^n+1$ **downto** 0 **do**
    $\mathcal{Z} \leftarrow \Phi(\mathcal{Z}, \xi_i)$;
  end for
  $\{\mathcal{Z} = \Phi^n(\mathcal{X}, \xi_{-2^n+1\to 0})$ the bounding set of all possible states at time 0 knowing the innovation process downto $-2^n\}$
  $n = n+1$
until $|\mathcal{Z}| = 1$
return $\mathcal{Z}$ $\{\mathcal{Z}$ is reduced to 1 state$\}$

Example: we show how this phenomenon holds in the simple system of Figure 3. The first stape is to show that our system is monotone.

We consider a lattice order on $\mathcal{X}$ which is represented as a Hasse diagram in figure 7. It is a partial order using component-wise ordering, meaning that, given two states $x, y \in \mathcal{X}$, we have $x \preceq y$ if $x^1 \leqslant y^1$ and $x^2 \leqslant y^2$.

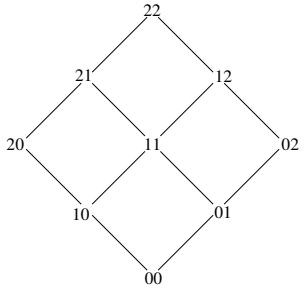

Fig. 7. Hasse diagram associated to the model of Figure 3

Definition 4. *An event $e \in \mathcal{E}$ is said to be **monotone** if it preserves the partial ordering (component-wise) on $\mathcal{X}$. That is*

$$\forall (x,y) \in \mathcal{X} \quad x \preceq y \;\Rightarrow\; \Phi(x,e) \preceq \Phi(y,e)$$

If all events are monotone, the global system is said to be monotone.

Using the representation of the transistion function of the system in figure 4, we can easily see that this one is monotone. We just have to check that there are not non monotone cases. A non monotone case could be seen like two transition arrows crossing each other, implying that the order hasn't been preserved after the application of the transition function. However, as we use a partial order, if the transition arrows are issued from two not comparable states, it doesn't matter. For example, the cross drawn by transition arrows $\Phi(02, a_2)$ and $\Phi(10, a_2)$ is not a non monotone case because 02 and 10 are not comparable. We could check that all other "cross" of Φ are similar to this last case and thus, the partial order is always preserved by Φ.

A monotone backward coupling scheme is represented in Figure 8. Trajectories issued from the maximum state 22 and the minimum state 00 are drawn starting at time $-1, -2, -4, -8$ and -16, following a doubling period scheme. Coupling is observed with $\tau = 16$ and the outputted state is 01, like in the backward coupling of Figure 6 and 5 (Of course, the same random sequence of events is used until time -11).

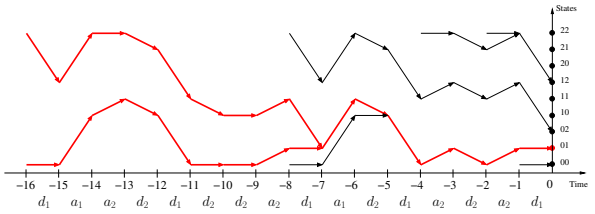

Fig. 8. Monotone backward coupling

IV. MONOTONICITY OF LOAD SHARING SYSTEMS

Under monotonicity assumptions, perfect sampling is an efficient method to simulate systems with large state spaces. Modeling a system with events, we know that a system is monotone if all the events are monotone. In queueing network models with finite size queues, a large scope of events are monotone, like the simple *arrivals* and *end of service* events for example. However, the modelling of a new class of systems could lead up to non monotone models. In this section, we show that the *control* events introduced in section II-C are monotone events. As we model those events with general index functions, this result implies that a large class of load sharing systems could be simulated with monotone perfect sampling.

A. Monotonicity of index model

We consider a *control* event e defined by :

$$\Phi(x,e) = \begin{cases} x \text{ if } x^i = 0 \\ x - \mathbb{1}_i + \mathbb{1}_j \text{ else} \end{cases}$$

where $i = \text{argmax}_{k=1}^{K}(I_k^{e,o}(x^k))$ (the origin of the transfer) and $j = \text{argmin}_{k=1}^{K}(I_k^{e,t}(x^k))$ (the target of the transfer)

Proposition 2. *if all index functions $I_k^{e,o}$ and $I_k^{e,t}$ are monotone and increasing, then the event e is monotone*

Proof: Let $x \preceq y$ two states and let e be a *control* event. Let i_x and j_x (resp. i_y and j_y) be the origin queue and the target queue for state x (resp. for state y)

$$i_x = \text{argmax}_{k=1}^{K}(I_k^{e,o}(x^k))$$
$$j_x = \text{argmin}_{k=1}^{K}(I_k^{e,t}(x^k))$$

$$i_y = \text{argmax}_{k=1}^{K}(I_k^{e,o}(y^k))$$

$$j_y = \text{argmin}_{k=1}^{K}(I_k^{e,t}(y^k))$$

The proof follows this scheme : we consider separately the process of *extraction* which consists in removing a task on the origin queue and the process of *allocation* which consists in placing the extracted task on the target queue. We show that those two processes are monotone. Then, the combination of extraction and allocation is also monotone.

Extraction

- **If** $i_x = i_y = i$ then $x - \mathbb{1}_i \preceq y - \mathbb{1}_i$
  Considering the bound case, we have :
  $$[x - \mathbb{1}_i]^+ \preceq [y - \mathbb{1}_i]^+$$
- **If** $i_x \neq i_y$
  $I_{i_y}^{e,o}(x^{i_y}) < I_{i_x}^{e,o}(x^{i_x}) \leqslant I_{i_x}^{e,o}(y^{i_x}) < I_{i_y}^{e,o}(y^{i_y})$
  then $x^{i_y} < y^{i_y} \Rightarrow x^{i_y} \leqslant y^{i_y} - 1 \Rightarrow x \preceq y - \mathbb{1}_{i_y} \Rightarrow x - \mathbb{1}_{i_x} \preceq y - \mathbb{1}_{i_y}$
  Considering the bound case, we have :
  $$[x - \mathbb{1}_{i_x}]^+ \preceq [y - \mathbb{1}_{i_y}]^+$$

Allocation

Denote by $\overrightarrow{C}$ the capacity vector $\{C_1, \ldots, C_K\}$ of the system.

- **If** $j_x = j_y = j$ then $x + \mathbb{1}_j \preceq y + \mathbb{1}_j$
  Considering the bound case, we have :
  $$(x + \mathbb{1}_j) \wedge \overrightarrow{C} \preceq (y + \mathbb{1}_j) \wedge \overrightarrow{C}$$
- **If** $j_x \neq j_y$
  $I_{j_x}^{e,t}(x^{j_x}) < I_{j_y}^{e,t}(x^{j_y}) \leqslant I_{j_y}^{e,t}(y^{j_y}) < I_{j_x}^{e,t}(y^{j_x})$
  then $x^{j_x} < y^{j_x} \Rightarrow x^{j_x} + 1 \leqslant y^{j_x} \Rightarrow x + \mathbb{1}_{j_x} \preceq y \Rightarrow x + \mathbb{1}_{j_x} \preceq y + \mathbb{1}_{j_y}$
  Considering the bound case, we have :
  $$(x + \mathbb{1}_{j_x}) \wedge \overrightarrow{C} \preceq (y + \mathbb{1}_{j_y}) \wedge \overrightarrow{C}$$

■

V. APPLICATIONS

In this section, the results obtained by monotone perfect sampling, using the Ψ^2 software ([15]) are discussed. All the simulation experiments were executed on a PC architecture with a Pentium 4, 2.8 GHz, 1Go RAM, Linux kernel 2.6.28-11-generic and the compiler gcc version 4.3.3. Simulation time estimations were obtained by using the gettimeofday primitive.

A. Policies comparison

We study load sharing policies on a small parallel architecture, composed of 8 nodes. Our aim is to compare the performances of the paradigms *push* and *pull*, when the control times, on each node, are driven by internal timers. We say, in this case, that the controller is independent from the computation. With both of those two paradigms, a transfer could be done from a sender node to a receiver node if the sender's load is greater than a threshold θ and the receiver's load is lower than θ. When the controller decides to transfer a local task or steal a remote task, it probes a randomly chosen node to know if a transfer is possible. If not, another node is probed in the same manner and so on, until a suitable place is found or the number of attempts reachs a static prob-limit l.

We apply the model of the section II on this system. The arrivals rate of tasks λ, the services rate μ and the controls rate ν are homogeneous on each queue. To model the policy described above, index are characterized by priority functions, like in the last example of section II-C. To model the random probing, the control event of a queue is divided in m control sub-events such that the rate of each sub-event is $\frac{\nu}{m}$. Each sub-event is characterized by a different priority function. Each priority function is defined according to a possible ordered combination of l queue among K, as a priority list. Thus, we have that $m = \frac{K!}{(K-l)!}$. Then, the random nature of the Markov process induces that this technique simulates a random probing.

As default setting, We take $\mu = 1$, $\nu = 1$ and we choose $\theta = 1$ for the decision threshold, that is, a queue is under-loaded if it is empty, in normal load if there is one task and overloaded if there are at least two tasks.

Statistical Estimation: The estimation of the mean response time of a task $\overline{r}$, for a given arrival rate λ, is obtained by the computation of the mean load of the system $\overline{l}$ and the application of the Little formula: $\overline{r} = \frac{\overline{l}}{K * \lambda}$. The sample size for the calculation of the mean load is $n = 1000$. As all samples are mutually independent, we apply the central limit theorem to compute confidence intervals at a level α. With $\alpha = 95\%$, the estimation error is lower than 10^{-1}.

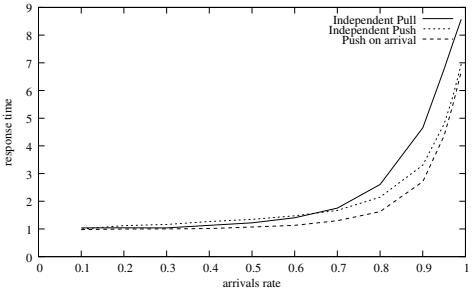

Fig. 9. Comparison of *push* and *pull* paradigms

Figure 9 shows the evolution of the response time of a task in function of the arrivals rate of tasks λ, for the policies *push* and *pull* with an independent controller and the policy *push* with a controller driven by arrivals.

Looking at the independent controller case, the tasks have a lower response time with the *push* paradigm than with the *pull* paradigm when the load is greater than 70%. Eager et al [6] finds that, at the opposite, *pull* is better than *push* at high load.

To analyze this contradiction, we use the following statement : "When the system is in a state such as one transfer could be done (unbalanced state), i.e there is at least one under-loaded node and at least one overloaded node, then performances are improved if

the policy realizes this transfer before some other events (arrivals, completions) change the state such as there is no possible transfer (balanced state). In the other case, we say the policy failed a transfer". Then, the best policy would have the lowest probability to fail the transfer.

Consider that the system is running for a long time with a load greater than 70% and has just reached an unbalanced state, i.e. there is at least one queue of which load is equal to 0 and at least one queue of which load is greater than 1. We analyze the probability of each policy to perform a transfer in this situation. This probability mainly depends on two factors which are the probability that a control be triggered before the system return to a balanced state and the probability that, once a control is triggered, the controller finds a suitable node for the transfer. With this model, the controller has an extremely low probability to fail because of the second factor. Thus, in this discussion, we only consider the influence of the triggering.

In the case where controls are driven by events, *pull* performs a transfer with probability 1 because a control is systematically triggered on an under-loaded queue which has reached its under-loaded state thanks to a task computation. With *push*, a transfer is done only if an arrival occurs on an overloaded node before the system become balanced with other events. In others words, depending on the following events, *push* could fail the transfer, so the probalility is lower than 1. Thus, when controls are driven by events, *pull* is intuitively better than *push*, this result is validated by experimentations of [6].

In the case of an independent controller with a control rate $\nu = 1$, a transfer is done if a control is triggered before any arrival occurs, leading up to a balanced state. With a *pull* policy, the control event musts be triggered on an under-loaded queue, while with a *push* policy, it must be triggered on an overloaded queue. As there are on average more overloaded queues than under-loaded queues when the load is greater than 70%, the probability that a control be triggered is higher for *push* than for *pull*. It justifies the result in Figure 9.

Figure 9 also shows that the policy *push on arrival* is better than *independent push* and *independent pull*, while the average number of controls is the same for every policies, because $\nu = \mu = 1$.

The conclusion of this discussion is that control times are a very influential parameter on the system performances, in particular for heavy loaded systems.

B. Parameter estimation

In the definition of a load sharing policy, *work stealing* and *work sharing* are fundamental paradigms which completly change the behaviour of the system. On the other hand, thresholds, prob-limit, number of tasks to transfer, are parameters of the policy which could be set with different values. Then, given a particular infrastructure and a type of application, the aim is to estimate the optimal value of each parameter. In this section, we estimate the performances of policies with an independent controller, varying the controls rate ν. The model of section V-A is also considered here. We make this experiment for both *push* and *pull* paradigms and look at the case where the load $\lambda = 0.9$.

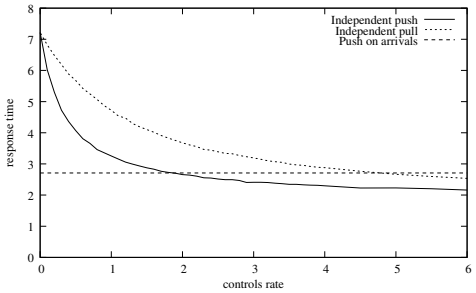

Fig. 10. Estimation of the control rate

Figure 10 shows the evolution of the response time when controls rate vary from 0 to 6. Even when the the controls rate increases, *pull* is still less efficient than *push*. However, *pull* outperforms the *push on arrivals* policy when the controls rate is greater than 5. In the same time, *independent push* is as good as *push on arrivals* with a controls rate approximatelly equal to 2, and the improvement obtained further is not significant. Thus, a system build such as controls are independents, will have to be set such as the controls rate be about the double of the processor speed of a node.

C. Scaling up

The aim of this section is to show that perfect sampling allows to compute the steady-state, even when the input model is very large. In this study, we examine the efficiency of the method, making experiments with Ψ^2. Then, the method is efficient if the program doesn't crash and the sampling time, i.e. the time needed to generate one state by a backward coupling scheme, stays reasonable.

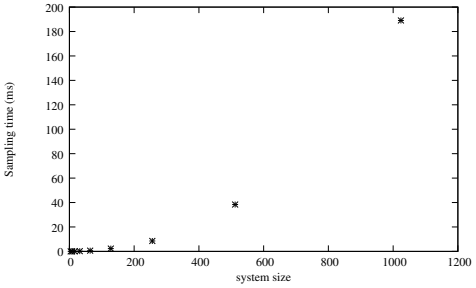

Fig. 11. Sampling time of the simulation

Figure 11 shows the evolution of the sampling time when the size of the system increases from 4 nodes to 1024 nodes. With a system of 1024 nodes in input, the sampling time is about 180 seconds, that is 3 minutes. Consider that a sample size of 1000 is sufficient to make the confidence intervals converge, then it will take 50 hours to obtain the result. One could say that it is relatively long to wait this time. However, it is possible to generate several small samples on several computers, and then merge them in one sample on which the statistics could be done. To ensure the

quality of the sample, random generators and random seeds of the computers have to be set correctly. Then, the time needed to obtain the result will be divided by the number of computers used. With five computers, for examle, the result is obtained in ten hours, that is one night.

As example, we study the performances of the *independent push* policy when the system size increases. We take $\nu = 2$ for the controls rate, $\lambda = 0.9$ for the input load and $l = 7$ for the prob-limit.

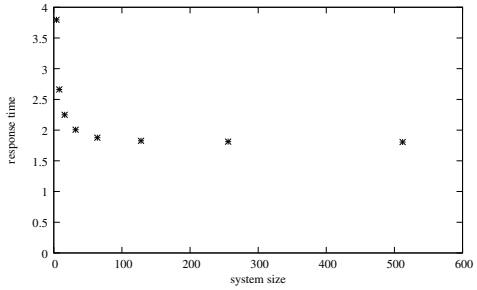

Fig. 12. Performance of *independent pull* in function of the system size

Figure 12 shows the response time of the *independent push* policy depending on the system size for a prob-limit $l = 7$. We can see that the response time reached an asymptotic bound as the number of nodes goes to infinity. Consequently, this result is in agreement with the mean field heuristic used in [1], [8], [10].

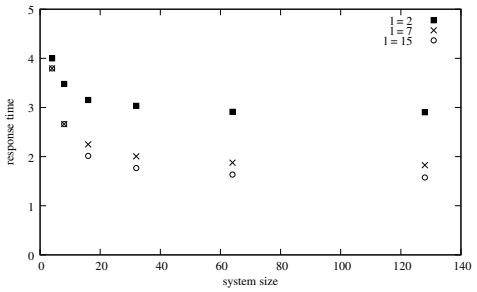

Fig. 13. Performance of *independent pull* depending on the prob limit

In Figure 13, we compare the performances of the *independent push* policy, setting the prob-limit with values $l = 2$, 7 and 15. This experiment shows that there is not a significant improvement taking $l = 15$ rather than $l = 7$. Consequently, in a system where the probing operation induces a cost, and then a lost in performances, it is interesting to estimate the value which ensure the best compromise.

VI. CONCLUSION

This work presents a method for the performance evaluation of load sharing systems, by the computation of an unbiaised sampling. Monotonicity is the key of efficiency, for the perfect sampling of models with large state spaces. Then, a large scope of load sharing policies could be described, with monotone index based models. However, when the control is triggered by task completions or when the number of transfered tasks is greater than one, the monotonicity is broken.

The advantages of this method are to provide accurate results and to be efficient with relatively large models. Moreover, perfect sampling could be used to estimate the error done by inaccurate methods such as approximations or mean field heuristic.

REFERENCES

[1] M. Benaïm and J.Y. Le Boudec. A class of mean field interaction models for computer and communication systems. *Perform. Eval.*, 65(11-12):823–838, 2008.

[2] R.D. Blumofe and C.E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[3] F.W. Burton and M.R. Sleep. Executing functional programs on a virtual tree of processors. In *FPCA '81: Proceedings of the 1981 conference on Functional programming languages and computer architecture*, pages 187–194, New York, NY, USA, 1981. ACM.

[4] S.P. Dandamudi, M. Kwok, and C. Lo. A comparative study of adaptive and hierarchical load sharing policies for distributed systems, 1998.

[5] D.L. Eager, E.D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Softw. Eng.*, 12(5):662–675, 1986.

[6] D.L. Eager, E.D. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Perform. Eval.*, 6(1):53–68, 1986.

[7] R.H. Halstead. Implementation of multilisp: Lisp on a multiprocessor. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 9–17, New York, NY, USA, 1984. ACM.

[8] B. Ycart M. Beguin, L. Gray and. The load transfer model. *The Annals of Applied Probability*, 8(2):337–353, 1998.

[9] R. Mirchandaney, D. Towsley, and J.A. Stankovic. Adaptive load sharing in heterogeneous distributed systems. *J. Parallel Distrib. Comput.*, 9(4):331–346, 1990.

[10] M. Mitzenmacher. Analyses of load stealing models based on differential equations. In *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 212–221, New York, NY, USA, 1998. ACM.

[11] J.G. Propp and D.B. Wilson. Exact sampling with coupled markov chains and applications to statistical mechanics. *Random Struct. Algorithms*, 9(1-2):223–252, 1996.

[12] M.S. Squillante and R.D. Nelson. Analysis of task migration in shared-memory multiprocessor scheduling. *SIGMETRICS Perform. Eval. Rev.*, 19(1):143–155, 1991.

[13] J.M. Vincent. Perfect simulation of queueing networks with blocking and rejection. In *SAINT-W '05: Proceedings of the 2005 Symposium on Applications and the Internet Workshops*, pages 268–271, Washington, DC, USA, 2005. IEEE Computer Society.

[14] J.M. Vincent and J. Vienne. Perfect simulation of index based routing queueing networks. *SIGMETRICS Perform. Eval. Rev.*, 34(2):24–25, 2006.

[15] J.M. Vincent and J. Vienne. Psi2 a software tool for the perfect simulation of finite queueing networks. In *QEST*, Edinburgh, sep 2007.