Estimation of network reliability on a parallel machine by means of a Monte Carlo technique

Mohamed El Khadiri, Raymond Marie, Gerardo Rubino

▶ To cite this version:

Mohamed El Khadiri, Raymond Marie, Gerardo Rubino. Estimation of network reliability on a parallel machine by means of a Monte Carlo technique. [Research Report] RR-1297, INRIA. 1990. inria-00075262

HAL Id: inria-00075262
https://inria.hal.science/inria-00075262

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1297

Programme 3
Réseaux et Systèmes Répartis

ESTIMATION OF NETWORK RELIABILITY ON A PARALLEL MACHINE BY MEANS OF A MONTE CARLO TECHNIQUE

Mohamed EL KHADIRI
Raymond MARIE
Gerardo RUBINO

Octobre 1990

IRISA INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX
FRANCE
Téléphone : 99.36.20.00
Télex : UNIRISA 950 473F
Télécopie : 99.38.38.32

Estimation of Network Reliability on a Parallel Machine by means of a Monte Carlo Technique

Mohamed El Khadiri, Raymond Marie, Gerardo Rubino
IRISA
Campus de Beaulieu, F-35042 Rennes Cedex, FRANCE

31 août 1990
Publication Interne No. 545 - 20 pages

Abstract. We consider the evaluation of reliability measures of communication networks. The used models are stochastic graphs and the exact computation of most of the standard reliability measures are very expensive in computational time. We explore here the estimation of these measures with a Monte Carlo method and using a powerful parallel computer. To illustrate the work, we consider a basic measure in the area, the 2-terminal (or source-to-terminal) reliability. In the general case, the computation of this measure belongs to the NP-hard class. The chosen machine is the iPSC2 hypercube of Intel. The Monte Carlo method together with the computational power of the chosen computer allows the evaluation of large networks while exact algorithms usually fails due to their high cost in time. The paper discusses the architectural aspects and the algorithmic solutions that we adopted to implement the technique.

NETWORK RELIABILITY, MONTE CARLO SIMULATION, iPSC2

Estimation de la fiabilité d'un réseau sur une machine parallèle par une méthode Monte Carlo

Résumé. Nous considérons ici le problème de l'évaluation de mesures de fiabilité pour les réseaux de communications. Ce problème appartient à la classe des NP-durs et sa résolution par des méthodes directes, même dans le cas de petits modèles, est prohibitive en temps de calcul. Dans cet article nous étudions l'estimation de ces mesures par un algorithme de type Monte Carlo mis en œuvre sur une machine parallèle à mémoire distribuée. Pour illustrer la méthode, nous traitons le problème de la fiabilité source-terminal d'un réseau de communications modélisé par un graphe stochastique. Cette technique d'évaluation et l'utilisation d'une machine puissante, l'iPSC2 de Intel, nous ont permis d'évaluer de très grands réseaux.

FIABILITÉ DES RÈSEAUX, MONTE CARLO, iPSC2

1 Introduction

In communication network modelling, one of the approaches to evaluate quantitatively the behaviour of the system from a reliability point of view is to use stochastic graphs. The network is represented by a graph and the components (nodes and links) are wighted by probabilities corresponding to the respective reliabilities of the elements. Let G be such a graph. Here G is supposed to be undirected, connected and without loops. For simplicity, we assume that the nodes are perfect (no message is destroyed or lost in them) but at an instant τ of interest, each link is in one of two states: either "working perfectly" or "completly down". Let X_l be the random variable defined by

$$X_l = \mathbf{1}_{\{\text{at time } \tau, \text{ link } l \text{ is working}\}}$$

where $\mathbf{1}_A$ denotes the indicator function of the event A. In other words, X_l takes the value 1 iff link l is working at time τ, 0 otherwise. Let us assume that the X_l's variables are independent and let r_l denote the reliability of link l, that is, $Pr(X_l = 1) = r_l$. At instant τ, the subset of working links defines a subgraph $\tilde{G}$ of G. There are many interesting measures that can be evaluated from this model. A basic one will be consider here but the following ideas can be applied to a wide class of problems.

Let us fix two nodes s and t of G. The measure which is considered in this paper is the probability that there is at least a path in $\tilde{G}$ between s and t. This number will be denoted by R_{st} in the sequel. If Y denotes the random variable

$$Y = \mathbf{1}_{\{\text{at time } \tau, \text{ the nodes } s \text{ and } t \text{ are connected in } \tilde{G}\}}$$

we have $Pr(Y = 1) = R_{st}$. There are many variations around this type of measure and the reader can find references for instance in [13]. See also [1] for a general survey and [14] for details on two efficient exact algorithms.

In the general case, this is a NP-hard problem [12] and it remains in this class even for restricted families of topologies (for instance, it remains NP-hard in planar graphs or in graphs with bounded degree [9]). More specifically, the computational time required to solve a medium size model (say many dozens of components) is in general prohibitive on a workstation. On the other side, if we accept a probabilistic answer, a Monte Carlo technique [8] can produce it more quickly (actually in polynomial time). This is the main topic of this work. Our goal is to deal with very large topologies (more than one thousand components) in order to evaluate the ability of the approach to give solutions for the case of non applicability of exact techniques. We have not analysed here the statistical concerns with the Monte Carlo approach; the interested reader can see [5],[10]. Another paper related to this problem is [15] in which planar topologies are considered.

The paper is organized as follows. The following section introduces the machine and its architecture. Comments on the algorithmic decisions that must be taken in this context are also given. In Section 3 we discuss in some detail the implementation. In particular, we discuss the main algorithm and the distribution of tasks on the hypercube, the used Monte Carlo method and the pseudo-random number generation problem on a multiprocessor machine. Section 4 contains examples of executions and the conclusions of this work are reported in Section 5.

2 The iPSC2 machine

The iPSC2 computer [6] [7] is a multiprocessor machine with distributed memory, composed of 2^n nodes numbered from 0 to $2^n - 1$, connected according to a hypercube (or shortly, cube) topology: Each node is directly connected to n other nodes and two nodes are neighbours iff the Hamming distance between the binary representation of their numbers is 1 (see Figure 1). Observe that, as a graph, an hypercube contains subgraphes corresponding to useful regular topologies such as rings, 2D grids (if $n \geq 4$), 3D grids (if $n \geq 6$), binary trees, etc. See [18] for an analysis of the hypercube topology. Each node of the iPSC2 is first composed of a 80386 processor and has at least 1 Mbyte of RAM. Moreover, there is a 80387 coprocessor and a cache of 64 Kbytes. Each node has no direct access to the local memory of the other nodes and there is no global memory.

As a distributed machine, the cooperation between the nodes is achieved by message passing. The iPSC2 offers to the programmer a communication network between its nodes which is built on dedied hardware: each node has a specialized processor called DCM (Direct Connect Module) [16] which avoids the utilization of the 80386 CPU and of the local RAM for the routing task. A PC/386 frontal allows to share the whole cube between several users: Each of them can then allocate any number of nodes. Also, the frontal processor loads the processes on the nodes and kills them when necessary. The whole system runs under UNIX V with supplementary specialized primitives. Several processes can be running on the same node. Each process can be identified by its number together with the identifier of the node on which it is running.

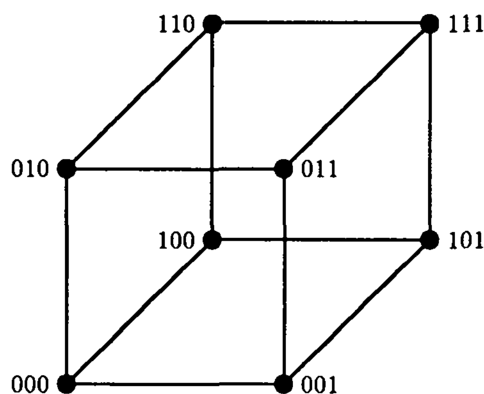

Figure 1: The 3-cube topology.

A program is then a set of communicating sequential tasks running on the different nodes. As a consequence of the machine architecture, the algorithms must satisfy certain rules [17]. For instance, the different tasks must equilibrate to optimally use the set of nodes. Also, the overhead induced by the exchange of messages must be controlled in order to avoid spending most of the time in communications. Concerning the programming aspects, let us also mention that the operating system allows the exchange of messages in two modes: with and without blocking. In the first one, a node waiting for (resp. sending) a message is blocked until the message arrives (resp. until the message leaves the node).

The estimation of the R_{st} measure by a "crude" Monte Carlo method consists of repeating

N times the following experience. For each link l a Bernoulli trial is performed with the distribution $(r_l, 1 - r_l)$. A subgraph $\tilde{G}$ is obtained and the value of Y is calculated. Then, if we denote by Y_i the value of Y in the ith replication, the estimator of R_{st} is

$$\widehat{R}_{st} = \frac{Y_1 + Y_2 + \cdots + Y_N}{N},$$

that is, the number of times that s and t are connected in the resulting subgraph, divided by the total number of generated subgraphs. To implement this experience in a computer (to simulate it) we need a pseudo-random number generator giving pseudo-realizations of a random variable $X \in \{0, 1\}$ and a procedure to decide if s and t are connected in $\tilde{G}$ or not (in fact, we use a more efficient Monte Carlo technique presented in Subsection 3.2).

Our first architectural decision was the choice between a *distributed* simulation in which each node takes in charge only a part of the network and a *parallel* simulation, in which each node receives a copy of the whole graph and performs independently local replications. We adopted the second solution since it reduces to a minimum the amount of necessary communications: each node has only to send the results of its local replications to a central supervisor who controls the whole application. We experimented some versions of the first one and they did not behave very well. In particular, the number of exchanged messages was proportional to the number of Bernoulli trials. The following section reports on the details of the finally chosen implementation.

3 Implementation

In this section we describe the architecture of the application and some of the problems that we had to solve to implement it.

3.1 Load balancing

Let us denote by C the number of allocated nodes of the iPSC2. Assume that we must perform N replications. From the load balancing point of view, the immediate method consisting of assigning N/C replications to each processor and then waiting that every one terminates its part of the work is not efficient. This is due to the non deterministic execution time for each replication. In particular, if the model has *bridge*-like regions (a *bridge* is an edge such that its delation makes the graph unconnected) we may have high variations in this execution time according to the fact that the *bridge*-like region allows the passing of messages or not. To illustrate this, we show in Figure 2 the variation of the necessary time to perform N/C replications when 64 processors are allocated and when the previous approach and the Monte Carlo algorithm presented in Subsection 3.2 are used. A large network is evaluated: there are 3047 edges, each with the same elementary reliability r. The graph has exactly three bridges. The total number of replications to perform is 1000 times the number of processors, that is 64000 replications. We plot the coefficient of variation (square root of variance divided by the absolute value of the mean) of the time used by a processor to perform its 1000 replications against the value of r. Observe that the drift can be of about 25 times the mean time of 1000 replications if the elementary reliabilities are high. Observe also that in real networks models, these elementary reliabilities are usually high.

Instead, we implemented a dynamic distribution of the load on the cube, taking into account the evolution of the work of each node. There are two types of processes here called

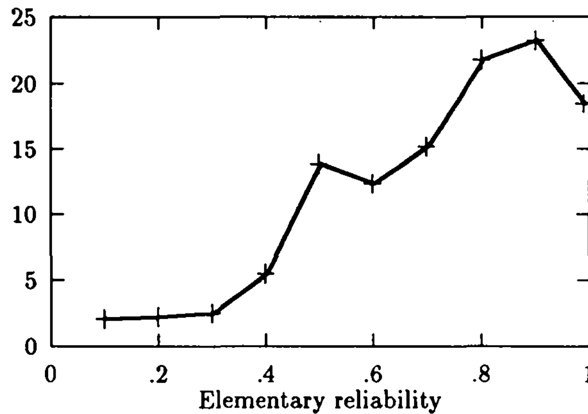

Figure 2: Coef. of variation in % of the time of execution of 1000 replications (sample size: 64 processors)

control process (or shortly *controller*) and *simulation* process. One of the C processors (the *master* node) runs alternatively a process of each type and the remaining $C-1$ nodes execute only a *simulation* process. The idea is the following. At the beginning, the control process assigns a *quota* to each simulation one, that is, a number of replications to be performed. This quota is the same for all the receivers and its value is $ceiling(N/C)$ where $ceiling(x)$ is the smallest integer greater than or equal to the real number x. When a simulation process achieves its quota it sends a report to the controller containing the total number of performed replications (in fact, the total number of replications that have been performed since the previous report) and the number of successes in these simulations. There will be a first report to arrive at the control process. Then, the control process asks each one of the remaining simulation processes for a report independently on the number of already performed replications. When all the reports have arrived, the controller computes the total number of already performed replications, say K, it updates also the total number of successes and it sends a new quota to all the simulation processes equal to $ceiling((N-K)/C)$ if $K < N$, 0 otherwise. Coming back to the simulation process job, once it sends its report, it starts a new series of replications and after each one, it looks at its local mailbox for a new quota. When this new quota is 0, it sends a last report and stops. If it already achieved its new quota (or bypassed it), it sends a new report. Otherwise, its next report will be sent when it will achieve this new quota or when a request will arrive from the control process. In this way, the simulation processes perform the replications continuously until the end of the application, which guarantees the load balancing. In the next section, we illustrate the speed-up obtained with C processors as a function of C (see Figure 5).

In a more detailed way, the tasks of the two processes are described below. We use a simple description langage with some particularities which are specific to the distributed application. The objective is to be precise enough in the presentation of the algorithms without being too technical with the programming details.

Processes communicate with each other by message passing. The messages may be of any data type and format. Also, "empty" messages (messages without data) can be used as signals to coordinate the processes. In our description langage, when a process sends a message it uses the `SEND` primitive that has three parameters: a message's type, the message's

buffer and the receiver's address. On the other side, the primitive READ allows a process to read a message from its mailbox. Reading a message is a *consummer* task, the message leaves the mailbox. This primitive has only two parameters, identical to the first two parameters of SEND.

We use three types of messages: *mesg_quota*, *mesg_request* and *mesg_report*. Messages of the first type are sent by the control process to the simulation processes and contain an integer, the quota value. The *mesg_request* messages are empty and they are used by the controller just to ask for reports. The messages of the *mesg_report* type are sent by the simulation processes to the controller. They are structured with two integer fields called *done* and *success*. The first one contains the number of performed replications (from the last report) and the second one contains the number of successes.

We use a function SENDER(mesg_type) returning the address of the process which sent the oldest message of the given type in the mailbox. The primitive WAIT(mesg_type) stops a process until a message of the given type has arrived. The primitive FLUSH(mesg_type, process(es)) allows to destroy all the messages of the given type in the mailbox of the named processes. Last, the function *oneReplication()* performs a replication (see the next subsection) and return 1 or 0 according to the fact that the result is a success or not.

The control process cummulates the number of successses in its counter *totalSuccesses* and the number of performed replications in *alreadyDone*. The coresponding variables in each simulation process are called *successes* and *done* respectively.

- **Simulation process.**

```
done := 0 ; success := 0
loop
     if (mesg_quota is present) then
          READ(mesg_quota, quota)
          if quota ≥ 0 then
               for i := done + 1,quota do
                    if (mesg_request is present) then
                         i := i - 1 ; break /* for */
                    endif
                    success := success + oneReplication()
               endfor
               done := i
          endif
            /* send report :  */
          report.done := done ; report.success := success
          SEND(mesg_report, report, control process)
          if quota = 0 then
               break /* loop */
          endif
          done := 0 ; success := 0
     else /* no mesg_quota message in mailbox */
          success := success + oneReplication()
          done := done + 1
     endif
endloop
```

- **Control process.**

```
alreadyDone := 0 ; totalSuccess := 0
while alreadyDone < N do
    /* distribute the current quota :  */
  quota := ceiling((N - alreadyDone)/C)
  SEND(mesg_quota, quota, every simulation process)
    /* wait for a first report and update counters :  */
  WAIT(mesg_report)
  analyzeReport()
    /* request report to every other simulation process */
  p := SENDER(mesg_report)
  SEND(mesg_request, empty, every simulation process exept p)
    /* receive all the remaining C - 1 reports :  */
  for j := 1,C - 1 do
      WAIT(mesg_report)
        /* update counters */
      analyzeReport()
  endfor
    /* prevent for any remaining request message :  */
  FLUSH(mesg_request, every simulation process)
endwhile
  /* send stop signal :  */
quota := 0
SEND(mesg_quota, every simulation process)
for j := 1,C do
    WAIT(mesg_report)
    analyzeReport() /* last update of counters */
endfor
R := totalSuccess/alreadyDone /* computing the output */
end

procedure analyzeReport()
    READ(mesg_report, report)
    alreadyDone := alreadyDone + report.done
    totalSuccess := totalSuccess + report.success
end
```

3.2 The used Monte Carlo algorithm

As stated before, the illustrative problem is the so called *source-to-terminal* or *2-terminal* one, in an undirected context. Let us denote by *nbNodes* the number of nodes of the graph G and by *nbEdges* the number of edges.

Instead of performing *nbEdges* Bernoulli trials and then using a routine to see if the nodes s and t are connected in the resulting sub-graph of G, our approach consists of doing these two tasks at the same time. As it is usual in connectivity tests, we perform a depth-first search starting with node s. Before visiting a new node y from an already visited node x, we perform a Bernoulli trial for the link $\{x, y\}$. If the result is not a success, the algorithm

behaves as if the link $\{x, y\}$ does not exist and the search continues. In this way, the number of Bernoulli trials in each replication is diminished (see Figure 7 in the next section).

Moreover, it is clear that the way in which the search algorithm finds the successive vertex that are adjacent to a given node is relevant on the time necessary to detect the connection between s and t or to decide that the two nodes are not connected. We used a dictionary of adjacent nodes sorted according to their respective distances to the terminal node t. More specifically, when starting from node x, the depth-first search first consider, among the nodes adjacent to x and not yet visited, the node y which is the closest one to t. This method has behaved much better than a blind exploration; it is called here the "topologic" algorithm since it depends strongly on the topology of the underlying graph (see Section 4). In [10] further details on the topologic algorithm and its behaviour compared with other Monte Carlo methods are given.

3.3 Independent random number generators

We used a classical congruential pseudo-random number generator [3]:

$$U_{k+1} = aU_k \bmod m, \quad k \geq 1$$

where $m = 2^{32}$ (the iPSC2 is a 32-bit machine), U_0 is relatively prime to m and $a = 3 \bmod 8$. The period of such a sequence is 2^{30}.

To implement a parallel simulation, we need C independent sequences of random numbers where C is the number of allocated nodes. To solve this problem, let us consider the whole sequence $U = (U_k)_{k=1,2,\ldots,2^{30}}$. Assume that we need 2^J independent random sequences to be used by 2^J processors in our cube (that is, assume that $C = 2^J$). The general case in which C is not a power of two is not a problem once this case is solved. We divide the sequence U into 2^J parts in order to build 2^J generators (see Figure 3). If U^j denotes the jth generator, $j = 1, 2, \ldots, 2^J$, the corresponding recurrence is

$$U^j_{k+1} = aU^j_k \bmod m, \quad k \geq 1$$

$$\text{where } U^j_1 = U_{(j-1)2^{30-J}+1}.$$

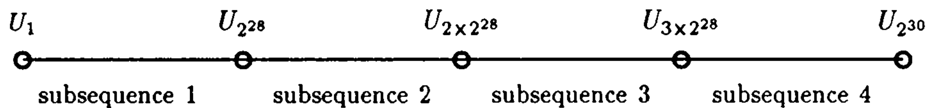

Figure 3: Partitionning the sequence U into 2^2 parts

To compute U^j_1, observe that

$$U_k = U_1 a^{k-1} \bmod m \quad k \geq 1.$$

Then,

$$\begin{aligned} U^j_1 &= U_{(j-1)2^{30-J}+1} \\ &= U_1 a^{\left((j-1)2^{30-J}\right)} \bmod m \\ &= U_1 \left(a^{\left(2^{30-J}\right)}\right)^{j-1} \bmod m. \end{aligned}$$

We need only to compute the factor $b = a^{(2^{30-J})} \bmod m$ which is done executing $30 - J$ times the sequence

$$x := x^2 \bmod m$$

where x is a variable initialized to a. Last, we compute

$$U_1^1 := b$$

$$\text{and } U_1^{j+1} := bU_1^j \bmod m, \quad j = 1, 2, \ldots, 2^J - 1.$$

4 Results

Consider the family of graphs (s_k) illustrated in Figure 4. The number of edges in s_k is $2k^2 - 2k + 1$. The first graph s_1 has only one edge between s and t. A simple duality argument gives that if every line in s_k has elementary probability 0.5 then $R_{st} = 0.5$. This result is used here to test the algorithm on arbitrarily large networks with a "reasonable" connectivity and a known exact value for the answer.

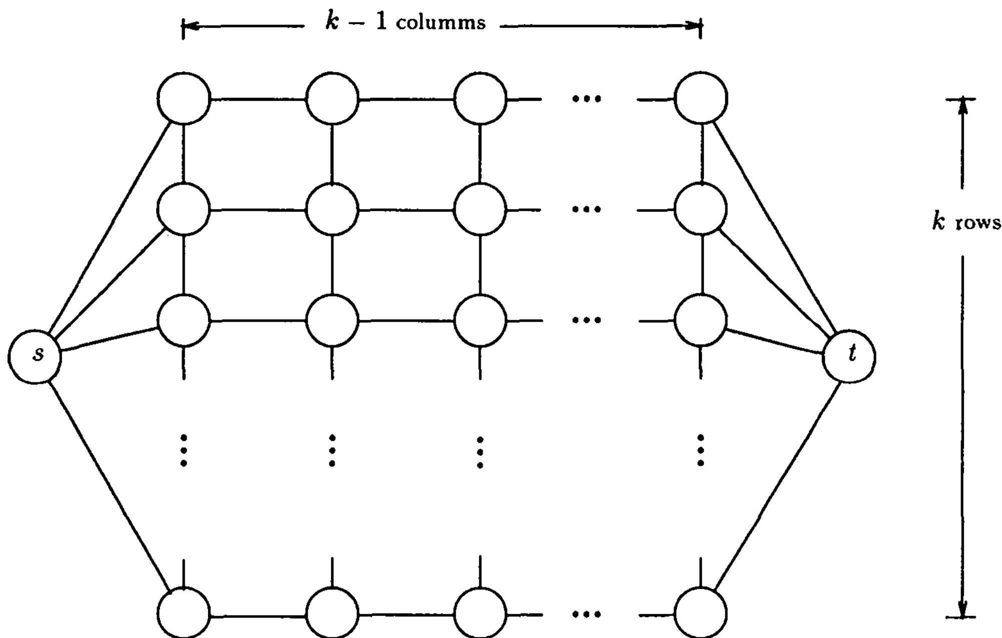

Figure 4: The s_k topology

First, we show in Figure 5 the speed-up obtained with C processors as a function of C. The considered measure is the time necessary to perform a sequential simulation divided by the time used by our algorithm on C processors [4]. Let us denote this ratio by $S(C)$. The plot corresponds to the execution time on s_{40} (3121 edges) with all the edges having the same elementary reliability $r = 0.5$. Observe that the optimal situation is $S(C) = C$. This kind of behaviour is obtained when a perfect load balancing is achieved and when the communication time is negligible. In our case, there is a small drift of the speed-up with respect to the ideal curve when C increases since the number of exchanged messages increases with C. The fact

that there are two processes running in node 0 introduces a very small overhead : we measured $S(1) = 0.9776$. In Annex A we propose a simple analytical model to have some insight on this behaviour.

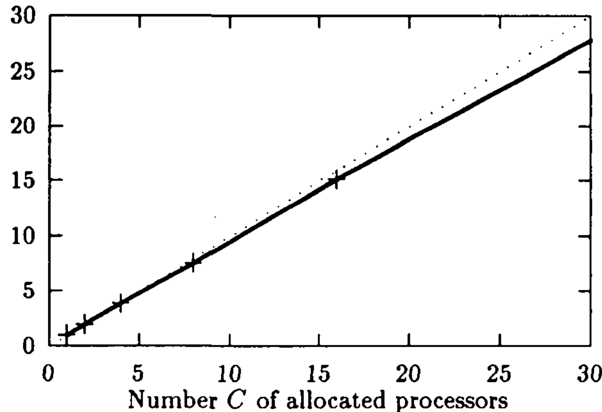

Figure 5: Speed-up as a function of the number of allocated processors (the time of a sequential simulation on 1 processor divided by the time of our algorithm running on C processors.

On the same family of graphs, we analysed the execution time as a function of the (common) elementary reliability r. In Figure 6 we show this variation on two graphs, s_{40} (3121 edges) and s_{60} (7081 edges). In the y-axis we put the mean execution time per replication, computed on a 64-processors cube. This mean is evaluated using a sample of 64000 replications. It is given in millisecs. The mean time per replication is greater in the case of s_{60} since there are much more paths than in the s_{40} graph and they are longer. Since the two graphs have a similar structure, the two curves have roughly the same shape. We can also observe that the difference between the mean execution times is smaller when $r \approx 0$ or $r \approx 1$. In the first case this is due to the fact that regardless to the number of lines in the graph, only the "neighbours" of s are relevant since the result is very often that s and t are disconnected, and this is detected very quickly. In the second one, the determinant factor is the distance between s and t since the lines are working almost for every trial. In Figure 7, we compare the algorithm with a version using in each simulation process a crude Monte Carlo method. We plot the ratio between the mean expectation time per replication in the two cases, called *gain* in the caption of the figure, always on s_{40} and s_{60}, as a function of r. As we can expect, the gain is smaller when $r \approx 0.5$. See, for instance, that on s_{40} our algorithm works about 17 times faster than the crude implementation when $r \approx 0.9$. Observe that in general, in "real" models the values of the elementary reliabilities are usually high.

The (s_k) family allows an efficient analysis of the variation on the execution time when the size of the network increases. We performed as usual 64000 replications on a 64-processors cube on s_{10j}, for $j = 1, 2, \ldots$ We computed the ratio between the execution time to evaluate s_{10j} and the execution time to evaluate s_{10}. This ratio is plotted in Figure 8. In the x-axis we put the size of the graphs where the unit is the size of s_{10}. We show four curves, two for the proposed algorithm and two for the version that uses crude Monte Carlo. In each cases, we measured the ratio when $r = 0.5$ and when $r = 0.9$. The important point to observe here is that in our solution, this *increase* ratio is below the $y = x/2$ line. Moreover, the ratio is sensitive to the value of r (which is not true in the crude Monte Carlo version) with smaller

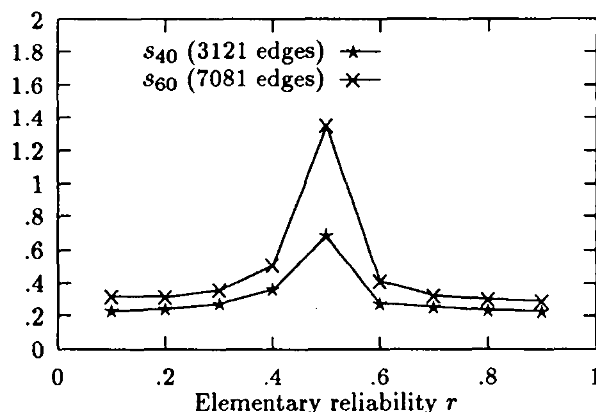

Figure 6: Mean time in millisecs per replication with 64 processors as a function of r, the elementary reliability of the edges.

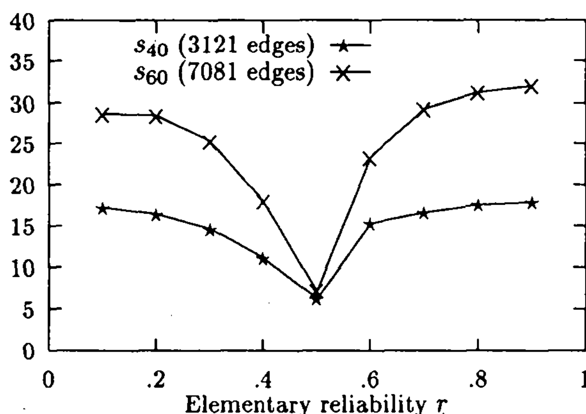

Figure 7: Gain of the *topologic* algorithm over *crude* Monte Carlo as a function of r, the elementary reliability of the edges (execution times ratio of the first one over the second).

values for high elementary reliabilities. Other tests with different values of r and different graphs confirm this behaviour.

To give an idea about the cost in time to obtain a reasonable answer on a large network, we present some values of the execution time computed with 64 processors and the respective 0.95% confidence intervals. We used a Gaussian approximation since the number of replications was very high. We evaluated a graph composed by 20 copies of s_4 in parallel (sharing the same source and terminal vertex) with all the elementary reliabilities equal to 0.5. The graph has $20 * 25 = 500$ edges. The theoretical reliability R_{st} is $1 - (0.5)^{20} = 0.99999905$. We performed 20098263 replications. The execution time was 8.96 minutes. The estimation of R_{st} was 0.99999925 and the size of the confidence interval was 0.00000076.

Consider now a series of four copies of s_{20} relied by its respective terminal and source points with bridges. This gives a network with $4 * 761 + 3 = 3047$ edges. Table 1 gives the execution time in secs together with the value of the estimation of R_{st} and the confidence interval

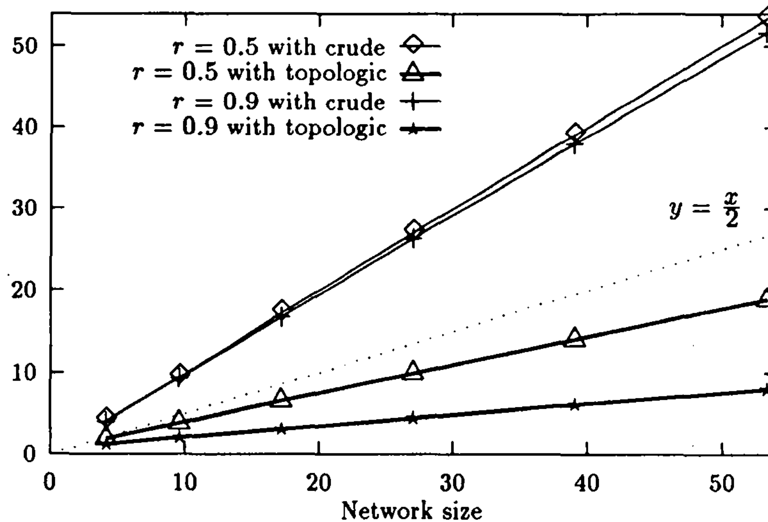

Figure 8: Execution time on a graph with $181m$ edges, divided by the execution time on a graph with 181 edges, as a function of m. The number of replications is 64000 and we use 64 processors.

size, for three different number of replications, in the case of the elementary reliabilities all equal to 0.99. In Table 2, the same information is presented when the edges have elementary reliabilities equal to 0.5. In this case, the theoretical reliability R_{st} is $(0.5)^7 = 0.0078125$.

# repl.	estimation	conf. int. size	exec. time
70344	9.7067269e-01	2.4937047e-03	16.6 secs
118617	9.7087264e-01	1.9140104e-03	21.5 secs
265608	9.7066730e-01	1.2834434e-03	43.1 secs

Table 1: case of $r = 0.99$

# repl.	estimation	conf. int. size	exec. time
104940	7.2994092e-03	1.0300743e-03	50.9 secs
202251	7.7032994e-03	7.6207882e-04	94.4 secs
250956	7.7941950e-03	6.8813523e-04	115.8 secs

Table 2: case of $r = 0.5$

5 Conclusions

Our experience on some real network evaluations (from the reliability point of view) learns us that exact algorithms are too expensive in time to be applied to them. This is the case, for instance, for networks with hundreds of links once the obvious reductions (as series-parallel simplifications) have been performed. When exact algorithms fail or when their computational

time is prohibitive, the Monte Carlo method can supply a probabilistic answer in a reasonable time. In particular, the implementation of this technique on a multiprocessor machine as reported in this paper has allowed us to handle with very large networks in a satisfactory way. Further work can be performed in several directions. From one side one can attempt to still reduce the computational time by trying, for example, other Monte Carlo techniques or by looking for other algorithmic approaches in the exploration of the stochastic graph. Furthermore, it is of interest to investigate the implementation of exact techniques as factoring algorithms [2], [11] in a parallel computer such as the iPSC2. Even if almost surely it cannot handle networks with hundreds of elements, the utilization of such a machine must improve the bounds on the size of the graphs that can be exactly solved in "reasonable" computational time on sequential computers.

References

[1] A.Agrawal and R.E.Barlow. A survey of network reliability and domination theory. *Op. Res.*, 32:478–492, 1984.

[2] A.Satyarayana and M.K.Chang. Network reliability and the factoring theorem. *Networks*, 13, 1983.

[3] D.Knuth. *The Art of Computer Programming.* Volume 2 : Seminumerical Algorithms, Addison-Wesley, 1969.

[4] D.L.Eager, J.Zahorjan, and E.D.Lazowska. Speed-up versus efficiency in parallel systems. *IEEE Trans. on Comp.*, 38(3):408–423, March 1989.

[5] G.S.Fishman. A comparison of four monte-carlo methods for estimating the probability of s-t conectedness. *IEEE Trans. Reliab.*, R-35(2), 1986.

[6] INTEL. *iPSC/2 Users Guide, Intel Scientific Computers.* Beaverton, 1988.

[7] INTEL. *Product Realease Notes, Realease 2.2, iPSC/2 System Software.* Beaverton, 1988.

[8] J.M.Hammerslay and D.C.Handscomb. *Monte Carlo Methods.* Halsted Press, Wiley & Sons. Inc., New York, 1979.

[9] J.S.Provan. The complexity of reliability computations in planar and acyclic graphs. *SIAM J. Comput.*, 15(3), Aug. 1986.

[10] M.El Khadiri and G.Rubino. *Efficient Monte Carlo Evaluation of Network Reliability Measures.* Technical Report, INRIA, Campus de Beaulieu, 35042 Rennes, France, 1990 (to appear).

[11] K.Wood. A factoring algorithm using polygon-to-chain reductions for computing k-terminal network reliability. *Networks*, 15:173–190, 1985.

[12] M.O.Ball. Computational complexity of network reliability analysis: an overview. *IEEE Trans. Reliab.*, R-35(3), 1986.

[13] M.O.Locks and A.Satyarayana editors. Network reliability – the state of the art. *IEEE Trans. Reliab.*, R-35(3), 1986.

[14] R.Marie and G.Rubino. Direct approaches to the 2-terminal reliabiity problem. In E.Orhun E.Gelembe and E.Başar, editors, *The Third International Symposium on Computer and Information Sciences*, pages 740–747, Ege University, Çeşme, Izmir, Turkey, 1988.

[15] R.M.Karp and M.Luby. Monte-carlo algorithms for the planar multiterminal network reliability problem. *Journal of Complexity*, 1:45–64, 1985.

[16] S.F.Nugent. The ipsc/2 direct-connect communications technology. In Geoffrey Fox, editor, *The Third Conference On Hypercube Concurrent Computers and Applications*, pages 51–60, California Institute of Technology, 1988.

[17] Y.Won S.Ranka and S.Sahni. Programming a hypercube multicomputer. *IEEE Software*, 69–77, September 1988.

[18] Y.Saad and M.H.Shultz. Topological properties of hypercubes. *Yale Reseach Report*, June 1985.

A Annex : on the speed-up function

Let us derive here a simple model to give an idea on the behaviour of the speed-up function as the number of processors increases. In the sequel, we will implicitely consider average times.

Let us denote by $T(p)$ the time needed by a configuration of p processors to perform a fixed number N of replications on a given graph with our algorithm. Let us denote by T_{prp} the preprocessing time that can be considered independent of p (preparation of the data structures and of the pseudo-random number generators, etc.). After the preprocessing, the control process asynchronously sends the data iteratively to the simulation processes. The p packets of messages are not received at the same instant. We denote by T_0 the delay between the end of the preprocessing phase and the instant at which the first simulation process is ready to start its replications. Then, we write

$$T(p) = T_{prp} + T_0 + T_{1,p} + T_{2,p} + \cdots + T_{p,p}.$$

Here, $T_{j,p}$ is the length of the interval in which there are exactly j simulation processes performing replications. This interval starts when the jth process receives the data and it ends when the data arrives to the next one (for $j < p$). When the last simulation process begins its replications, the whole p processors work in parallel for the rest of the replications (this last interval has length $T_{p,p}$). From this it follows that

$$T_{j,p} = T_{j,p+1} \quad \text{for } j = 1, 2, \ldots, p-1. \tag{1}$$

Experimental results show that the speed of the simulation process running on node 0 is not significantly affected by the control process runing in the same node. If we denote by T the time needed by one single processor to achieve N replications (excluding the preprocessing time) then its speed is $v = N/T$ and we can write

$$vT_{1,p} + 2vT_{2,p} + \cdots + pvT_{p,p} = N$$

which gives

$$T_{p,p} = \frac{T}{p} - \frac{T_{1,p} + 2T_{2,p} + \cdots + (p-1)T_{p-1,p}}{p}.$$

Then, we have

$$T(p) = T_{prp} + T_0 + \Delta(p) + \frac{T}{p}$$

where

$$\Delta(p) = T_{1,p}\left(1 - \frac{1}{p}\right) + T_{2,p}\left(1 - \frac{2}{p}\right) + \cdots + T_{p-1,p}\left(1 - \frac{p-1}{p}\right).$$

Proposition A.1 *The function $p \mapsto \Delta(p)$ increases with p.*

Proof.

$$\Delta(p) = \sum_{j=1}^{p-1} T_{j,p}\left(1 - \frac{j}{p}\right)$$

and

$$\begin{aligned} \Delta(p+1) &= \sum_{j=1}^{p} T_{j,p+1}\left(1 - \frac{j}{p+1}\right) \\ &\doteq \sum_{j=1}^{p-1} T_{j,p}\left(1 - \frac{j}{p+1}\right) + T_{p,p+1}\left(1 - \frac{p}{p+1}\right) \quad \text{(from (1))}. \end{aligned}$$

So,

$$\Delta(p+1) - \Delta(p) = \frac{1}{p(p+1)} \sum_{j=1}^{p-1} jT_{j,p} + \frac{1}{p+1} T_{p,p+1} > 0$$

□

Let us denote by $S(p)$ the speed-up function and by $E(p)$ the efficiency function. We have

$$S(p) \stackrel{\text{def}}{=} \frac{T_{prp} + T}{T(p)} = \frac{T_{prp} + T}{T_0 + T_{prp} + \Delta(p) + \frac{T}{p}}$$

and

$$E(p) \stackrel{\text{def}}{=} \frac{S(p)}{p} = \frac{T_{prp} + T}{p(T_0 + T_{prp} + \Delta(p)) + T}.$$

It is clear that $E(p) < 1$ for all p and that it decreases with p. This gives the last result.

Proposition A.2 *The function $p \mapsto p - S(p)$ increases with p.*

Proof. Since $E(p)$ is decreasing, if $p_1 > p_2$ then

$$-\frac{S(p_1)}{p_1} > -\frac{S(p_2)}{p_2}.$$

Adding 1 to the two sides, we get

$$\frac{p_1 - S(p_1)}{p_1} > \frac{p_2 - S(p_2)}{p_2}.$$

Also, since $E(p) < 1$ for all p we have $p - S(p) > 0$. This allows us to conclude that if $p_1 > p_2$ then $p_1 - S(p_1) > p_2 - S(p_2)$. □

PI 542 **A NEW APPROACH TO VISUAL SERVOING IN ROBOTICS**
Bernard ESPIAU, François CHAUMETTE, Patrick RIVES
Juillet 1990, 44 Pages.

PI 543 **SIMPLE DISTRIBUTED SOLUTIONS TO THE READERS-WRITERS PROBLEM**
Michel RAYNAL
Juillet 1990, 10 Pages.

PI 544 **IMPLEMENTATION AND EVALUATION OF DISTRIBUTED SYNCHRONIZATION ON A DISTRIBUTED MEMORY PARALLEL MACHINE**
André COUVERT, René PEDRONO, Michel RAYNAL
Juillet 1990, 14 Pages.

PI 545 **ESTIMATION OF NETWORK RELIABILITY ON A PARALLEL MACHINE BY MEANS OF A MONTE CARLO TECHNIQUE**
Mohamed EL KHADIRI, Raymond MARIE, Gerardo RUBINO
Août 1990, 20 Pages.

PI 546 **LIMIT THEOREMS FOR MIXING PROCESSES**
Bernard DELYON
Septembre 1990, 22 Pages.

PI 547 **PERFORMANCES DES COMMUNICATIONS SUR LE T-NODE**
Frédéric GUIDEC
Septembre 1990, 38 Pages.

PI 548 **LES PREDICATS COLLECTIFS : UN MOYEN D'EXPRESSION DU CONTROLE DU PARALLELISME *OU* EN PROLOG**
René QUINIOU, Laurent TRILLING
Septembre 1990, 34 Pages.

PI 549 **NORMALISATION SOUS HYPOTHESE D'ABSENCE DE LIEN APPLICATION AU CAS NOMINAL**
François DAUDE
Septembre 1990, 42 Pages.

PI 550 **MULTISCALE SIGNAL PROCESSING : FROM QMF TO WAVELETS**
Albert BENVENISTE
Septembre 1990, 28 Pages.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249-6399