Parallel machines for multiset transformation and their programming style

Jean-Pierre Banâtre, Anne Coutant, Daniel Le Métayer

INRIA

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° 759

PARALLEL MACHINES FOR MULTISET TRANSFORMATION AND THEIR PROGRAMMING STYLE

Jean-Pierre BANATRE
Anne COUTANT
Daniel LE METAYER

NOVEMBRE 1987

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042-RENNES CÉDEX
FRANCE
Téléphone : 99 36 20 00
Télex: UNIRISA 950 473 F
Télécopie: 99 38 38 32

PARALLEL MACHINES FOR MULTISET TRANSFORMATION AND THEIR PROGRAMMING STYLE

Jean-Pierre Banâtre (*)

Anne Coutant (**)

Daniel Le Métayer (*)

(*) IRISA/INRIA campus de Beaulieu 35042 RENNES CEDEX

(**) IRISA/CNRS campus de Beaulieu 35042 RENNES CEDEX

Publication Interne n° 378 - Novembre 87 - 26 pages.

Abstract

One of the most challenging problems in the field of computing science today concerns the development of software for more and more powerful parallel machines. In order to tackle this issue we present a new paradigm for parallel processing; this model, called Γ, is based on the chemical reaction metaphor: the only data structure is the multiset and the computation can be seen as a succession of chemical reactions consuming elements of the multiset and producing new elements according to specific rules. We detail some programming examples showing the power of the model. Furthermore, due to its lack of imperative features, this language can be very naturally implemented in a distributed way. We describe two parallel machines supporting the execution of Γ-programs. These machines differ essentially in their communication schemes: the first one is synchronous and the second one asynchronous. So we advocate the separation of the design of programs for massively parallel machines into two steps which can be verified in a formal way: the construction of a program with implicit parallelism (Γ-program) and its translation into a network of processes.

Keywords and phrases : parallelism, descriptive programming, parallel evaluation, parallel machines.

DEUX MACHINES PARALLELES POUR LA TRANSFORMATION DE MULTIENSEMBLES ET LEUR STYLE DE PROGRAMMATION

Résumé.

L'une des gageures actuelles dans le domaine de l'informatique concerne le développement de logiciels destinés à des machines parallèles de plus en plus efficaces. Dans le but d'aborder ce probème, nous présentons un paradigme de calcul parallèle; ce modèle appelé Γ est basé sur la métaphore de la réaction chimique : le multiensemble est la seule structure de données, et l'évaluation peut être vue comme une succession de réactions chimiques consommant des éléments du multiensemble et produisant de nouveaux éléments selon des règles bien spécifiques. Nous détaillons quelques exemples de programmation mettant en avant la puissance du modèle. Par ailleurs, ce langage peut, de par l'absence de caractéristiques impératives, conduire à une mise en oeuvre distribuée. Nous décrivons deux machines parallèles permettant l'exécution de programmes Γ. Ces deux machines se distinguent dans leurs schémas de communication, l'une étant synchrone et l'autre asynchrone. Finalement, nous préconisons la séparation de la tâche de conception de programmes pour machines massivement parallèles en deux étapes - chacune pouvant être prouvée formellement : la construction d'un programme avec parallélisme implicite (programme Γ) puis la transformation en un réseau de processus.

Mots-clés.

parallélisme, programmation descriptive, évaluation parallèle, machines parallèles.

1. Introduction.

Non conventional models of computing such as cellular automata [CODD 68], neuronic models and systolic models [KUNG 82] are now investigated as means of describing highly parallel computations for a wide spectrum of applications such as image processing, numerical computations, speech understanding ...

At the same time, machines with a significant number of processing elements begin to appear [HILLIS 85] and people may think that computing is at the dawn of the era of massively parallel machines for the masses. In fact, we are still far from this situation because we do not know how to master these new "monsters". Let us quote P.J. Denning [DENNING 86]:

"We do not know how to program these new (massively parallel) machines.... The software barrier - the set of limitations that impedes our effective use of parallel hardware technology - is not a brick wall that can be broken down with a heavy ram. It is a deep, thick jungle through which slow progress will be achieved by constant chopping and hacking."

So it is clear that changes in programming languages are needed...in particular it should be possible to dynamically create computations, to synchronize them and to allow information exchange between them. Several languages have been proposed with this aim: in particular C.A.R. Hoare proposed a model called CSP (Communicating Sequential Processes) [HOARE 78] allowing the definition, activation and synchronization of communicating processes. However, due to the lack of an appropriate programming methodology, it appears that it is far more difficult to build concurrent programs than usual sequential programs (although there are still important advances to be made in this area as well). This is mainly due to the fact that the programmer has to mentally manage several threads of control instead of one, as usual.

Our view is that this traditional approach to parallel programming is too imperative to become a general model for concurrent programming. We believe that only very high level languages which make parallelism implicit will allow the description of massively parallel applications.The model which is proposed in this paper is based on the chemical reaction metaphor: the computation is a succession of applications of rules which consume elements of a multiset while producing new ones and inserting them in the initial multiset. The computation terminates when no rule can be applied. The application of rules is made in a non-deterministic way and a parallel interpretation of the model is

straightforward. So this model (called the Γ-model) acts as a multiset transformer. The relevance of this model to program construction is shown in [BANATRE 86] and [COUTANT 86] which present a systematic method for Γ-program derivation. This point is not detailed here as we are mainly interested in machine architecture. We describe two parallel machines which support the execution of Γ-programs. They differ essentially in the communication schemes between processors : the first one is synchronous and the second one is asynchronous. So we split the construction of programs for parallel machines into two steps: the design of a program with implicit parallelism (Γ-program) and its translation into a program with explicit parallelism (with processes). The important point is that these two steps can be verified in a formal way.

The Γ-model is quickly presented in section 2. Section 3 is dedicated to the construction of Γ-programs for a path-finding problem, the Minimax algorithm and the cycle detection problem. Section 4 describes the structure of the Γ-machines and gives some experimental results. The conclusion discusses related work and suggests issues for further research.

2. The Γ-model.

The Γ-model can be described as a multiset transformer: the computation is a succession of applications of rules which consume elements of the multiset while producing new elements. The computation ends when no rule can be applied. The application of rules is performed in a non-deterministic way and a parallel interpretation of this computational model is straightforward.

Data structures.

The basic information structuring facility is the multiset which is the same as a set except that it may contain multiple occurrences of the same element. Atomic components of multisets may be of type real, character, integer, tuple of type ..., multiset of type ...

Γ-operator.

The main feature of the model is the Γ-operator which can be defined in the following way:

$\Gamma(R,A)\ (M) =$

if $\exists\ x_1, ..., x_n \in M$ **such that** $R\ (x_1, ..., x_n)$ **then**

$\Gamma(R,A)\ (\ (M - \{x_1, ..., x_n\}) \cup A\ (x_1, ..., x_n)\ )$

else M.

Operator R is called the "reaction condition"; it is a boolean function indicating in which case some

elements of the multiset can react. The A function ("action") describes the result of this reaction. Let us point out that if the reaction condition holds for several subsets at the same time, the choice which is made among them is not deterministic; if these subsets are disjoint the reactions can even take place at the same time. However appropriate restrictions on the definition of condition R and action A may ensure determinacy; this point is not developed here.

This is not the most general definition of Γ; actually the Γ operator can take any number of couples (Reaction condition, Action), each reaction condition indicating in which case the associated action can be applied; however most common programs can be expressed with only one couple (Reaction condition, Action), so we will restrict our discussion to this particular case.

Examples of Γ-programs.

Let us now take two examples to illustrate the programming style entailed by the Γ model.

Example1.

The sieve of Eratosthenes can be written in a concise and elegant way using Γ:

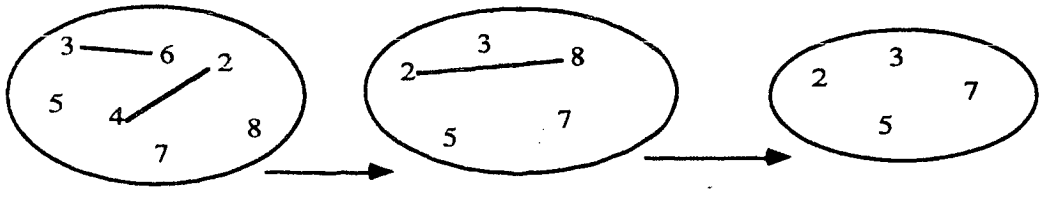
sieve (n) = **Γ(R,A) ({2, ..., n})** where

R (X1, X2) = **multiple (X1, X2)**

A (X1, X2) = **{X2}**

where multiple (X1, X2) is true if and only if X1 is a multiple of X2.

The following figure describes the computation of sieve (8). Of course this is one among the possible paths leading to the stable state.



Example 2

Let us consider a sorting program. The data structure is a multiset M of couples (index, value) where the index X.i of an element X gives the position of the value X.v in a virtual sequence of values to be sorted. All indexes are different and in a range [1, ..., n] where n = Card(M). The Γ-program is the

following :

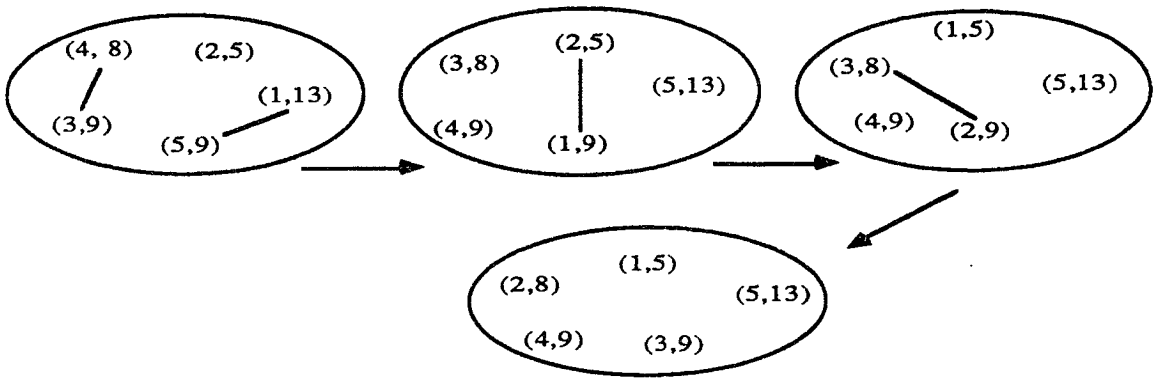
sort (M) = Γ (R,A) (M) where

R (X, Y) = (X.i > Y.i and X.v < Y.v)

A (X, Y) = { (Y.i, X.v) , (X.i, Y.v) }

The following figure describes the computation of:

sort ({ (4, 8) , (2, 5) , (3, 9) , (1, 13) , (5, 9) }) :



The stable state is reached as no couple of elements verifies the reaction condition. Of course this is only one of the possible sequences of configurations leading to the result.

This program can be seen as a generalized form of the exchange sort algorithm as any couple of ill-ordered elements can exchange their positions at any time.

3. Three Programming examples.

We consider three classical applications : a minimum-path-finding program, the well-known minimax algorithm and the problem of cycle detection in graphs.

3.1 Path-finding example.

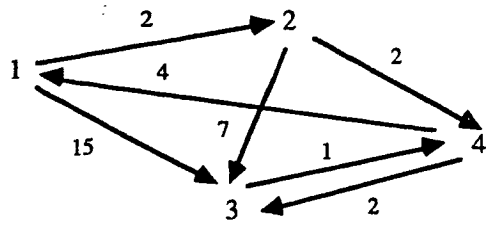
Let us consider a directed graph in which a nonnegative cost is associated with each edge. The purpose of the algorithm we describe here is to find the shortest path from a particular vertex r of the graph (called the root) to each vertex v. The shortest path is defined to be the path with lowest cost, and the cost of a path is the sum of the costs of its edges.

The graph may be represented by a multiset of tuples (i, j, c, l) where

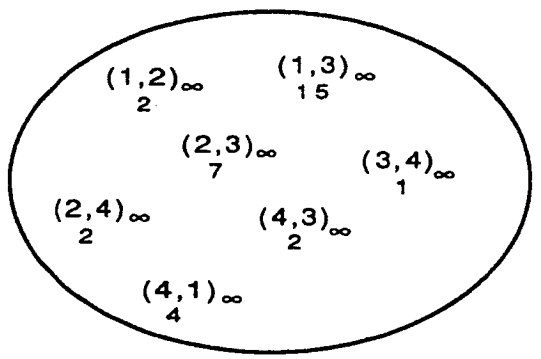
- i and j are vertex identifiers such that (i, j) is an edge of the graph,
- c is the cost of the edge (i, j),
- l is the length of the shortest known path from the root r to vertex j via vertex i.

In the following diagrams, we represent these tuples as $(i, j)_{\substack{l \\ c}}$

Considering the directed graph :



the initial multiset M_0 representing this graph is :



The following program computes the shortest path rαij, where α denotes a sequence of vertices, for each edge (i,j) of the graph :

Γ(R1,A1) (M0 ∪ {(r,r,0,0)}) where

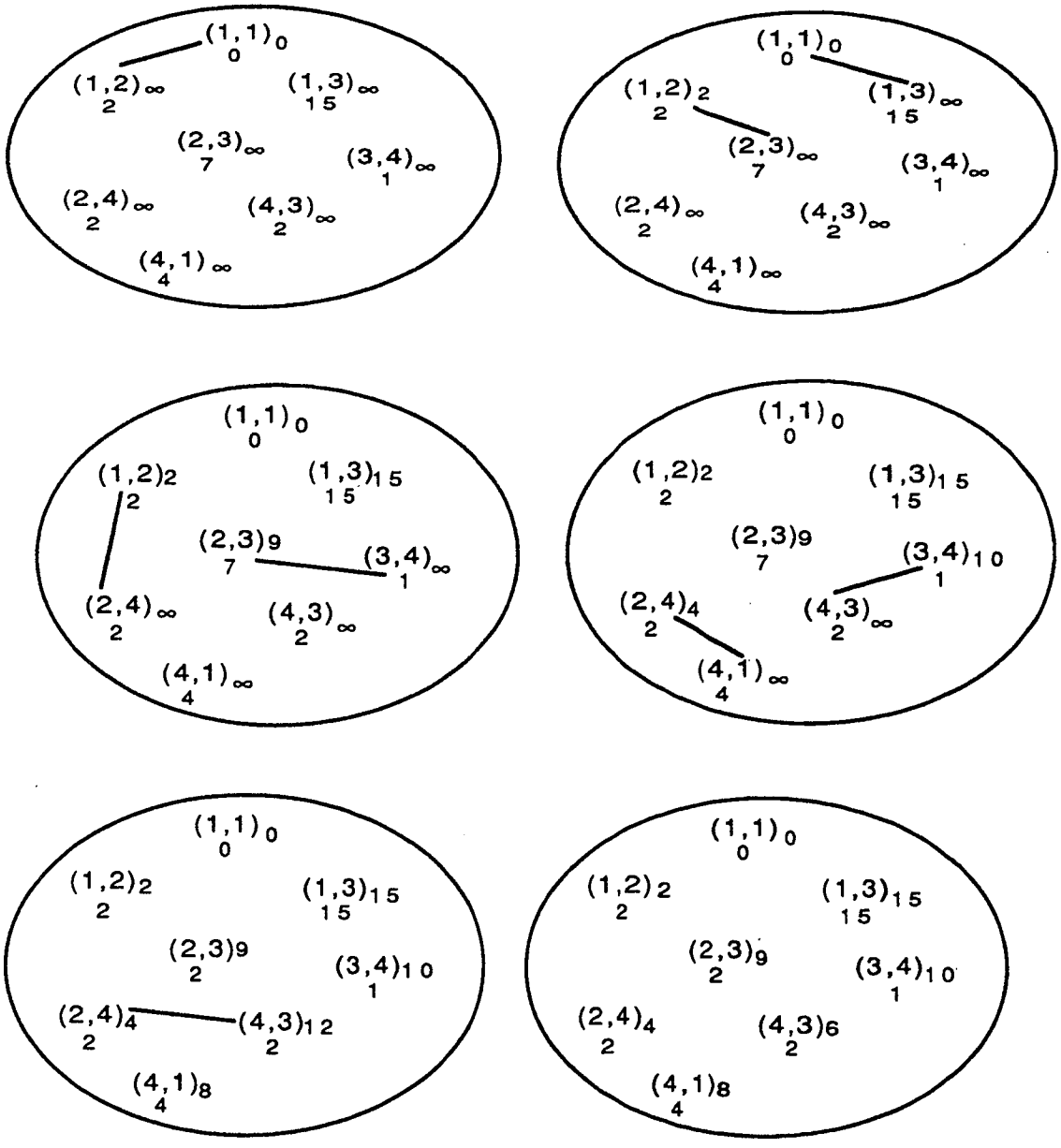
R1(X,Y) = (X.j = Y.i) and (X.l + Y.c < Y.l)

A1(X,Y) = { X, (Y.i, Y.j, Y.c, X.l + Y.c) }

A tuple $(r,r)_{\substack{0 \\ 0}}$ is added to M_0 to represent the root and to ensure the start of computation.

Two tuples $(a,b)_{\substack{l1 \\ c1}}$ and $(b,c)_{\substack{l2 \\ c2}}$ may interact if the path rαabc of length l1 + c2 is shorter than the path

rβbc of length l2. The action updates the length l2 of the shortest known path from the root to vertex c. Let us describe a possible evolution of the multiset M = $M_0 \cup \{(1,1,0,0)\}$ during the evaluation of Γ(R1,A1) (M).



The final multiset M1 may contain tuples $(i,a)_{li}^{ci}$, $(j,a)_{lj}^{cj}$ where $i \neq j$. In order to define the shortest path,it is necessary to keep only the tuple $(k,a)_{lk}^{ck}$ with lk = Min lj s. t. $(j,a)_{lj}^{cj} \in$ M1.

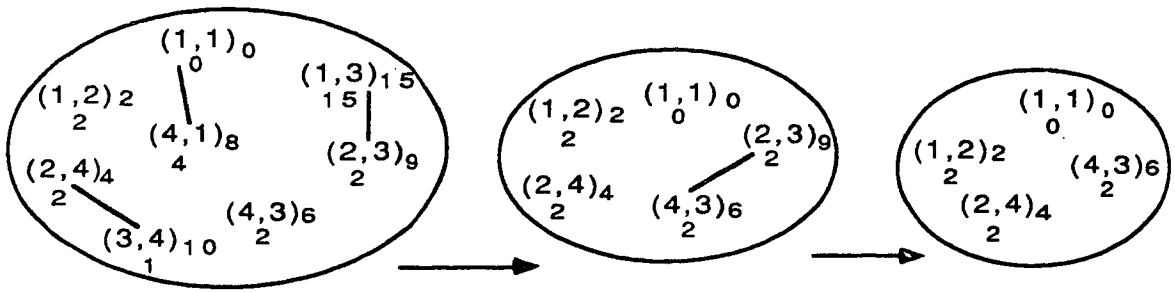
The following Γ-program performs the appropriate computation :

$\Gamma(R_2,A_2)$ (M) where

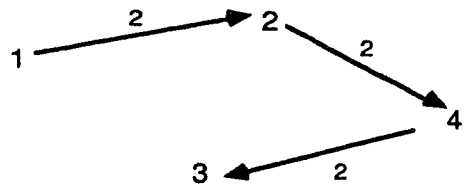
R_2 (X,Y) = (X.j = Y.j) and (X.l ≥ Y.l)

A_2 (X,Y) = {Y}

The figure below gives a possible evolution of M1 during the evaluation of $\Gamma(R_2,A_2)(M1)$.



The final state of M represents the tree of the shortest paths from vertex 1 to vertices 2,3 and 4.



We have defined a program which computes the tree of the shortest paths (if any) from a given root r to the vertices in a graph. This program is a composition of two Γ-programs :

shortest_paths (r, M_0) = $\Gamma(R_2,A_2)$ ($\Gamma(R_1,A_1)$ ($M_0 \cup$ { (r,r,0,0)}))

R_1(X,Y) = (X.j = Y.i) and (X.l + Y.c < Y.l)

A_1(X,Y) = { X, (Y.i, Y.j, Y.c, X.l + Y.c) }

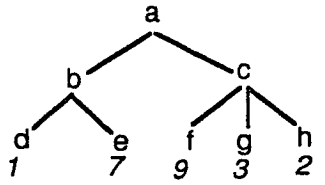
R_2 (X,Y) = (X.j = Y.j) and (X.l ≥ Y.l)

A_2 (X,Y) = {Y}

3.2 Minimax example.

Minimax is a technique for searching game-trees, in order to determine the best move in a given position of a game with two adversaries [WINSTON 84]. The nodes in a game-tree represent board configurations, and they are linked by branches representing the possible moves between them. The algorithm starts with a partially developed lookahead game-tree in which the leaves are associated with

advantage-specifying values. Suppose that positive numbers indicate favour to one player, called the maximizing player (the player wanting to maximize the score), and negative numbers indicate favour to the other, the minimizing player. The maximizing player is looking for a path leading to the largest positive score assuming that his opponent will always make the best choice (thus minimizing the score). So the value of a node is evaluated by taking the minimum or the maximum of the values of its sons, according to the level of node. Working up from the leaves, the values of the possible moves in the initial state are computed, and the best move is finally determined. Let us take an example of a game-tree :



Maximizing level
Minimizing level

In this case, b should get the value 1 = min (1,7) and c the value 2 = min (9,3,2). Finally a gets the value 2 and c represents the best move for the player.

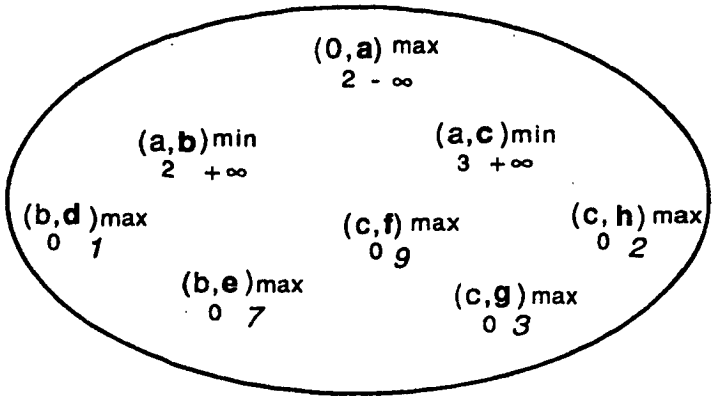
Let us now express the MINIMAX algorithm in the Γ-formalism. The data structure must represent the nodes of the tree, the relationships between them, their level and their value. These requirements lead us to define a multiset of tuples (ident, father, level, nsons, cval) where

- ident is the identifier of the node,
- father is the identifier of the father of the node; so (father, ident) is a branch of the tree,
- level is the function max or min corresponding to the level of the node,
- nsons is the number of sons of the node,
- cval is the current value of the node.

We represent such a tuple in a graphic way by : (father, **ident**) level

nsons *cval*

Let M be the multiset corresponding to the previous game-tree, its representation is the following:



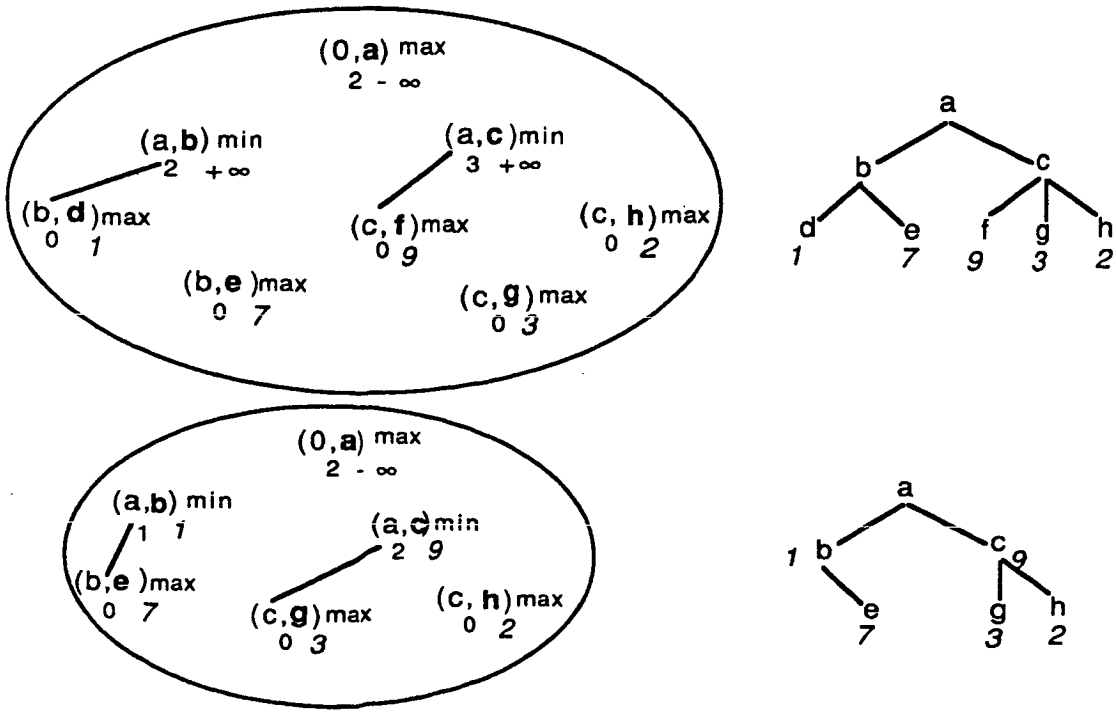
MINIMAX can be defined in Γ by :

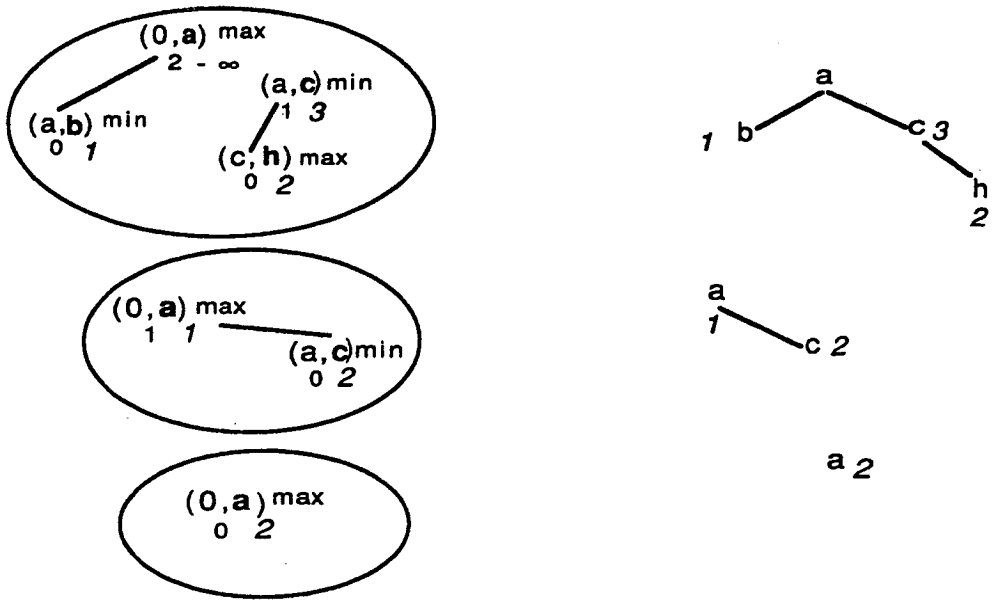
Minimax (M) = Γ (R,A) (M) , where

R(X,Y) = (X.ident = Y.father) and (Y.nsons = 0)

A(X,Y)={(X.ident,X.father,X.level,X.nsons -1,X.level(X.cval,Y.cval))}

Two tuples X and Y may interact if the node represented by X is the father of the node represented by Y, and the node represented by Y is a leaf of the current tree. The action removes the tuple Y and updates the tuple X : X has one child less as Y is removed from the tree, and the value of X is replaced by the maximum or the minimum of the values of the nodes. The following figure describes one possible evolution of the multiset during the evaluation of Minimax (M).



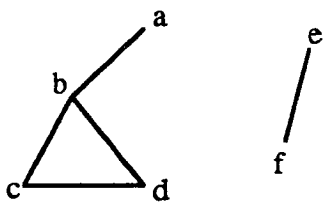


So the value we are looking for, in this case the value 2, is the cval field of the unique element of the result.

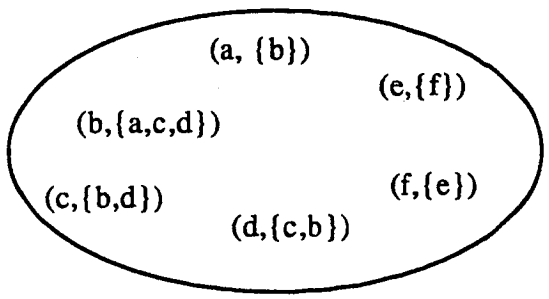
3.3 Cycle detection in a non-directed graph.

The purpose of this algorithm is to detect cycles in a non-directed graph. The graph is represented by a multiset E of couples (e,V) where e is a vertex and V the set of neighbours of this vertex.

Consider the following graph :



Its representation is the following :



In order to find the cycles, we use the following property : if a graph is such that every vertex

has at least two neighbours, then it contains only cycles. The program will proceed by detecting vertices isolated or with only one neighbour and eliminating them. Vertices which are not eliminated at the end of the computation belong to a cycle.

The program can be expressed as :

$$\text{cycles (E)} = \Gamma((R_1,A_1), (R_2,A_2))\ (E_0) \qquad \text{where}$$

$$R_1((e_1,V_1),(e_2,V_2)) = (V_2 = \{e_1\})$$

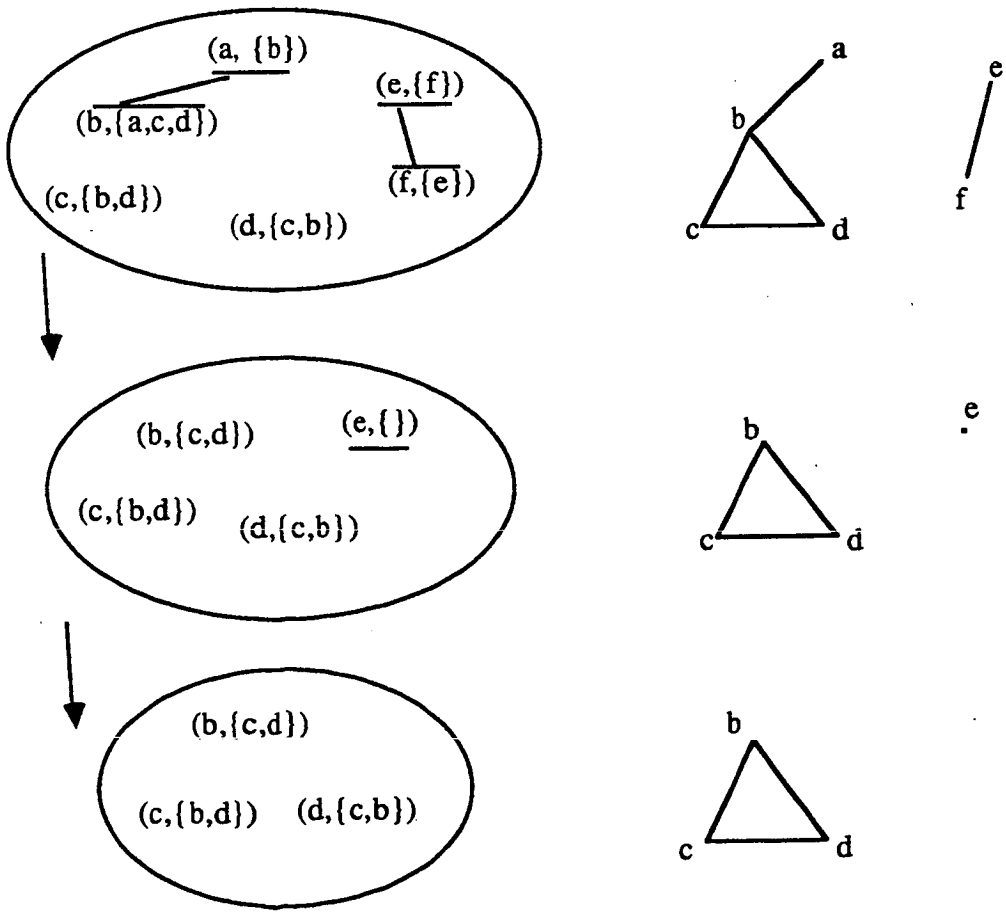
$$A_1((e_1,V_1),(e_2,V_2)) = \{(e_1, V_1 - \{e_2\})\}$$

$$R_2((e,V)) = (\text{Card}(V) = 0)$$

$$A_2((e,V)) = \{\}$$

R_1 detects a vertex e_2 with only one neighbour ($V_2 = \{e_1\}$) and A_1 eliminates it from the multiset; R_2 detects vertices without any neighbour and eliminates them.

Let us describe a possible execution of $\Gamma((R_1,A_1), (R_2,A_2))$ when applied to the multiset M represented above :



4. Two parallel machines implementing the Γ-model.

The evaluation of Γ-programs involves two different kinds of tasks :

(1) the search of the elements of the multiset verifying the reaction condition,

(2) the application of the action to these elements.

The action application is clearly a local operation : it can be carried out independently of other actions on the rest of the multiset. So the main problem to tackle while designing a parallel implementation is the distribution of the search process. For the sake of simplicity, we assume that we have a network of processes and that they are as many processors as there are values in the multiset. The general case can be handled in the very same way by considering sets of values instead of values.

The first choice concerning the implementation of the search process concerns the move of values; as all couples of values have to be realized, we have two solutions : either we consider an interconnection network (and the values do not have to move through the network), or we assume that values move through the network. In the first case, the examination of all pairs will be achieved by a communication protocol insuring that any processor will communicate with any other processor in a finite amount of time. In the second case, the realization of all couples will depend on the movement protocol indicating the direction in which values have to move. A second implementation choice concerns the control of the system; this control can be centralized : one processor has an overall view of the network and is responsible for the communication protocol and termination. When the control is ditributed, no processor has a general view of the system and each particular processing element must take decisions concerning communication and termination. This second solution may present a higher level of parallelism, as the synchronization is less constraining. However, problems such as absence of deadlock, or termination detection are more difficult to tackle and may entail a loss of efficiency. We explain now in detail two solutions for the parallel implementation of Γ : the first one is centralized and based on an interconnection network, and the second one is distributed and relies on a chain architecture.

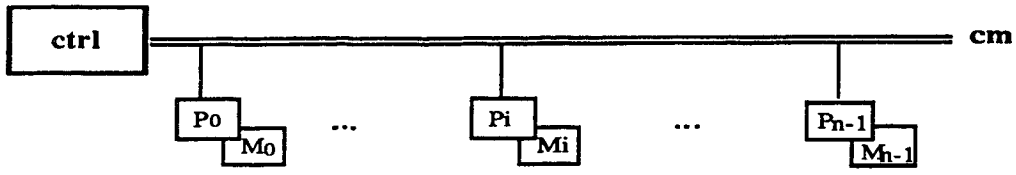
4.1. A synchronous Γ-machine.

We describe the general organization of a synchronous Γ-machine built on an interconnection

network with centralized control, and then give an example of execution.

4.1.1 General organisation.

The overall organization of this machine can be figured out as follows :

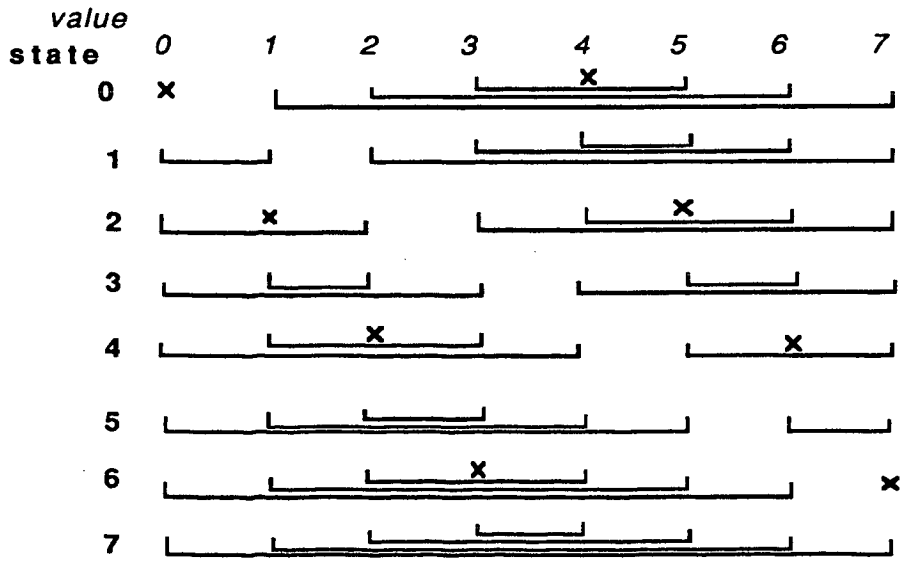


where : Pi, Mi represent processors with their local memory,

ctrl is the controller which issues commands to Pi's via a communication medium cm (an interconnection network). At each step, processor ctrl spreads a signal over the network to trigger the communications.

From this figure, it is clear that there is a logical connection between every couple of processors.

In order to form all possible pairs and thus try all possible reactions, it is possible to design a n-step iterative algorithm as follows : to each processor k from 0 to n-1 is attached a variable i_k representing the identity of the processor with which processor k has to communicate in order to evaluate the reaction condition. The values i_k are initialized to (n-k) mod n. At the end of each step, the value of each variable i_k is modified in the following way : $i_k := (i_k + 1)$ mod n, thus producing a new configuration, and a new set of pairs. The figure below presents the set of configurations obtained for a network of 8 processors. Uncoupled values are represented by a x-symbol.



In the first state, the processor ctrl triggers (by sending a signal) the communications between processors 1 and 7, 2 and 6, 3 and 5; processors 0 and 4 remain inactive. When a processor has finished its computation, it sends a signal to the controller. So when the controller has received n signals, the current step is over and a new signal is spread over the network (if the termination state is not yet achieved).

One can notice that any sequence of n steps generates all possible configurations starting from any step. This property is exploited to detect termination : the system can stop whenever n-1 successive configurations have been tried without any reaction. In order to check this property, a variable t is attached to the controller. This variable is initialized to 0 and is incremented by one at each step without reaction. If a reaction occurs, t is reset to 0; this means that the controller is aware of any reaction, so when a reaction occurs on a processor Pi, the processor Pi sends a signal to the controller.

Let us now describe the behaviour of the processes in a formal way. To this aim we use the language CSP ([HOARE 78]). We just recall the most important constructs of the language :

P ? V	where P is a process name and V a variable name, is the input of a value from P. After execution, V denotes the imported value.
P ! E	where P is a process name and E an expression, is the output of the value of E to the process P.
$[GC_0 \mid\mid ... \mid\mid GC_{n-1}]$	where GC_0, ..., GC_{n-1} are Guarded Commands, is an alternative command : it executes one arbitrarily selected executable guarded command.
$*[GC_0 \mid\mid ... \mid\mid GC_{n-1}]$	where GC_0, ..., GC_{n-1} are Guarded Commands, is a repetitive command : it executes one executable guarded command (arbitrarily selected) until all guards fail.
[||i (i = 0,n-1) GCi]	where GC_0, ..., GC_{n-1} are Guarded Commands, is a shorthand notation for [GC0 || ... || GCn-1]
Pi (i = 0,n-1) !! E	is a broadcast of expression E to processes P0, ..., Pn-1.

ctrl denotes the controller process and Pi (i=0,n-1) the calculus processes.

The following signals are sent or received by the controller :

S1 signal sent to processors Pi to trigger the communications.

S2 signal sent by a processor Pi to ctrl after occurence of a reaction.

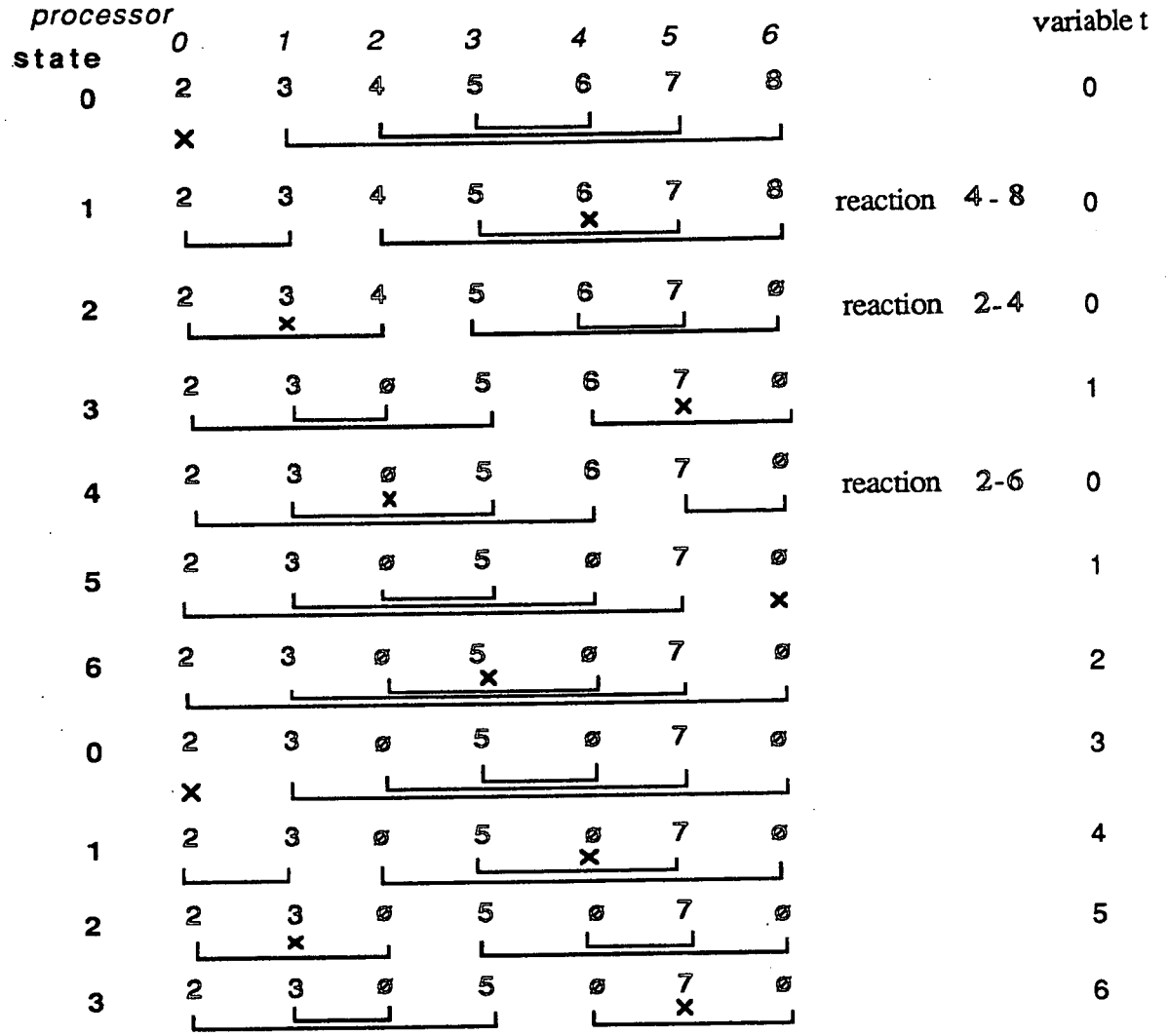
S3 " " " " " at the end of a computation without reaction.

```
begin
  ctrl ::  begin
             signal  S1, S2, S3 ;
             var signal  S;
             var int t init 0;
             *[ Pi (i = 0,n-1) !! S1  →  (bool m init false;
                                          *[||i (i = 0,n-1)  Pi ? S  →  [S = S2 → m := true
                                                                         ||
                                                                         S = S3 → skip ]
                                          ];
                                          [ m → t := 0  ||  ¬m →  t := t+1]
                                          );
                                          [ t = n-1 →  exit  ||  t < n-1 →  skip ]
             ];
           end
  Pi ::    begin
             var typeV Vi,Vi_k;
             var int i_k init (n-i) mod n;
             signal S1,S2,S3;
             *[ ctrl ? S1    →      [i_k < i  → [ P(i_k) ! Vi  →  P(i_k) ? Vi;
                                                                  ctrl ! S3 ]
                                    ||
                                    i_k > i  → [ P(i_k) ? Vi_k  →
                                                  [R(Vi,Vi_k)  → (Vi,Vi_k) := A(Vi,Vi_k);
                                                                 P(i_k) ! Vi_k;
                                                                 ctrl ! S2
                                                  ||
                                                  R(Vi_k,Vi)  → (Vi,Vi_k) := A(Vi_k,Vi);
                                                                 P(i_k) ! Vi_k;
                                                                 ctrl ! S2
                                                  ||
                                                  ¬R(Vi,Vi_k) ∧ ¬R(Vi_k,Vi) → P(i_k) ! Vi_k;
                                                                               ctrl ! S3
                                                  ]
                                          ]
                                    ||
                                    i_k = i     →  ctrl ! S3
                                    ];
                                    i_k := (i_k + 1) mod n
             ]
           end
```

end

4.1.2. Example of execution.

The following figure describes a possible evaluation of sieve : 8. The initial multiset is {2, ..., 8}. The values are represented by relief numbers; the empty value is ø.



On this particular example, a good improvement would be to reset t to zero only if a new value is created (which may interact with others). As the value ø can be considered as neutral, it will never react with other values. So, the evaluation of sieve : 8 could be done in only 7 steps.

4.2. An asynchronous Γ-machine.

The proposed solution consists in spreading the values of the multiset over a vector of processors. There is no shared memory in the system and a processor only knows its two neighbours. Each value V is associated with two integer indicators N1 and N2 and a boolean direction D (True means forwards and False means backwards). A value V can only move in the direction D and this

direction is inverted only when the value reaches one end of the network. This ensures that two values must meet after a finite amount of time (actually after at most N-1 exchanges if N is the total number of processors). Indicators N1 and N2 are used to detect that no more reactions can occur in the system. N1 is the number of exchanges undergone by a value and N2 is the number of consecutive exchanges with values V' such that N1'> N-1. It can be proved that if a value possesses an indicator N2 such that N2 ≥ N-1, then no more reactions can occur in the system. Let us now describe in a more formal way the behaviour of a processor i.

We assume that we have N processes P(1), ..., P(N) (one per processor); the algorithm executed by an intermediate process is the following :

local variables :

typeV V :	value.
bool D :	direction of the value.
int N_1, N_2 :	indicators.
int N :	total number of processors.
typeV V' :	working value.
int N_1', N_2' :	working variables.

```
*  [  N2 = N-1                  →  exit
   ||  N2 ≠ N-1 and D           →  P(i+1) ! (V, N1, N2) ;
                                   P(i+1) ? (V, N1, N2) ;
                                   D := F
   ||  N2 ≠ N-1 and ¬D          →  P(i-1) ? (V', N1', N2') ;
                                   [ R(V, V')   →   {V, V'} := A(V, V') ;
                                                    (N1, N2, N1', N2') := (0,0,0,0)
                                   ||
                                     R(V',V)    →   {V, V'} := A(V',V) ;
                                                    (N1, N2, N1', N2') := (0,0,0,0)
                                   ||
                                   ⌉R(V, V') ∧ ⌉R(V',V)  →
                                                (N1, N1') := (N1 + 1, N1' + 1);
                                                [ N1 ≥ N - 1 ∧ N1' ≥ N - 1  →
                                                        (N2, N2') := (N2+1, N2'+1)
                                                || ⌉(N1 ≥ N - 1 ∧ N1' ≥ N-1) →
                                                        (N2, N2') := (0,0)
                                                ]
                                   ]
                                   P(i -1) ! (V, N1, N2) ;
```

$(V, N_1, N_2) := (V', N_1', N_2')$;
$D := T$

]

Processors P_1 and P_N execute the same piece of code except that D is always True for P_1 and always False for P_N.

The basic task executed by a processor consists in communicating with one of its neighbours (according to the direction of its value) and to apply R on their values; if R turns out to be false, the processors exchange their values and update the indicators; otherwise A is applied and the new values (with indicators set to zero) are shared by the two processors. Furthermore we can prove that if any processor with N2 = N-1 stops, there will be no further attempt in the system to communicate with it. So we have a completely distributed termination condition.

The correctness of this distributed implementation has been proved : no deadlock can occur in the system and the system stops if and only if no more reactions can occur. The proof takes advantage of the regularity of the network and the communication scheme. The central property is the fact that in N-1 consecutive exchanges a value is exchanged with N-1 different values.

Example of execution on the Γ-machine.

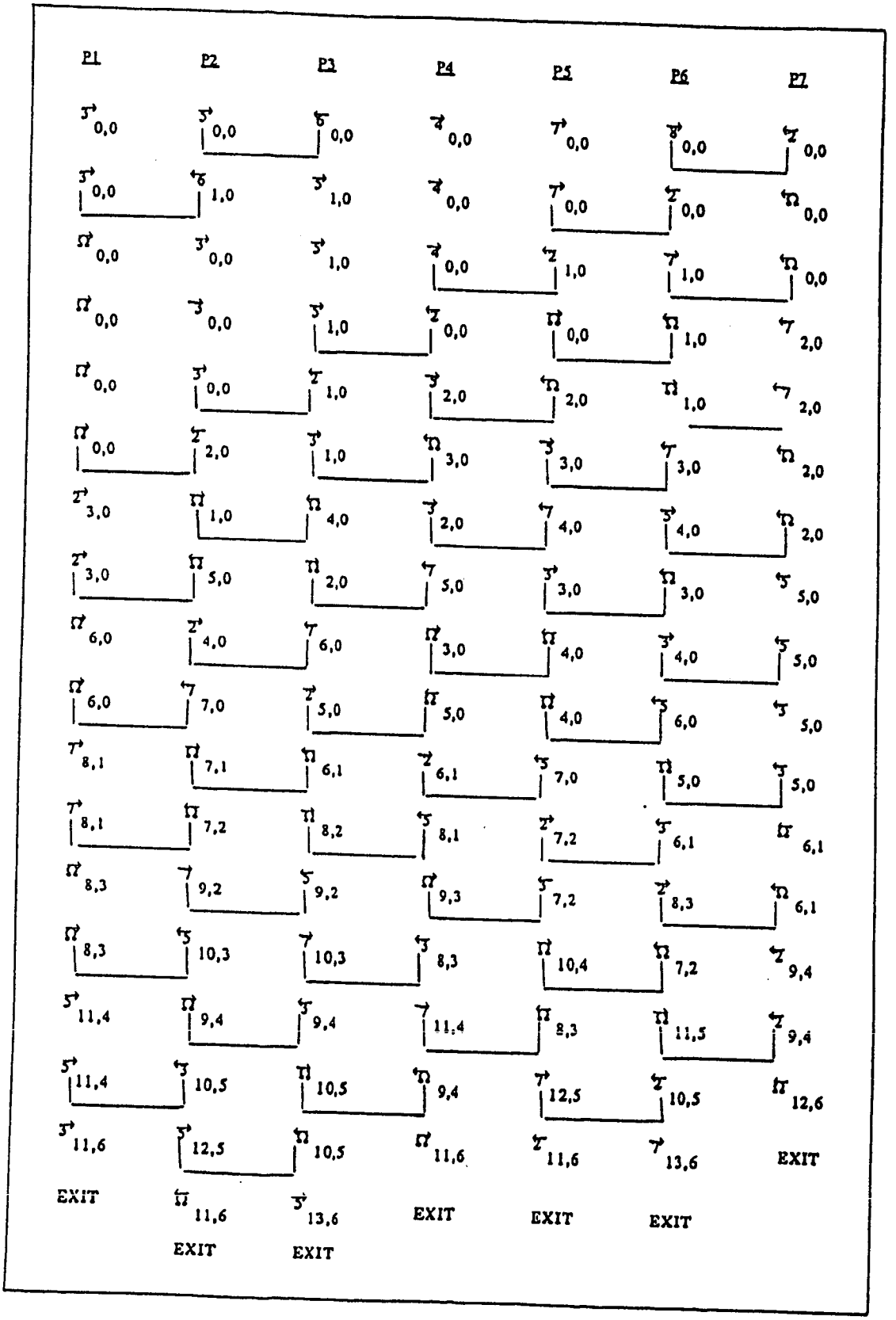
In order to illustrate this algorithm, we describe one possible execution of the sieve example described in section 2. Directions are figured by arrows over the values and indicators N_1 and N_2 are represented in index position. Communicating processors are underlined. The symbol Ω denotes the dummy value (or absence of value): the only particularity of this value is that it cannot react with any other value. In this example a dummy value is generated when a reaction takes place since a reaction produces only one element.

This figure represents snapshots of one among many possible computations of this program. We can however notice a few interesting points about this example :

- the system has a self-balancing property : although we have chosen arbitrary initial directions, the system organized itself so that, after some steps, all inner processors are always involved in a communication. This property is true whenever the tasks executed by each processor take the same time.

- there is no deadlock in the system : this property can be formally proved.

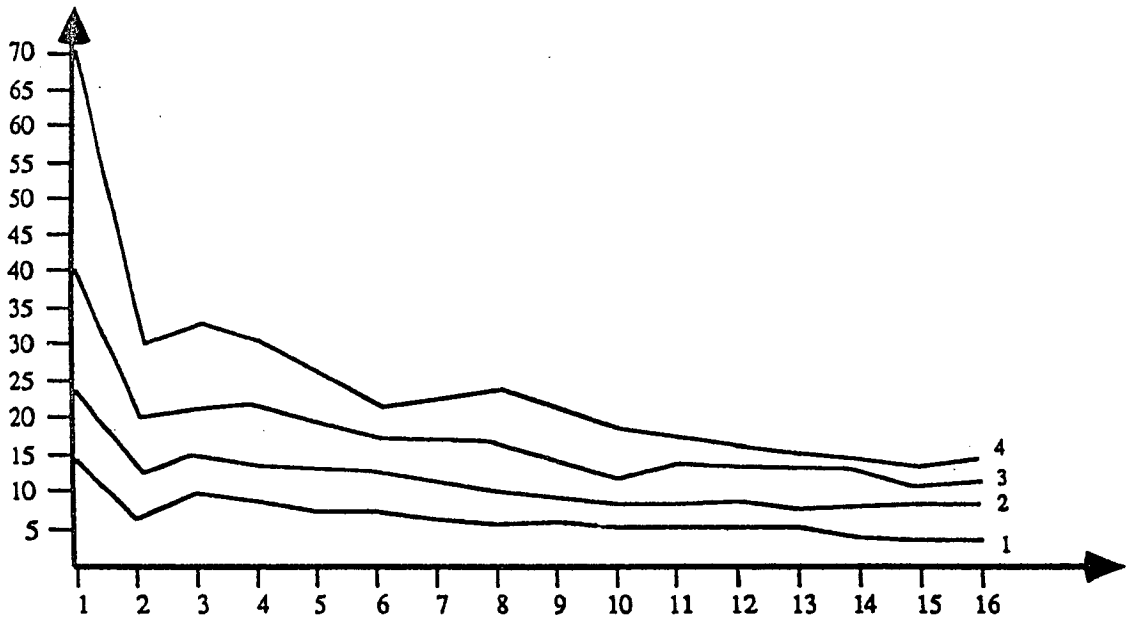
- the termination test works correctly : when a processor stops, no neighbour will attempt to communicate with it.



4.3 Experimental results and discussion.

The second proposal has been implemented on an iPSC hypercube with 16 processors. The machine is configured as a chain of processors and processes described above are associated with node processors. An appropriate interface allowing the observation of the computation has also been developed.

The results obtained are encouraging since the execution time can be divided by a factor ranging from 5 to 12 on examples such as those presented in section 2. The following figures describe the results obtained for the sieve of Eratosthenes : they give the evolution of the execution time with a number of processors ranging from 1 to 16.



1:sieve:30 2:sieve:40 3:sieve:50 4:sieve:60

The results can still be improved significantly by some simple optimizations.

The first proposal has not been experimented on iPSC because this machine is asynchronous and it is obvious that the implementation of synchrony will take too much overhead. In order to fully assess this synchronous Γ-machine, it would be necessary to evaluate it on a synchronous machine such a connection machine or a SIMD machine. Let us notice that termination detection is performed in a much simpler way with the synchronous machine than with the asynchronous one, but the synchronization constraints involve a loss of parallelism.

5. Discussion and issues for further research.

This paper has presented a model of computation called Γ. We have shown the relevance of Γ to different classical applications such as AI problems (Minimax algorithm) and graph problems (path-finding, cycle detection). The descriptive nature of the language makes it suitable for a highly parallel interpretation. Two parallel architectures have been proposed : (i) a synchronous machine assuming that processors change their states every time period and that there is a connection between any couple of processors and (ii) an asynchronous machine where control is fully distributed and values have to move through the network in order to allow the examination of all pairs. An experimentation of this latter machine on iPSC has also been presented.

In the rest of this section, we compare the Γ-model with the model of guarded commands, give some details on the Γ-program development method and present some further research ideas.

5.1 Relationship between the guarded command and the Γ-model.

The reader may wonder about the relationship between our proposal and the Dijkstra guarded command construct. Actually, let us show how guarded commands may be expressed in the Γ-model. An iterative guarded command looks as follows :

$$*[\ C(x)\ \text{----}>\ \Psi(x)\ ]$$

It expresses that the loop will go on applying Ψ to the variable x while C(x) remains true. It can be translated into the following Γ-program:

$$\Gamma(C', \Psi') \quad \text{where}$$
$$C'(y)=C(y)$$
$$\Psi'(y)=\{\Psi\ (y)\}$$

The reaction condition expresses that if the unique element y of the multiset is such that C(y) then

it is replaced by $\Psi(y)$.

More generally, a loop composed of n guarded commands and operating on k variables denoted by the vector $X=(x_1, ..., x_k)$ may be written as:

$$
\begin{aligned}
&*[\ C_1(X) \text{ ---> } \Psi_1(X) \\
&\quad \ldots \\
&\quad C_n(X) \text{ ---> } \Psi_n(X) \\
&\ ]
\end{aligned}
$$

and translated into:

$\Gamma((C'_1, \Psi'_1),\ldots ,(C'_n,\Psi'_n))$ where

$C'i(y)=Ci(y)$

$\Psi'i(y)=\{\Psi i\ (y)\}$

In these examples, the state of computation is represented by the state of the variables. The net effect of executing a statement Ψ_i is to change the values of the variables. Of course, no parallelism is possible as in both cases the argument multiset possesses only one element. There is no simple way of allowing several actions to operate simultaneously on disjoint subsets of the global state. This could only be achieved by **explicitly** programming the constitution of multisets and their modification. Even this solution would imply very important changes in the semantics of the usual guarded commands. Furthermore, the data structuring facilities (arrays) used in conjunction with guarded commands entail a sequential style of programming. It is clear that more descriptive and more general data structures are necessary in order to construct programs with a high potential for parallelism.

5.2 Program development for Γ-machines.

Another interest of the Γ-model is that it can be used as a basis for systematic program derivation.

Our program derivation technique is quite similar to the method used in [DIJKSTRA 76] and [GRIES 81]. The specification S is split up into two parts I and V such that $I \wedge V \Rightarrow S$. I is

an invariant property which should remain true throughout the computation whereas V is the property which should be established at the end of the computation. The application of this technique in [DIJKSTRA 76] and [GRIES 81] allows the development of iterations; as far as we are concerned, we derive Γ-programs.

As an introduction to the method, let us derive the sieve program. Let M be the multiset being transformed during the computation of sieve(n), the initial state of M is the set of all integers in the range 2..n, and the specification S (in first order logic language) of the final state of M may be :

(1) $\forall x \in M \quad x \in [2..n]$

(2) $\forall x \in [2..n] \ (\forall y \in [2..n] \ \neg multiple(x,y) \Rightarrow x \in M)$

(3) $\forall x \, \forall y \in M \ \neg multiple(x,y)$

That is to say : (1) M must contain only elements from the range 2..n ; (2) all prime numbers in the range 2..n belong to M ; and (3) each element of M is a prime number. We choose I = (1) ∧ (2) as the invariant as it is a weakened form of S which is true for the initial state. The computation has to take place as long as the variant V = (3) does not hold; in other words, as long as the formula $\neg V = \exists x \, \exists y \in M$ multiple(x,y) is true. This is expressed by a reaction condition R(x,y) = multiple(x,y). The associated action has to preserve the invariant and to transform the multiset in "the right direction" (so as to reach a state verifying the variant property in a finite number of steps). Let us choose as a termination function f(M) the number of couples (x,y) of elements of M such that multiple(x,y); f(M) is bounded by 0. According to the invariant, the action can (1) neither add a value, (2) nor remove the value y as it may be a prime number; so the only possible action is A(x,y) = {y}. This action maintains the invariant I ; and as a non prime number is eliminated from M, there is at least one couple of multiples less in M, hence f(M) is decreased at least by 1. The derived program is :

sieve(n) = **Γ(R,A) ({2, ..., n})** where

R(x,y) = **multiple(x,y)**

A(x,y) = **{y}**

The interested reader may refer to ([BANATRE 87], [COUTANT 86]) for the description of the systematic method for the derivation of Γ-programs.

5.3 Issues for further research.

The work presented in this paper may be pursued along several lines, let us mention some of them.

Improvement of the evaluation scheme.

Several optimizations may be proposed in order to improve the performance of the Γ-machines presented in section 4. As far as the synchronous Γ-machine is concerned, one may imagine to consider the destruction of a value as an absence of reaction and then the variable t has not to be reset to zero (this situation occurs on the sieve example exhibited in section 4.1). This simple optimization can avoid unnecessary steps which would not involve any reaction.

Concerning the asynchronous machine, an obvious improvement consists in associating two values V_i and V_{i+1} with each processor. The basic cycle of a processor would be to send V_{i+1} to the right neighbour, to wait for a value from the left neighbour and to apply R (and possibly A) to this value and the local value V_i. Then a processor has to send its result to the left neighbour, wait for the result of the right neighbour and apply R (and possibly A) on its two local values. In other terms, a processor would behave exactly as a couple of neighbour processors in the previous system. This modification has a drastic effect on the efficiency as, in the best case, it allows us to halve execution time, or to achieve the same performance with half the number of processors. Other optimizations are currently under investigation and other evaluation schemes are under consideration.

Static analysis of Γ-programs.

Another way of optimizing Γ-programs could be by static analysis of their logical properties and program transformation. To come back to the optimization suggested above for the synchronous Γ-machine, it could be interesting to study the body of the action and the transformations applied to the aguments in order to decide whether the improvement may be applied or not. This topic should receive further attention in the future.

Implementation of a system for systematic program development.

An interesting area of investigation concerns the design of a suitable programmer interface for semi-automatic program development. Such an interface would allow for representation of relevant informations concerning the development process such as invariant and variant properties, a record of choices already investigated and of the choices still to be considered...

It would also be very interesting to put forward a method allowing the analysis of failures in the program development process. Such an analysis would certainly improve the productivity of the programmer and most probably clarify our understanding of this program development process.

REFERENCES

[BANATRE 86] Banâtre J.-P. and Le Métayer D., A new computational model and its discipline of programming, INRIA research report N° 566, Sept. 86.

[BANATRE 87] Banâtre J.-P. and Le Métayer D., A new approach to systematic program derivation, in Proceedings of the 1987 Workshop on Software Specification and Design (Monterey April 3-4), IEEE pp. 208-215.

[CODD 68] Codd E. F., Cellular Automata, Academic Press, 1968.

[COUTANT 86] Synthèse de programmes dans le formalisme Γ, D.E.A. Rennes, June 1986.

[DENNING 86] Denning P. J., Parallel computing and its evolution, Comm. ACM 29,12 (December 86), pp. 1163-1167.

[DIJKSTRA 76] Dijkstra E. W., A discipline of programming, Prentice-Hall, Englewood Cliffs, N.J., 1976.

[GRIES 81] Gries D., The science of programming, Springer-Verlag, 1981.

[HILLIS 85] Hillis W. D., The connection machine, MIT press, Cambridge, Mass., 1985.

[HOARE 78] Hoare C. A. R., Communicating Sequential Processes, Comm. ACM 21, 8 (August 1978), pp. 666-677.

[KUNG 82] Kung H. T., Why Systolic Architectures ?, Computer Magazine 15,1 (January 1982), pp. 37-46.

[WINSTON 84] Winston P. H., Artificial Intelligence, Addison-Wesley, 1984.

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