Part 1
Introduction to deliverable: purpose and approach 1
1. Introduction 1
2. Original description of the task 1
3. Virtual machine approach 2
4. Libraries approach 3
5. Overview of the work and organization of this report 4
Part 2
MENTAL: an Artificial Neural Networks description language and its' associated virtual machine 7
1. Artificial Neural Networks programming problems 7
1.1. Motivations 7
1.2. Main characteristics of Artificial Neural Networks 7
1.2.1. Static characteristics 7
1.2.2. Dynamic characteristics 8
1.3. Main difficulties 8
2. Principles of MENTAL 9
2.1. Principles of the virtual machine 9
2.1.1. Cells 9
2.1.2. Aggregates and connectors 11
2.1.3. Activation of synchronous aggregates 12
2.1.4. Activation of asynchronous aggregates 12
2.2. Principles of MENTAL: the A.N.N. description language 13
2.2.1. Overview of an A.N.N. program 13
2.2.2. Features of the language for static description 13
2.2.3. Features of the language for dynamic description 17
3. The virtual machine 18
3.1. Instructions set of the virtual machine 18
3.2. Grammar 19
Statements 19
Expressions 20
3.3. Activation of synchronous aggregates 21
3.4. Activation of asynchronous aggregates 21
3.5. From virtual machine to target machines 23
3.5.1. Toward SMART 23
3.5.2. Toward SuperNode 23
4. The A.N.N. description language 23
4.1. Instructions' set of the A.N.N. description language 23
4.2. Grammar 24
Program and external definitions 24
Statements 25
Expressions 25
Declarations 26
4.3. Game of life 28
4.3.1. Description of the problem 28
4.3.2. MENTAL program (first instance) 29
4.3.3. MENTAL Program (second instance) 31
4.4. Hoppfield 36
4.4.1. Description of the problem 36
4.4.2. MENTAL program 38
4.5. Back-propagation 40
4.5.1. Description of the problem: the "encoder" problem 40
4.5.2. MENTAL program 41
4.6. Kohonen 44
4.6.1. Description of the problem 44
4.6.2. MENTAL program 47
4.7. From A.N.N. description language to virtual machine 48

5. Comparative study with other neural networks programming environments....48
5.1. Artificial Neural Network simulators....48
5.1.1. P3:....48
5.1.2. Genesis....50
5.1.3. Nnsim....51
5.1.4. Nacre, Slogan et N....52
5.1.5. The Rochester Connectionnist Simulator....52
5.1.6. Nestor Development System (NDS)....53
5.1.7. Mirrors/ II....53
5.1.8. GN lami simulator....53
5.1.9. Cognitron....54
5.1.10. Ecn neurocomputing GmbH....54
5.1.11. DESIRE/NEUNET....55
5.2. Artificial Neural Network Languages....55
5.2.1. Condela....55
5.2.2. Nesila....56
5.2.3. Cellsim and CELIP....57
Part 3
Developments around SMART....59
1. Synopsis....59
2. The SMART appeal....59
3. An efficient hardware architecture....60
4. Software tools....61
5. State of development:....65
Part 4
Developments around SuperNode....67
1. The machine....67
1.1. The Transputer....67
1.2. SuperNode architectural model....67
2. Occam and parallel C....69
2.1. Occam....69
2.2. IMAG C....69
2.3. ANSI C (Parallel)....70
3. Neural....70
3.1. Main characteristics of Neural....70
3.2. Static characteristics....70
3.3. Dynamic characteristics....71
3.4. Execution on massively parallel machine characteristics....71
4. Present state of Development....72
Part 5
Applications....73
1. Applications using SMART....73
2. Applications using SuperNode....73
2.1. Parallel genetic algorithm and combinatorial optimization problems....74
2.1.1. Synopsis....74
2.1.2. Introduction....74
2.1.3. Mathematical formalization of the graph partitioning problem....75
2.1.4. Genetic algorithm solution to the graph partitioning problem....76
2.1.5. SuperNode implementation and benchmarking....78
2.1.6. Conclusions and future directions....83
2.2. A parallel genetic Boltzmann machine....84
2.2.1. Synopsis....84
2.2.2. Probabilistic inference....84
2.2.3. Parallel genetic algorithm....86
2.2.4. A parallel genetic Boltzmann machine....87
2.2.5. Conclusion and perspectives....87

Part 6
Conclusions and perspectives 89
Bibliography 91

MENTAL: A VIRTUAL MACHINE APPROACH TO ARTIFICIAL NEURAL NETWORKS PROGRAMMING

NERVES

ESPRIT B.R.A. PROJECT N°3049

FINAL REPORT

LANGUAGE AND SOFTWARE TOOLS: RESEARCH PACKAGE C TASK C1

June 30th 1991

Pierre Bessière[1]

LGI - IMAG / LASCO3

Laboratoire de Génie Informatique - Institut d'Informatique et de Mathématique Appliquée de Grenoble / LAboratoire de Sciences COgnitive - BP53X, F-38041 Grenoble Cedex, FRANCE.

Ali Chams[2]

LTIRF / INPG

Laboratoire de Traitement d'Image et de Reconnaissance des Formes / Institut National Polytechnique de Grenoble - 46 avenue Felix Viallet, F-38031 Grenoble Cedex, FRANCE

Philippe Chol[3]

LGI - IMAG / LASCO3

Laboratoire de Génie Informatique - Institut d'Informatique et de Mathématique Appliquée de Grenoble / LAboratoire de Sciences COgnitive - BP53X, F-38041 Grenoble Cedex, FRANCE.

---

[1]*E-mail: bessiere@imag.imag.fr; Phone: 33-76.51.45.72; Fax: 33-76.44.66.75; Telex: UJF 980 134 F*

[2]*E-mail: chams@tirf.imag.fr; Phone: 33-76.57.45.50.*

[3]*E-mail: chol@imag.imag.fr; Phone: 33-76.51.51.22.; Fax: 33-76.44.66.75; Telex: UJF 980 134 F*

SYNOPSIS:

1. CONTENTS

This report is the final **deliverable** of **ESPRIT B.R.A.** (Basic Research Action) **3049**, called **NE.R.V.E.S.** (NEurocomputing Research and V.L.S.I. for Enhanced Systems).

It corresponds to research package C (**Language and Software Tools**), task C1 (**High Level Specification Language**).

The three partners mainly involved in this task are:

- L.G.I./I.M.A.G. (Laboratoire de Génie Informatique / institut d'Informatique et de Mathématique Appliquée de Grenoble).
- L.T.I.R.F./I.N.P.G. (Laboratoire de Traitement d'Image et de Reconnaissance des Formes / Institut National Polytechnique de Grenoble)
- LA.M.I./E.P.F.L. (LAboratoire de Micro Informatique / Ecole Polytechnique Fédéral de Lausanne)

The purpose of this task is the design of a high level specification language for Artificial Neural Networks (A.N.N.).

2. OVERVIEW

An overview of the work is presented by figure 1. below.

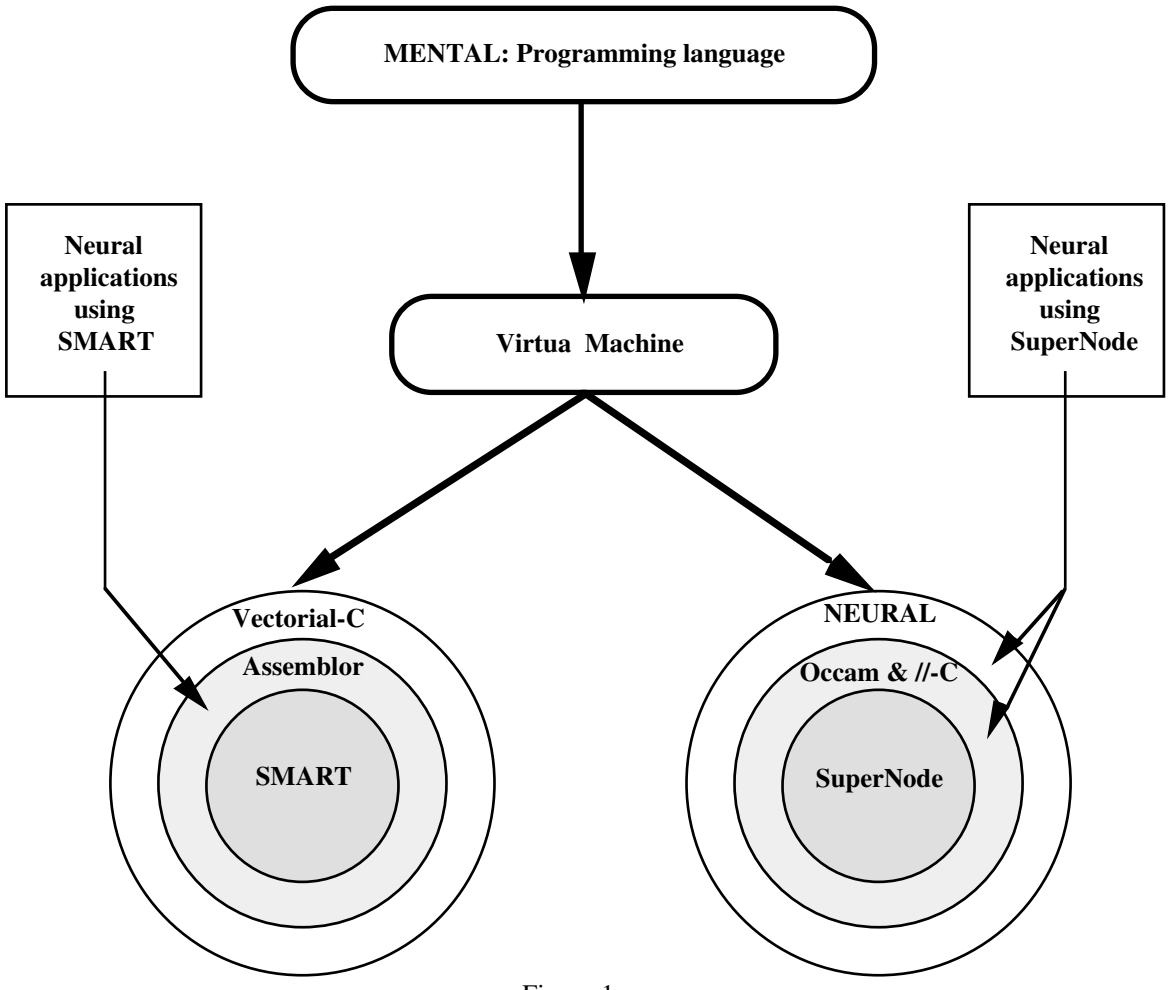

Figure 1.

The task was developed along three axes:

- A top-down axis where, starting from the different A.N.N. models, we tried to specify a high level description language (called MENTAL) for those models and a virtual machine to describe their execution. The virtual machine had to take into account the programming models of the different target parallel machines. The problems of the compiler from MENTAL to the virtual machine and of the compilers from the virtual machine to the different target machines were also considered in this part of the work. This axis is represented in figure 1. by bold frames and arrows.

- A bottom-up axis where, starting from the two target machines (SMART and SuperNode), we developed several layers of software around the kernel language of the machine. The work done along this axis is shown on figure 1. as concentric circles. The external layer corresponds to the language that will be the target language of the compilers going from the virtual machine to the target ones:
  1 - Vectorial-C, an extension of C dedicated to manipulate matrices and vectors, for SMART.
  2 - NEURAL, an extension of Occam dedicated to A.N.N. for SuperNode.

  Along this axis we also developed an assembly language for SMART. For SuperNode we used at this intermediary level either Occam, C3L parallel C or an home made parallel C.

- Finally, an applications axis where we developed several A.N.N. applications that actually run on either SMART or SuperNode. Those applications have been developed using directly one of the "internal" languages. We by-passed MENTAL because we wanted to have those applications running before any implementations of the high level language being completed. This axis is figured by the two square boxes and their associated arrows. For SMART several A.N.N. applications are under development using the assembly language and will be used as tests of its implementation. For SuperNode we developed numerous A.N.N. applications using either NEURAL, Occam or parallel C.

3. RESULTS

The main achievements of this task are:

- Along the top-down axis we achieved an extensive specification of MENTAL the A.N.N. description language and its' associated virtual machine. We specified, studied and presents in this report the following points:
  * the instruction set of the virtual machine and the semantic of its instructions;
  * the grammar of the virtual machine's language;
  * the problems encountered by the compilers going from the virtual machine to the target machines;
  * the instruction set of MENTAL;
  * Its' grammar;
  * the problems of the compiler going from MENTAL to the virtual machine;
  * several examples of well known A.N.N. models programmed using MENTAL

  and finally a comparative study with other A.N.N. programming environments.

- Along the bottom-up axis we achieved:
  * the specification and prototype implementation of an assembly language for SMART;
  * the specification of a Vectoriel-C, an extension of C dedicated to manipulate matrices and vectors for SMART;
  * the specification and prototype implementation of NEURAL, an extension of Occam dedicated to A.N.N., for SuperNode.

Finally, along the application axis, we achieved:

* A study of the implementation of two well known A.N.N. models (Kohonen's model and Herault-Jutten's source separation algorithm) in SMART's assembly language. These algorithms are not running yet due to the present unavailability of SMART.

* An implementation of a parallel genetic algorithm on SuperNode and MegaNode (up to 128 Transputers). Its' use for different optimization problems as for instance the "graph partitioning problem".

* An implementation of a "Genetic Boltzmann Machine" on SuperNode which is presently used for some robotic experimentations.

4. PERSPECTIVES

The development of this task reached a point where all the preliminary work has been done in order to realize a high level specification language for Artificial Neural Networks.

The feasibility of such a language has been proved.

However, It appears obvious that in order to get a real implementation of such a language, susceptible to be either distributed or commercialize a considerable effort should still be done. This kind of work is a matter for an industrial type ESPRIT project and not any longer for a B.R.A. type project.

PART 1

INTRODUCTION TO DELIVERABLE: PURPOSE AND APPROACH

1. INTRODUCTION

This report is the final **deliverable** of **ESPRIT B.R.A.** (Basic Research Action) **3049**, called **NE.R.V.E.S.** (NEurocomputing Research and V.L.S.I. for Enhanced Systems).

It corresponds to research package C (**Language and Software Tools**), task C1 (**High Level Specification Language**).

The three partners mainly involved in this task are:

- L.G.I./I.M.A.G. (Laboratoire de Génie Informatique / institut d'Informatique et de Mathématique Appliquée de Grenoble).
- L.T.I.R.F./I.N.P.G. (Laboratoire de Traitement d'Image et de Reconnaissance des Formes / Institut National Polytechnique de Grenoble)
- LA.M.I./E.P.F.L. (LAboratoire de Micro Informatique / Ecole Polytechnique Fédéral de Lausanne)

The different partners were engaged for the following efforts:

- L.G.I./I.M.A.G.: 17 man.months;
- LA.M.I./E.P.F.L.: 6 man.months (Swiss funds);
- L.T.I.R.F./I.N.P.G.: 3 man.months funded by C.E.C. plus 9 man.months with their own funds.

There was no swiss participation to the second year of this task. All their effort has been done during the first year. We should report to deliverable nb. 1 to get a description of their work.

2. ORIGINAL DESCRIPTION OF THE TASK

The following describes the task as stated in the contract:

The purpose of this task is the design of a high level specification language for neural networks. This environment will be based on a general purpose programming language or programming model which fits well the characteristics of such architectures.

Such a language will be useful for the following reasons:

- *as a way to program the different neuro-computers (for instance Smart) and circuits used by the partners of the project;*
- *as a way to specify simulation of neural networks on non dedicated computers (for instance, Sun work station and T-node of Transputers);*
- *as a mean of communication between the partners to exchange ideas and software and to compare their different experiences.*

The different sub-tasks are:

1 - conception and design of a neural virtual machine and its' set of instructions;

2 - conception and design of the high level specification language and the compiler to transform this code into code for the virtual machine;

3 - conception and design of compilers and/or libraries to implement the virtual machine on the different considered hardwares;

4 - conception and design of programming tools such as graphical interfaces to both specified neural nets and to visualized results of computations.

The main theoretical difficulties of this task will be the correlated definition of both the virtual machine and the high level specification language (sub-tasks 1 and 2). The first problem is to design a language general enough to program a large range of neural networks. The second problem is to conceive a virtual machine independent of the different considered hardwares. Those two sub-tasks will require a lot of theoretical work. That doesn't mean that sub-tasks 3 and 4 won't need very substantial programming efforts.

Two different approaches for this task have been considered since the beginning of the work:

- the first one, called the "virtual machine" approach, has very ambitious scientific objectives and high research interest, but requires much more work than funds available for this project;
- the second one, called "libraries" approach, has more pragmatic objectives and suppose more realistic efforts.

The two approaches are introduced in the next two paragraphs. However, it was decided at the beginning of the project to follow both tracks in parallel. The libraries approach to achieve some practical results by the end of this project and the virtual machine approach because of its scientific interest.

3. VIRTUAL MACHINE APPROACH

This section introduces the virtual machine approach to A.N.N. (Artificial Neural Networks) programming.

We will introduce mainly the concepts of a virtual machine which combines various programming paradigms encountered in A.N.N..

The implementation principle is a classical virtual machine approach (see fig. 1.1. below).

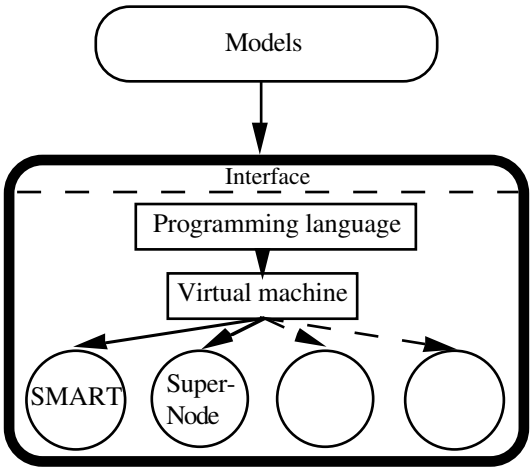

Figure 1.1.

The work to be done corresponds to the black frame:

1 - definition, specification and implementation of a high level programming language for A.N.N.;
2 - definition, specification and implementation of the virtual machine for A.N.N.;
3 - compiler from the high level programming language to the virtual machines;
4 - compilers from the virtual machines to the different target machines;
5 - man-machine interfaces.

In the context of this project two target machines were firstly selected: SMART and SuperNode. Detailed descriptions of those machines can be found in parts 3 and 4 respectively.

The introduction of a virtual machine has several well known advantages:

- the definition of the virtual machine instructions ensures clarification of the fundamental concepts and their semantic properties;
- the programming language for A.N.N. is independent of the actual target machines;

- only the compiler between the virtual machine and the target machine need to be written when a new target machine is considered.

However, this approach has some difficulties, especially, the clarification of the fundamental concepts when defining the virtual machine. This is one of the advanced research topics.

The detailed description of this approach and of the work realized up to now is discussed in part 2 of this report.

It should be noted that three international publications directly describing work done for this task have already be published:

- *Un modèle de machine abstraite pour la programmation de réseaux de neurones formels*
  Pierre Bessière, Ali Chams, Traian Muntean & Philippe Chol
  Proceedings of N.S.I.90 (Neurosciences et Sciences de l'Ingénieur), Aussois, 1990

- *A virtual machine model for Artificial Neural Network programming*
  Pierre Bessière, Ali Chams & Traian Muntean
  Proceedings of I.N.N.C.90 (International Neural Network Conference), Paris, 1990

- *From hardware to software: designing a "Neurostation"*
  Pierre Bessière, Ali Chams, Anne Guérin, Jeanny Hérault,
  Christian Jutten & Jean-Christophe Lawson
  in *Introduction to V.L.S.I. design of Artificial Neural Networks*
  Kluever Academic Publishers, 1990

and five more international publications describing applications related with the task:

- *A parallel genetic algorithm for the graph partitionning problem*
  E-G. Talbi & P. Bessière
  A.C.M. International Conference on SuperComputing, Cologne, Germany, Juin 1991

- *Superlinear performance of a genetic algorithm on the SuperNode parallel architecture*
  E-G. Talbi & P. Bessière
  S.I.A.M. News, 1991

- *Genetic parallel algorithm: performances and applications*
  Proceedings of Intenational Conference on Novel Optimization Technics, Copenhagen, Danemark, February 1991

- *A genetic Boltzmann machine*
  P. Bessière
  I.J.C.N.N.91 International Join Conference on Neural Networks, Seattle, U.S.A., 1991

- *GENETIC ALGORITHMS APPLIED TO FORMAL NEURAL NETWORKS: parallel genetic implementation of a Boltzmann machine and associated robotic experimentations*
  P. Bessière
  Submited to E.C.A.L.91 First European Conference on Artificial Life, Paris, France, Décembre 1991

4. LIBRARIES APPROACH

This approach involves in a great emphasis on the human interface and specialization on interactive programs that can be utilized both for research and didactic purposes.

The libraries approach consists of the development of portable and comprehensible code for simulating A.N.N.s. In this case, creating a neural network language does not follow the same approach as that of the virtual

machine. It has rather been decided to write a set of programs using a standard general purpose language, i.e. C, on widely available workstations, i.e. Sun, under a standard operating system, i.e. Unix.

These programs cover most of the popular neural networks learning algorithms:

1 - Hopfield network: synchronous and asynchronous update, auto- and hetero-associative modes;
2 - Adaline network;
3 - Back-propagation: feed-forward and recurrent networks;
4 - Boltzmann machine;
5 - Kohonen's self organizing map: uni and bi-dimensional maps.

Items 3 and 5 above also have graphical output to improve interaction and understanding of network operation. Similar interfaces for other models are under development. All models run on Appolo and Sun workstations. Except the graphics interface, they also compile and execute on I.B.M. PC.s and compatibles.

A detailed description of this part of the work may be found in a second document relative to the task C1 and written by EPFL.

5. OVERVIEW OF THE WORK AND ORGANIZATION OF THIS REPORT

An overview of the work is presented by figure 1.2. below.

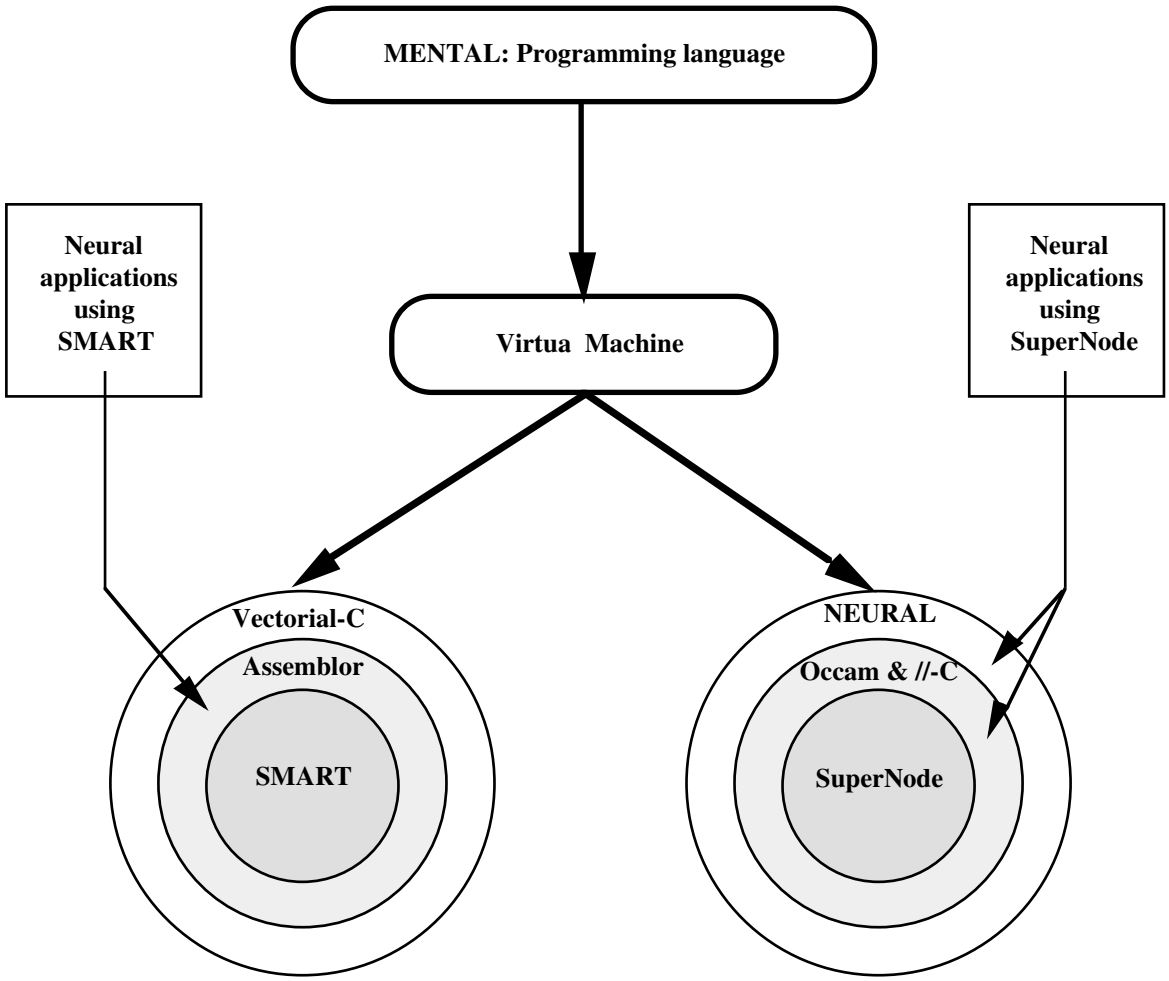

Figure 1.2

The task was developed along three axes:

- A top-down axis where, starting from the different A.N.N. models, we tried to specify a high level description language (called MENTAL) for those models and a virtual machine to describe their execution. The virtual machine had to take into account the programming models of the different target parallel machines. The problems of the compiler from MENTAL to the virtual machine and of the compilers from the virtual machine to the different target machines were also considered in this part of the work. This axis is represented in figure 1.2. by bold frames and arrows.

- A bottom-up axis where, starting from the two target machines (SMART and SuperNode), we developed several layers of software around the kernel language of the machine. The work done along this axis is shown on figure 1.2. as concentric circles. The external layer corresponds to the language that will be the target language of the compilers going from the virtual machine to the target ones:

1- Vectorial-C, an extension of C dedicated to manipulate matrices and vectors, for SMART.

2- NEURAL, an extension of Occam dedicated to A.N.N. [Chol88], for SuperNode.

Along this axis we also developed an assembly language for SMART. For SuperNode we used at this intermediary level either Occam, C3L parallel C or an home made parallel C.

- Finally, an applications axis where we developed several A.N.N. applications that actually run on either SMART or SuperNode. Those applications have been developed using directly one of the "internal" languages. We by-passed MENTAL because we wanted to have those applications running before any implementations of the high level language being completed. This axis is figured by the two square boxes and their associated arrows. For SMART several A.N.N. applications were developed using the assembly language and used as tests of its implementation. For SuperNode we developed numerous A.N.N. applications using either NEURAL, Occam or parallel C.

This report is organized along those three same axis (see figure 1.3.):

- Part 2 concerns the work done along the bottom-up axis. It describes extensively MENTAL the A.N.N. description language and its' associated virtual machine. After an analysis of the main A.N.N. programming difficulties and characteristics, we present a rapid overview of the principles of both the virtual machine and the high level description language. Going into some more detailed descriptions, we then present extensively:
  * the instruction set of the virtual machine and the semantic of its instructions;
  * the grammar of the virtual machine's language;
  * the problems encountered by the compilers going from the virtual machine to the target machines;
  * the instruction set of MENTAL;
  * Its' grammar;
  * the problems of the compiler going from MENTAL to the virtual machine;
  * several examples of well known A.N.N. models programmed using MENTAL

  and finally a comparative study with other A.N.N. programming environments.
- Part 3 and 4 describe respectively the developments around SMART and SuperNode after a short review of both architectures.
- Part 5 present extensive descriptions of some of the applications developed using either SMART or SuperNode. For instance, on SuperNode we describe:
  * a parallel implementation of genetic algorithms used for combinatorial optimization's problems;
  * a parallel genetic Boltzmann machine;
  * and a Conway's game of life implemented on SuperNode using NEURAL.
- Finally, part 6 present the conclusions of this study and the different perspectives.

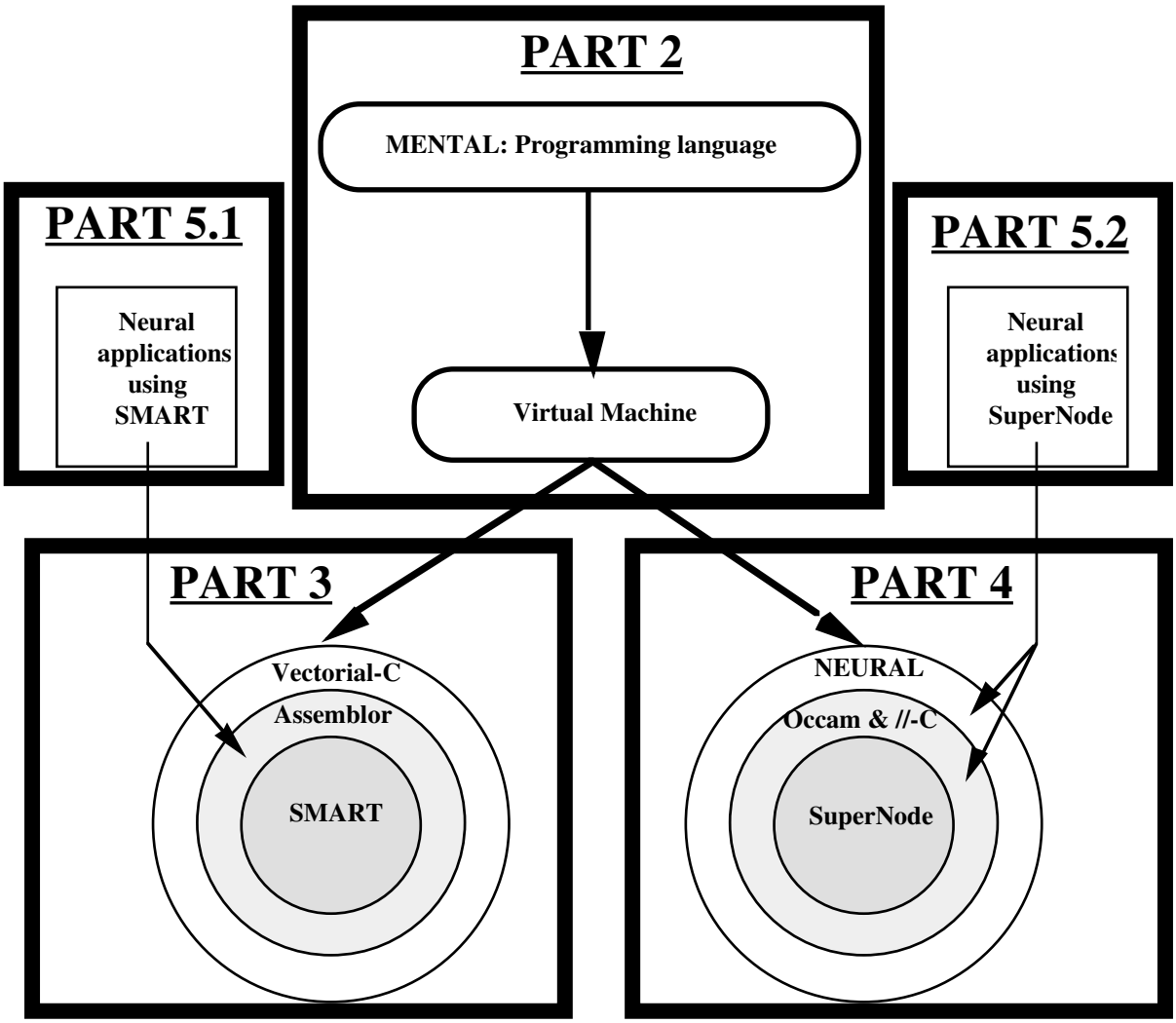
PART 2
MENTAL: Programming language
Virtual Machine
PART 5.1
Neural applications using SMART
PART 5.2
Neural applications using SuperNode
PART 3
Vectorial-C
Assemblor
SMART
PART 4
NEURAL
Occam & //-C
SuperNode

Figure 1.3.

PART 2

MENTAL: AN ARTIFICIAL NEURAL NETWORKS DESCRIPTION LANGUAGE AND ITS' ASSOCIATED VIRTUAL MACHINE

1. ARTIFICIAL NEURAL NETWORKS PROGRAMMING PROBLEMS

1.1. Motivations

A large number of A.N.N. models have been described, each different in detail. These various models, and indeed many in the literature, clearly have many features in common, but they are just as clearly distinct models.

How to characterize a general purpose machine to implement A.N.N. is one of the basic research topics.

In general, A.N.N. are intrinsically highly parallel. However, most existing implementations are still sequential.

Numerous works are under way to build parallel machines dedicated to A.N.N.. Such parallel computers need a programming environment with user friendly interfaces and powerful programming languages.

Since several machines of this kind are under construction in Grenoble (SuperNode, SMART), we were specially motivated to design a new programming environment to support A.N.N. models on different machines. This language must offer to programmers powerful tools to manipulate A.N.N..

1.2. Main characteristics of Artificial Neural Networks

In this paragraph we present a comprehensive analysis of A.N.N.'s characteristics.

The purpose of this analysis is to point out A.N.N.'s particularities. This is of critical importance because a good high level programming language for A.N.N. will have to propose powerful features to describe easily those particularities. Those features will be useful to "program" A.N.N. and to build a graphical interface to display the result of a computation.

Two main aspects are considered:

- the static characteristics, essentially related to the net's topology, concern the description of an A.N.N. as a static object;
- the dynamic characteristics concern the description of an A.N.N. as an object evolving in time.

1.2.1. Static characteristics

Artificial Neural Networks may be seen statically as a set of numerous simple processes (usually called cells) exchanging information through a huge number of connections.

Some order of magnitudes should be first recalled:

- the number of cells in a net, ranges from few units in the simplest A.N.N. to 10^{10} in the brain, (typically of the order of few hundreds in present A.N.N.);
- the number of connections per cell ranges from less than ten for very regularly structured nets (see, for instance, "game of life" [Gardner70] and [Poundstone85], other Cellular Automata Nets [Toffoli85] and "competitive learning" [Rumelhart86b]) to 10^5 in parts of the human brain, (typically of the order of a few tens in present A.N.N.).

This demonstrates that a net is a very complex object, made up of a huge number of components: hundreds of cells and thousands of connections. Such an object is obviously difficult to describe. However, A.N.N. are highly organized objects with a rich "topology", which means that:

- numbers of their cells are identical;
- same patterns of regular connections may be found at different places in a net;
- cells are usually organized into clusters and sub-clusters of increasing connectivity (for a formal definition of connectivity see [Ghosh89]);
- most of the time, an A.N.N. programmer has a mental representation of a net or parts of it as spatial objects in 2D or 3D spaces, this mental representation is a great help to address a specific cell or a particular connection (see [Zisper86]) and to visualize networks in a comprehensible manner.

A high level language for A.N.N. should, definitely, take advantage of this rich topology to help the programmer describing a net and to help the man-machine interface program to visualize a net. It should provide special features to code this topology. We will describe the proposed features in § 2.2.2..

1.2.2. Dynamic characteristics

Different types of A.N.N. have very different kinds of dynamic characteristics ranging from full synchronicity to completely asynchronous ones. The type of dynamic features of an A.N.N. is one of its major characteristics. The study of A.N.N. as dynamic complex systems is a very promising approach (see, for instance, [Weisbuch89]). The parallelism between the different units ranges from full parallelism between all units to parallelism inside a group of units with sequential activities between the different groups.

Cellular Automata (see [Gardner70] & [Toffoli85]) are completely synchronous and fully parallel. All cells are activated at the same time.

In competitive learning (see [Rumelhart86b]) cells are topologically organized in layers and within each layer in clusters. There is a synchronous parallel activation of cells to transfer information from one layer to the next and also within each cluster. However, the activation of those different sets of cells is organized sequentially:

1 - transfer information from layer n-1 to layer n;
2 - activate clusters of layer n;
3 - transfer information from layer n to layer n+1;
etc...

In nets organized topologically in layers, like the ones for Back-Propagation (see, for instance, [Rumelhart86c]), the activation of the cells is parallel and synchronous within a layer and sequential to propagate activities from one layer to the next.

In a Boltzmann machine (see for instance [Ackley88]) each cell's activation time is independent of all others. On sequential machines this is simulated by random order of activation, on parallel machines this means that the activity of cells is completely asynchronous.

In more realistic biological nets, activation is usually completely asynchronous. The activity of a cell is determined by the number and intensity of stimuli and by various internal parameters.

According to those remarks, the information concerning cell activation may be distributed in two different places:

- in a global control program for activating a synchronous group of cells;
- in each cell for asynchronously activating this particular cell.

A high level programming language for A.N.N. must offer the two different possibilities (see [Chol88])**.** This will be detailed in § 2.2.3..

1.3. Main difficulties

Two main difficulties have been encountered:

- Usually, A.N.N. have simple treatment processes and complex interconnection schemes between them. Consequently, a programming language for A.N.N. has to propose powerful features to describe this aspect. The first difficulty has been precisely to find a viable and efficient way to specify the network architecture and connectivity since it is not a common feature of existing programming languages. This corresponds to the static description of a network and is discussed in § 2.2.2..

- The design of a programming language should be considered independently of the features of the different computers. Classical sequential architectures, even if they lead to different computers, share the same underlying programming paradigm. In parallel computing the situation is quite different, there are several very different parallel programming models due to various possible control structures of concurrency; synchrony or asynchrony of process behavior is to be considered, and non-determinism due to multiple execution paths can arise. A hard problem for a parallel programming language for A.N.N. is to make a synthesis between those different parallel programming paradigms. Those aspects are the subject of sections 2.1. and 3..

2. PRINCIPLES OF MENTAL

2.1. Principles of the virtual machine

The synthesis between the different parallel programming paradigms is based on the following concepts introduced for the description of any A.N.N. at the virtual machine level:

- cell;
- aggregate and connector.

As a very first approximation, a network could be represented graphically by figure 2.1. where circles represent cells, sets of cells represent aggregates, and lozenges are connectors.

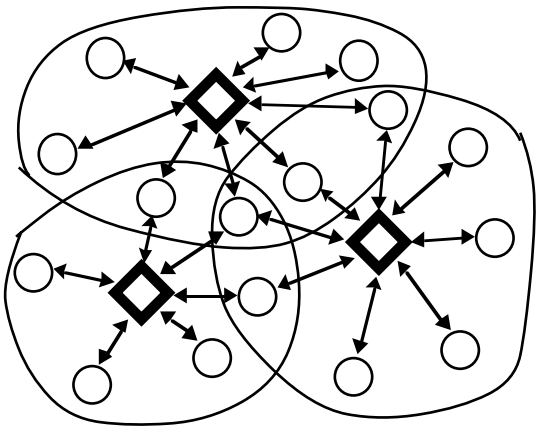

Figure 2.1.

2.1.1. Cells

A cell is a process which gets some inputs from the surrounding cells, executes its own code, produces some outputs and returns them to the surrounding.

A cell executes its program as soon as its triggering conditions are met. Among these triggering conditions, availability of the inputs are one of the main prerequisite. Some outputs are computed and may be broadcast through the net to be used as inputs of other cells. Consequently, there is no way to control explicitly the execution of cells' programs. Cells behavior is only a function of their interaction and their description program, so the control structures of the language should reflect the type of synchronizations attached to any interaction. This function will be devoted to objects called connectors which are described in the next section.

A cell may be part of one or several aggregates (set of cells). The following scheme represents a cell when it belongs to one aggregate (see figure 2.2.):

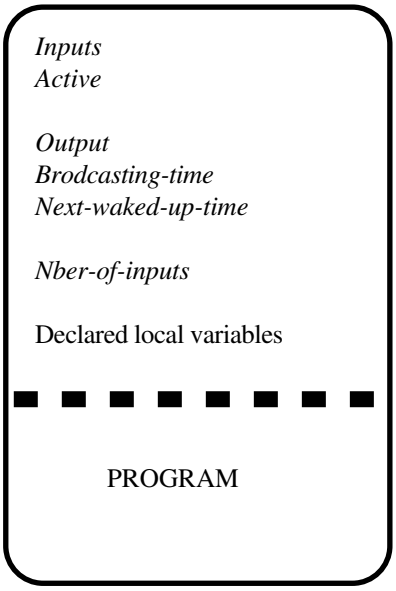

Figure 2.2.

The only constraints imposed on a cell are on its interface with the connectors. It should respect some given communication protocols to exchange information with the connector. Its own internal program may be anything. A synchronous cell should accept a scalar as its Input and returns a scalar as its Output. An asynchronous cell, should, further more, return a Broadcasting-time (which can be superior to the local time to take into account modeling of delays) and a Next-wake-up-time (which tells the connector when to wake up the cell).

In *italics* appear the predeclared variables of every cells. These variables are intrinsic parts of the data structure of a cell. They have the following use:

- *Inputs*: this variable stocks the value of the inputs sent by the connector.
- *Active*: this variable is Boolean, set to one by the connector when it wants to activate the cell.
- *Output*: this variable is a scalar and must be set by the cell to a value that will be made available to the connector as the result of the cell computation.
- *Broadcasting-time*: this variable is set by the cell to the time at which the cell makes its result available to the connector. it may be notably used to simulate delays either in the cell's computation or in the synaptic transfers.
- *Next-wake-up-time*: this variable is set by the cell to indicate to the connector the time of the next wake up.
- *Nber-of-inputs*: this variable is automatically set to the length of the vector *Inputs*. It may be used in the cell's program for iterative treatments on the inputs.

Then, the programmer may specified any number of local declared variables to be used by the cell's program beside the variables imposed by the connector: an *Output*, a *Broadcasting-time* and a *Next-wake-up-time*.

However, any cell, belonged to several aggregates, may be represented by following scheme (see figure 2.3.).

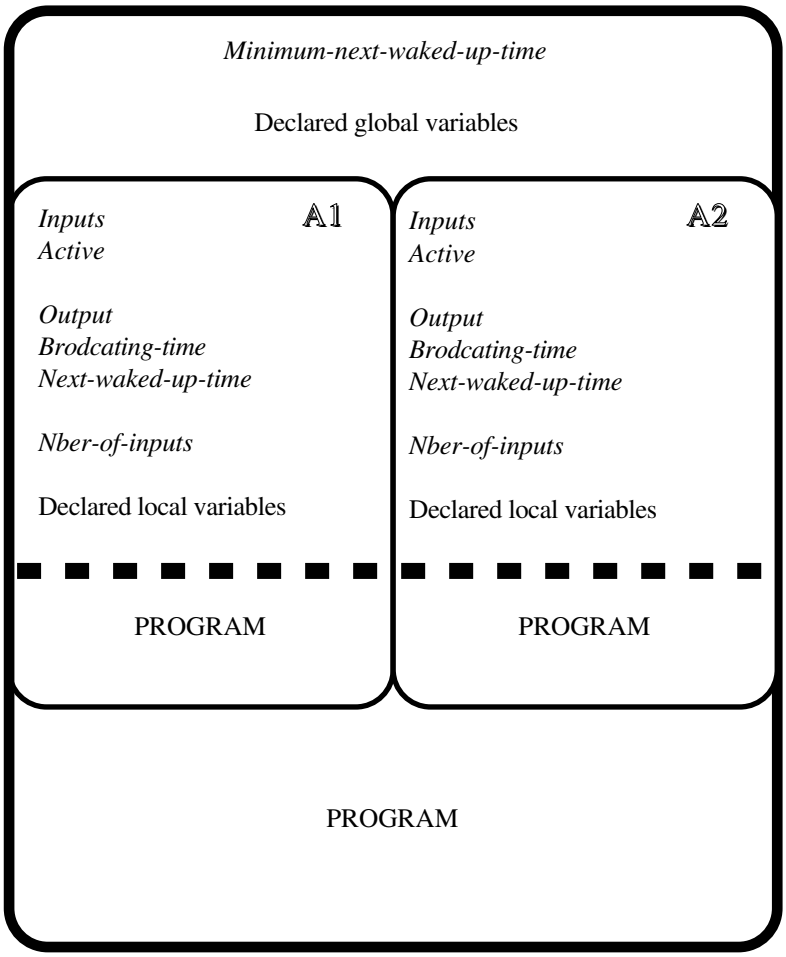

Figure 2.3.

When a cell belongs to several aggregates, it is divided into as many processes as aggregates (Here, two aggregates A1 and A2), each of these processes is equivalent to a cell for one aggregate. However, some extra features are necessary: an extra pre-declared global variable called *Minimum-next-wake-up-time* (equal to the minimum of the next-wake-up-time of all processes) used by the connectors' protocol, some extra declared global variables, and a common piece of code.

A cell belonging to several aggregates may either be activated independently by each one (for instance, a cell belonging both to a layer and to a cluster in competitive learning [Rumelhart86b]), or imposed that all aggregates must be synchronized when the activating of this cell took place (for instance, a cell belonging to several aggregates in a Boltzmann machine [Ackley88]).

2.1.2. Aggregates and connectors

An aggregate is a set of potentially fully connected cells, whose activities are temporarily correlated. Most of the time an aggregate corresponds to an highly internally connected part of the net.

Any cell in an aggregate can potentially exchange information with any other cell in the same aggregate. This exchange of information can be synchronous or asynchronous. In both cases, this correlation is expressed by a protocol. The connectors are communication objects defined to implement those protocols.

The principal information stored in a connector is a connection matrix $\boldsymbol{W}$. If an aggregate connects N cells, the matrix size is N^2. In most cases the connection matrix is sparse, and its structure reflects the topological structure of the aggregate.

In fact, the connection matrix encodes more than communication capabilities, it contains entries (real or integer) usually called synaptic weights. These weights are used as parameters for a composition function Π and an integration function Σ which are used to transform information exchanged by the cells. The broadcasted information should go through the function $\Sigma o \Pi$ before being transmitted to the cells of an aggregate. Typically, Π is a multiplication function and Σ is a summation one.

A basic operation of the virtual machine is "**activate**"; the activation of an aggregate. Roughly speaking, this activation consists in broadcasting the informations delivered by every cell of the aggregate to the others, through the transformation $\Sigma o \Pi$. This process implies the automatic execution of the programs associated to each cells of an aggregate, as soon as their inputs are available and their triggering conditions are met. In fact, the differences between synchronous and asynchronous models appear in this operation. The synchronous and asynchronous activation of aggregates are detailed in the two next section.

Another basic operation of the virtual machine is "**learn**": executing learning in a given aggregate. A learning function Λ is associated with each aggregate. For any given connection matrix and some parameters reflecting the past activity of the network, Λ produces a new connection matrix.

2.1.3. Activation of synchronous aggregates

If we suppose that each cell delivers an output, so each synchronous aggregate has a vector $\boldsymbol{V}$ representing the output values for all the cells; then v_j (*j*th component of $\boldsymbol{V}$) gives the output of the *j*th cell in the aggregate.

The composition function Π can be applied for each couple (w_{ij}, v_j) to obtain c_{ij}, then the integration function Σ produces for all N-tuples of c_{ij} the resultant vector $\boldsymbol{R}$. In the case where Π is multiplication and Σ is addition, the whole operation can be written using a matrix operation $\boldsymbol{R = W * V}$.

Each r_i can be an input for the *j*th cell, whenever the inputs are available, the *j*th cell then re-executes its own program automatically to produce the new v_j, for another iteration.

In the synchronous case, the role of a connector is very simple; transmitting data between cells from a static ?-set to a static !-set of cells.

2.1.4. Activation of asynchronous aggregates

The asynchronous model has "time" as an extra information to be exchanged between cells. Three values are delivered by an asynchronous cell after execution of its program:

- the output;
- the time when the values are delivered;
- the expected time for the next exchange of information between the connector and the cell.

The temporal information is controlled by the connector which will wake up the cells at the right time. Asynchronous models use the data structures described previously for synchronous models, with the additional items:

- a vector $\boldsymbol{B}$ containing the arrival time of the last messages coming from each cell (b_i for the *i*th cell);
- a vector ***Nmin*** containing the next activation time for each cell;
- a counter $\boldsymbol{t}$ representing the last activation time for a cell in the same aggregate which can be interpreted as the own local time of the aggregate.

After the initialization phase of $\boldsymbol{W}$, $\boldsymbol{V}$, $\boldsymbol{B}$, ***Nmin***, and $\boldsymbol{t}$ ($\boldsymbol{t} = 0$), the activation of an aggregate is described by the following algorithm:

- search the next cell(s) to wake up and update $\boldsymbol{t}$;
- send to all such cells the vector $\boldsymbol{V}$ after application of the composition function Π;
- send to these cells the vector $\boldsymbol{B}$;
- all cells execute the function Σ and their own programs as soon as $\boldsymbol{V}$ and $\boldsymbol{B}$ are available;
- each cell i sends to the connector its output values, the present time (b_i), and its next activation time (n_i).

Note that the synchronous model can be emulated in the asynchronous mode by waking up the cells at each clock step.

The role of the connector is much more complicated here than in the synchronous case. In particular the ?-set of cells (cells that have to be consulted) and the !-set of cells (cells that have to be waked up) are not any more statically specified and must be computed by the connector at each step. The !-set is determined comparing ***t*** to the ***Nmin*** of all the cells of the aggregate and the ?-set is determined after the !-set by computing all the cells of the aggregate which are connected to those of the !-set and are consequently potential sources of information.

It should also be noted that different aggregates may be forced to implicitly synchronized their activities at some time steps, due to the fact that they share some cells.

2.2. Principles of MENTAL: the A.N.N. description language

2.2.1. Overview of an A.N.N. program

An A.N.N. program in our high level language has the usual structure of a C program. First comes the instructions for the preprocessor preceded by a # sign, second comes the definitions part of the program where the different functions, groups, aggregates and cells are defined, finally, comes the program itself. The usual main() segment of a C program is replaced in MENTAL by a control() segment which plays the same role than the usual main but may be also considered as the code of the virtual machine. The A.N.N. program is compose of three parts:

1 - The preprocessor instructions part.

2 - A declaration part where the behavior of the different processes, the topology of the various sets of processes and the structure of the links between them are described.

3 - A control program which encodes how the network will evolve dynamically.

Let us now introduced the different specific features of MENTAL. They are extensions of the usual C language and detailed grammar will be given further on in § 4.2..

2.2.2. Features of the language for static description

Most of the proposed languages, oriented to the description of neural nets, lack of a special feature to take into account the topology of the net (see: General framework for parallel distributed processing [Rumelhart86a], R.C.S. [Goddard87], P3 [Zisper86], Pygmalion neural network programming environment [Angeniol88] [Derot89] [Hewetson89], etc...). In particular, each single connection has to be described by stating its departure cell and its goal cell. Our language is designed to offer the possibility of defining sets of cells and sets of connections in a synthetic way. These features will be described in this section. It should be noted that CONDELA [Köhle88] and NESILA [Korb89] offers some comparable features.

a - Groups

A group is a set of cells sharing some "structural" or "topological" properties.

The phrase "Sharing some structural properties" is defined to mean that all cells of the group have in common some aspects of their behavior. It might be their activation scheme and as well their learning behavior or any other behavioral aspects.

A good instance of such set of cells is a layer of formal neurons within a multi-layer network. All the cells of such a layer share the same behavior even if they are not connected to one another.

Often, such sets of cells are mentally organized in the mind of the programmer into geometric structures.

For a group of cells, sharing some topological properties means that in between the different cells of the set some regular patterns of connections might be found.

A good instance of such a set of cells is the union of two layers in a multi-layer network. The feed-forward pattern of connections from one layer to the next is the relevant pattern.

The same remark as previously can be made about the geometric mental representation of such a set of cells.

A group is a set of cells, which members are given in a slot called CELLS (see § b). Other interesting characteristics of a group are topological characteristics, namely SHAPE (see § c), PAVING (see § d) and NEIGHBORHOOD (see § e).

Furthermore, associated with the topological structure of groups, comes an easy way to address cells, connections and sub-components of these two elements, namely the "Access functions" (see § f).

b - Cells

The first characteristic of a group of cells is to be a set of cells. As such, a group may be defined as any set in extension (a list of the cells belonging to the set), in comprehension (by the characteristics of cells belonging to this set) or as the result of algebraic operations on other sets (union, intersection, complementation, etc...). The nature of all cells is given, allowing the compiler to find the corresponding code.

c - Shape

The SHAPE of a group is a geometrical space where the cells are mentally placed by the programmer.

Shapes could be "open" spaces like straight lines, plans, hyperplans or hypercubes; or they could be "closed" spaces like spheres or torus (see figure 2.4.).

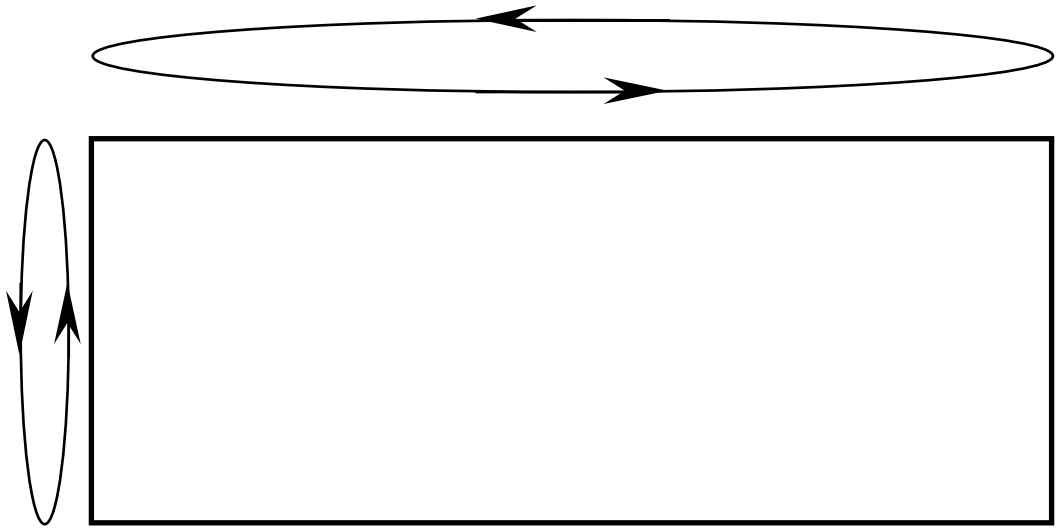

Figure 2.4.

d - Paving

The PAVING defines where, for a given space, the cells can be located.

For instance a plan can be associated with a square paving (see figure 2.5.) or an hexagonal paving. With each paving comes a specific way to address the cells.

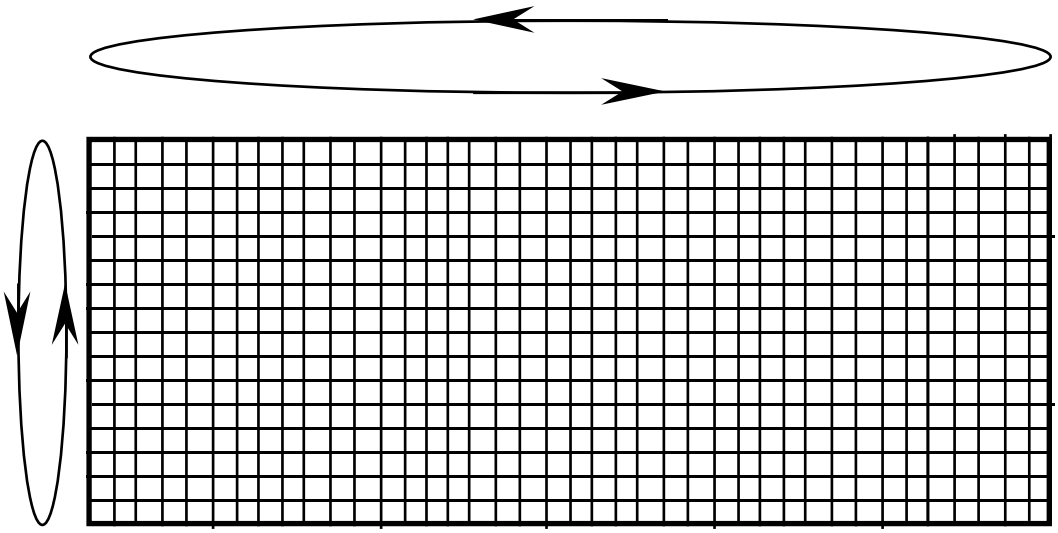

Figure 2.5.

e - Neighborhood

For a given shape and paving, we can define a NEIGHBORHOOD. The Neighborhood defines the standard pattern of links existing between cells.

For instance, on a plan with a square paving we can define a "4-nearest-neighbors", a "8-nearest-neighbors" (see figure 2.6.) or a "12-nearest-hamming-neighbors" neighborhood.

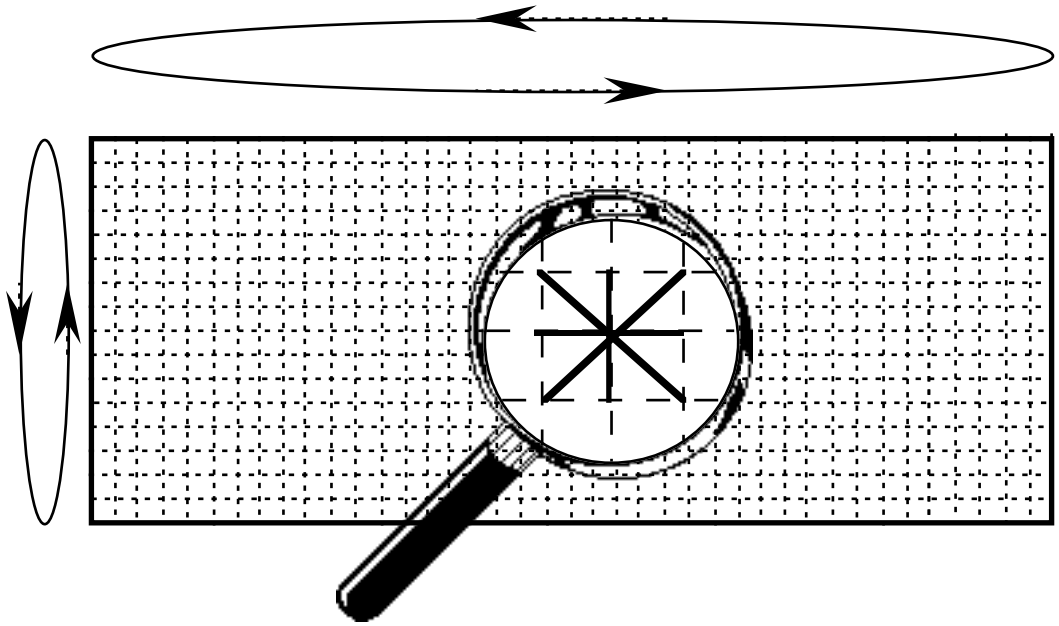

Figure 2.6.

A neighborhood offers the possibility to defined very concisely a large number of connections; however, it may be useful to be able to define a particular link from a given cell to a specific other one. Consequently, the language allows the definition of links between cells on a cell to cell basis. It also allows to disconnect some cells that otherwise would have been connected given the definition of the neighborhood.

Once a shape, a paving and a neighborhood have been defined for a group of cells:

- a programmer has a mental representation of this group of cells, and knows how to address precisely each cell and each link (see § f);
- a compiler may compute a connection matrix;
- a man-machine interface has some information about how to display a group of cells.

f - Connectivity

For a given shape, a given paving and a given neighborhood, we can finally described the connectivity of the net.

Standard types of connectivity are described by predefined key-words such as **Complete**, **Symmetric**, **Full-forward**, etc...

Non standard connectivity may be described using a piece of code describing by means of embedded loops, written in C, which cell to connect to which other cell. A specific **Connect** function taking two cells as its' parameters is available to do this.

Another mean to describe a connectivity is to take a predefined connectivity and to disconnect few cells among the connected ones. A specific **Disconnect** function taking also two cells as its' parameters has been designed for that use.

g - Access functions

Groups of cells are the only objects to be explicitly named.

Other objects (cells, weights, connections, variables, etc...) are addressed relatively to those groups using access functions.

For instance, a cell is addressed using the access function *Cell*. As a cell is a member of at least one group, eventually several, *Cell* takes as first argument the name of one of these groups, and as following arguments, integers, representing the indices of the addressed cell associated with the paving of the selected group. The number of necessary indices corresponds to the dimensionality of the group's shape.

Examples:

Let suppose that we had defined a torus FOO of size NxM with a square paving and 8-nearest-neighbors neighborhood, then we have the following access expressions:

- the cell at the top, left corner -> *Cell* (FOO, 0, 0) or *Cell* (FOO, N, M) (because of the torus shape of FOO);
- the weight of cell (1,1) to cell (0,0) -> *Weight* (*Cell* (FOO, 1, 1), *Cell* (FOO, 0, 0))
- etc...

Here follows a list of main access function:

- *Time* given an aggregate returns the value of the internal clock of this aggregate;
- *Weight* given two cells or an aggregate and two cells returns the weight of that link;
- *Cell* given an aggregate and some indices returns a pointer to a cell;
- *Output* given a cell returns the output of this cell;
- *Broadcasting-time* given a cell returns the last time of broadcasting of this cell;
- *Next-wake-up-time* given a cell returns the time when this cell wants to be waked up;
- *Inputs* given a cell returns the input(s) of this cell.

A simplified grammar of main predefined access functions would look as follows:

Explanation of used notation for the grammar:

- <, >, ::=, [,], {, }, | are the symbols of the BNF meta-language with their usual meanings;
- <Foo>: means Foo is a non terminal symbol of the grammar;
- Underline words are terminal symbols of the grammar;

<aggregate-name> ::= <identifier>

<time> ::= *Time* (<aggregate-name>)

<weight> ::= *Weight* (<cell> , <cell>) |
Weight (<aggregate-name> , <cell> , <cell>)

<cell> ::= *Cell* (<aggregate-name> , <list-of-indices>)

<list-of-indices> ::= <index> |
<index>, <list-of-indices>

<index> ::= <integer>

<output> ::= *Output* (<cell>)

<broadcasting-time> ::= *Broadcasting-time* (<cell>)

<next-wake-up-time> ::= *Next-wake-up-time* (<cell>)

<inputs> ::= *Inputs* (<cell>) [<index>]

However, for the last five, which are access functions to internal variables of a cell, the <cell> argument may be omited when used inside the cell's body.

2.2.3. Features of the language for dynamic description

The "dynamic description features" of the language represent a powerful tool, allowing the description of any A.N.N. dynamic features as, for instance, the update of its weights during execution. We explain these features in the following sections.

a - Aggregates

An aggregates is a set formed by a group of cells which have correlated temporal activities. Its static characteristics may be described the same way than for a group. This has already been described in § 2.2.2..

However an aggregate has various dynamic characteristics which are described in this section.

?-set

The ?-set is a specific slot for synchronous aggregates.

It specifies which cells have to be queried for information when the aggregate's clock is updated. It is specific for synchronous aggregates because, for asynchronous aggregates the ?-set may not be specifies statically and has to be computed at each update of the aggregate's clock.

!-set

The !-set is a specific slot for synchronous aggregates.

It specifies which cells will need to get new information when the aggregate's clock is updated. As for the ?-set, it is specific for synchronous aggregates because, for asynchronous aggregates the !-set may not be specifies statically and has to be computed at each update of the aggregate's clock.

Π-function

This slot contains a piece of code written in C specifying what transformation should be applied to the values coming out of the cells, when transiting through the connector, on their way to the activated cells.

Typically, in numerous A.N.N. models this transformation is a multiplication.

A specific ***Pi*** intermediary variable is available to address this intermediary result.

ΣoΠ-function

The ΣoΠ-function is a specific slot for synchronous aggregates.

It is used to give information to the compiler going from the virtual machine to SMART.

Often, the Π-function and the integration function made by the cell on the incoming information may be combined together and expressed under a form of matrices operations. Typically, for numerous A.N.N. models the ΣoΠ-function is the product of the vector of cells' outputs by the matrix of weights.

The SMART machine, which has been specially designed to be preferment for matrix-matrix and vector-matrix operations, takes direct advantage of such an information.

Stop-conditions

The Stop-condition slot described how an activated aggregate may decide to stop.

A specific keyword **synchronous-mode** is proposed to described the standard stop condition of a synchronous aggregate.

Otherwise, a piece of code is required, computing some Boolean expression. When this expression evaluates to true, it means that the stop conditions are met and that the activated aggregate should stop.

Λ-program

Finally, the Λ-program is a piece of C code describing the learning algorithm of the aggregate.

Typically, it takes the weight matrix as an input and return a new weight matrix as its output. However, it may modify only a very few weights.

b - Cells

Each type of cells, used in the network, must be described as a procedure to determine its job. This procedure might have several inputs to indicate the aggregates that share the mention cell. The body of the procedure is modular, each module represents the task desired by an aggregate, so we have the possibility the declare global or local variables for the procedure. The code itself is written in C language. This type of procedure is very useful when we define an aggregate because we can assign the value of the variable *Cell*, used by an aggregate, by calling this procedure.

c - Control program

This part is similar to the part "main" in C language, in other word, it represents the motor of the simulation. All the initializations of the network, data acquisition, monitoring, etc ... can be describe in this part.

3. THE VIRTUAL MACHINE

3.1. Instructions set of the virtual machine

The virtual machine has only two specific instructions: **Activate** and **Learn**.

Activate is used to order the activation of an aggregate. Depending of the nature of the target machine the compiler will either generate C-vectoriel instructions for a synchronous activation of the aggregate on SMART or NEURAL or Occam instructions for an asynchronous activation of the aggregate on SuperNode.

The synchronous and asynchronous activation of aggregates have been described in principle in § 2.1.3. and 2.1.4.. They will be describe in some details, just bellow, in § 3.3. and § 3.4.. Section 3.4. proposes Occam harness for the needed processes, the compiler from the virtual machine to SuperNode having just to fill the empty slots of these harness.

Learn is used to order the operation of learning on an aggregate. It consists, essentially in executing the C program written by the programmer in the "Λ-proram" slot of the aggregate description. Typically, this program takes the weight matrix as an input and return a new weight matrix as its' output. Any kind of operation on this matrix is possible. Internal variables of the aggregates, used as "memory traces" of what went on during the previous activation steps may be of great influence on the kind of modifications applied to the weight matrix.

The program of the virtual machine appears in the MENTAL code as the Control() segment of code.

Beside the Activate and Learn instructions, it is possible to access and modify any part of the data structured representing aggregates and cells. However, those modifications may occur only at some allowed interuption points of the program, where it may be guarantied that no concerned aggregates are active. For instance, no such modifications may occur in a PAR constructor (described in next paragraph). These access and modifications are done using a usual C syntax and the provided access function to the data structure.

Some special constructors like, for instance, the PAR constructor may be used. The PAR constructor allows the parallel activation of different aggregates. Its' semantic is the same than the Occam' PAR constructor or than the parallel ANSI C PAR.

3.2. Grammar

This grammar is a sub-set of the usual ANSI C grammar augmented by few specific items.

Explanations and remarks about used notation for the grammar:

- <, >, ::=, [,], {, }, | are the symbols of the BNF meta-language with their usual meanings;
- Note that [,], {, } may be used both as symbols of the meta-language (not underlined) or as terminal symbols of the described language (underlined);
- <Foo>: means Foo is a non terminal symbol of the grammar;
- Underline words are terminal symbols of the grammar;
- Bold expressions are part if the grammar that differ from usual C language grammar;

<virtual-machine-program> ::= <VM-declarator><VM-body>

<VM-declarator> ::= **Control** ([<parameter-list>])

<parameter-list> ::= <identifier> |
<identifier> , <parameter-list>

<identifier> Syntactically defined

<VM-body> ::= { [<declaration-list>]<statement-list> }

Statements

<compound-statement> ::= { [<declaration-list>][<statement-list>] }

<declaration-list> ::= <declaration> |
<declaration><declaration-list>

/* Only simple declarations will be allowed inside the virtual machine program. The part of the grammar defining precisely what declarations are allowed is not written yet.*/

<statement-list> ::= <statement> |
<statement><statement-list>

<statement> ::= <compound-statement> |
<expression> ; |
if (<expression>) <statement> |
if (<expression>) <statement> else <statement> |
while (<expression>) <statement> |
do <statement> while (<expression>) ; |
for ([<expression>] ; [<expression>] ; [<expression>])
<statement> |
; |
par |
par ([<expression>] ; [<expression>] ; [<expression>])
<statement>

Expressions

<expression> ::= <primary> |
* <expression> |
& <expression> |
- <expression> |
! <expression> |
~ <expression> |
++ <lvalue> |
- - <lvalue> |
<lvalue> ++ |
<lvalue> - - |
sizeof <expression> |
(<type-name>) <expression> |
<expression><binop><expression>|
<expression> ? <expression> : <expression> |
<lvalue><asgnop><expression> |
<expression> , <expression>

<primary> ::= <identifier> |
<constant> |
<string> |
(<expression>) |
<primary> ([<expression-list>]) |
<primary> [<expression>] |
<lvalue> . <identifier> |
<primary> -> <identifier>

<lvalue> ::= <identifier> |
<primary> [<expression>] |
<lvalue> . <identifier> |
<primary> -> <identifier> |

```
                                    * <expression> |
                                    ( <lvalue> )

<binop>                  ::=        * | / | % | + | - | >> | << | < | > | <= | >= | == | != |
                                    & | ^ | | | && | || | ?:

<asgnop>                 ::=        = | += | -= | *= | /= | %= | >>= | <<= | &= | ^= | |=
```

3.3. Activation of synchronous aggregates

A synchronous aggregate corresponds roughly to a data structure and several procedures. The data structure gather all data used by the aggregate in a formal way; in this case all the access functions can share the data which can be weight matrix, input, output, etc ... The procedures represent all the tasks required when an activation is took place; they are obtained from the description of the aggregates. Three major procedures are expected, for learning activating and for the cell's tasks, when an aggregate has several types of cells it is possible to have several procedures for their tasks. The control part describes the synchronization between the different procedures to perform desired goal of the network.

According to SMART's architecture, the data structure must be designed in the shape of matrix and vectors; so at each activation we have to prepare the right data for the concerning aggregate. While the procedures will be written in C vectorial-language and during the execution the pipe-lines will be filled by the procedures instructions according the control part and the synchronization is determined by the order of the instruction in the pipe-lines.

3.4. Activation of asynchronous aggregates

The Occam write of cell, aggregate and control skeleton processes is a first step to a compiler. All kind of cells, aggregate and control processes can be put into this skeleton.

The cell process receives from one or many aggregates an activation message. The cell executes one part of its behavior linked to one of the active aggregate. The common part is protected by Booleans which determine if the condition of activation of the common part is fulfil. If then the common part is executed and the aggregates are 'released' by the reception of the new state of the cell and many other informations like their next wake up time.

Three ways in communications, three types of communications :

control to aggregates:	control.aggregate
aggregate to cells :	aggregate.cell
cell to aggregates :	cell.aggregate

The control.aggregate connector is a vector of channels linking the control process to all the aggregates. The aggregate.cell connector is a kind of matrix of channels linking all the aggregates to all the cells. The last type cell.aggregate allow to connect all the cells toward all the aggregates. Each aggregate and each cell is denoted by an unique number calculated at the instantiation, as this is the case in ANSI C.

```
PROC cell ( me : INT , CA : part.of.cell.aggregate , AC : part.of.aggregate.cell )
        < declaration of common data of the cell>
        SEQ
                <Initialisation of the cell body>
                <Determination of the aggregate vector which located the aggregates whom this cell depend>
                ALT   <Among the aggregates whom this cell depend>
                <the identifier i varies from 1 to n , with n the number of aggregates linked to this cell>
                        AC[ aggregate[i] ]? activation code , data ,  …
                                <process of the cell attached at the aggregate N° aggregate[i]>
                < IF
```

```
                    condition
                        common process
                    TRUE
                        SKIP>
                PAR <for all the aggregates which have activated this cell>
                    CA [aggregate[i] ]! result, msg, time, …
: -- End of Cell procedure
```

The aggregate process receives messages controling its data flow and its state (learn, recognize, …). Once activated it interacts with cells.

```
PROC aggregate (    control : part.of.control.aggregate,
                    send : part.of.aggregate.cell , rec : part.of.cell.aggregate)
        < declarations of aggregate's structures : table of hours … >
        SEQ
                < Initialisation of structures : weights , … , continue := TRUE>
                msg  := code activate aggregate in recognize mode
                WHILE continue DO
                        SEQ
                                IF      sensible <=0
                                                control? msg
                                        TRUE
                                                sensible := sensible  - 1
                                CASE code(msg) OF
                                        code0   :       aggregate.active( send , rec )
                                                        …
                                        code.stop       :       arret.aggregate()
: -- End of Aggregate procedure
```

Lets take a look to the aggregate.active(send , rec) procedure :

```
PROC aggregate.active ( send : part.of.aggregate.cell , rec : part.of.cell.aggregate )
        SEQ
                <Determine active cells>
                PAR for i designing cells to activate
                        send [ cell[i] ]! activation, data, …
                -- Get result, next-wake-up-time, etc…
                PAR for i designing cells to activate
                        rec [ cellules[i] ]? result [ cell[i] ]
                <Set up informations>
:End of Aggregate.active procedure
```

The control process sends to the aggregate processes all informations concerning their activation, the flow management and the number of execution steps.

```
PROC control ( c : control.aggregate )
        <Data declaration … >
        SEQ
                <Initialisation>
                PAR ( or SEQ ) <Replicator>
                        -- d can be data for a clamp
```

```
                -- or correspond to activation of aggregates like :
                -- activation code , forward , 1000 tours .
                c [ aggregate [ i ] ] ! d
        PAR <Replicator>
                …
: -- End of Control procedure
```

The deadlock free implementation is guarantied by the fact that cells can not be involved in a deadlock circuit. The control process must wait the aggregate processes, the aggregate processes must wait the cell processes, but when a cell process want to send a message it is always after the insurance that the receiver, the aggregate process, is ready to accept the communication. Therefore there is no deadlock.

3.5. From virtual machine to target machines

3.5.1. Toward SMART

The compilation from the virtual machine to SMART has to respect its approach, since it is designed to deal with sparse matrices.

SMART's approach infers matrices from the object oriented description. It simply groups *synapses* in clusters as large as possible, according to the various dynamics and adaptation rules. Hence we get a small set of connection matrices, presenting of course a sparse topology. Then *neurons* are grouped according to their activation laws. If we get too many matrices or groups, there is a design problem. Some reductions must be performed. For instance, in order to get different *neural* responses let us set a function with one extra argument to select the suitable response. Thus a set of very large tasks is defined. We might call them sequentially but each task is easy to parallelize since it consists of performing the same operation many times on independent data.

3.5.2. Toward SuperNode

The compiler from the virtual machine to SuperNode will be rather simple.

SuperNode is based on C.S.P. programming model (Communicating Sequential Processes) [Hoare85]. Cells and connectors are naturally implemented as processes.

Connectors have been designed in such a way that for an aggregate of N cells, N links are needed, one from each cell to the aggregate. With these N links any topology of connections, even full connectivity of N^2 links, may be emulated. Of course, this saving of links is done to the detriment of the length of the exchanged messages. This choice was made deliberately due to the inherent preference of few long communications to many short ones in SuperNode architectures.

Each cell is compiled as a process designed on the model given in Figure 11 (for cells belonging to only one aggregate) and Figure 12 (for cells belonging to several aggregates). This task is easy for the compiler, because a cell may be seen as a harness written in Occam, having some empty slots (like the "program slot" and the "declared variables slot") that only have to be filled according to what is expressed in the high level language.

Connectors are a more difficult matter. Connectors are responsible for information exchanges, synchronization, and probing and activating cells. All connectors run in parallel. Cells belonging to several aggregates impose some implicit synchronisation between the different aggregates. Here again a kind of Occam harness for connectors may be written, however, the compiler has a much more difficult task in filling the empty slots than in the cells case.

4. THE A.N.N. DESCRIPTION LANGUAGE

4.1. Instructions' set of the A.N.N. description language

The instructions set of the A.N.N. description language is precisely defined in the grammar proposed in next section (§ 4.2.).

The instructions set covers all needs to describe the different data structures of cells, aggregates and connectors. It also provides all the predfined access functions to this data structures. Finally, it includes some usual C items.

4.2. Grammar

Explanations and remarks about used notation for the grammar:

- <, >, ::=, [,], {, }, | are the symbols of the BNF meta-language with their usual meanings;
- Note that [,], {, } may be used both as symbols of the meta-language (not underlined) or as terminal symbols of the described language (underlined);
- <Foo>: means Foo is a non terminal symbol of the grammar;
- Underline words are terminal symbols of the grammar;
- Bold expressions are part if the grammar that differ from usual C language grammar;

Program and external definitions

<program> ::= <external-definition> |
<external-definition><program>

<external-definition> ::= <function-definition> |
<data-definition> |
<cell-definition>

<function-definition> ::= [<type-specifier>]<function-declarator><function-body>

<function-declarator> ::= <declarator> ([<parameter-list>])

<parameter-list> ::= <identifier> |
<identifier> , <parameter-list>

<identifier> Syntaxically defined

<function-body> ::= <type-decl-list><function-statement>

<function-statement> ::= { [<declaration-list>]<statement-list> }

<data-definition> ::= [extern][<type-specifier>][<init-declarator-list>] ; |
[static][<type-specifier>][<init-declarator-list>] ;

<cell-definition> ::= **cell<cell-declarator><cell-body>**

<cell-declarator> ::= <declarator>([<parameter-list>])

<cell-body> ::= <type-decl-list>**<cell-statement>**

<cell-statement> ::= { [<declaration-list>]<statement-list>**<cell-sub-statement-list>**
<statement-list> }

<cell-sub-statement-list> ::= **<cell-sub-statement>** |
<cell-sub-statement><cell-sub-statement-list>

<cell-sub-statement> ::= **respectively-to-aggregate:**<identifier>
{ [<declaration-list>]<statement-list> }

Statements

<compound-statement> ::= { [<declaration-list>][<statement-list>] }

<declaration-list> ::= <declaration> |
<declaration><declaration-list>

<statement-list> ::= <statement> |
<statement><statement-list>

<statement> ::= <compound-statement> |
<expression> ; |
if (<expression>) <statement> |
if (<expression>) <statement> else <statement> |
while (<expression>) <statement> |
do <statement> while (<expression>) ; |
for ([<expression>] ; [<expression>] ; [<expression>])
<statement> |
switch (<expression>) <statement> |
case <constant-expression> : <statement> |
default : <statement> |
break ; |
continue ; |
return ; |
return <expression> ; |
goto <identifier> ; |
<identifier> : <statement> |
; |
wait-for <expression> ; |
par |
par ([<expression>] ; [<expression>] ; [<expression>])
<statement>

Expressions

<expression> ::= <primary> |
* <expression> |
& <expression> |
- <expression> |
! <expression> |
~ <expression> |
++ <lvalue> |
- - <lvalue> |
<lvalue> ++ |
<lvalue> - - |
sizeof <expression> |
(<type-name>) <expression> |
<expression><binop><expression>|
<expression> ? <expression> : <expression> |
<lvalue><asgnop><expression> |

<expression> , <expression>

<primary> ::= <identifier> |
<constant> |
<string> |
(<expression>) |
<primary> ([<expression-list>]) |
<primary> [<expression>] |
<lvalue> . <identifier> |
<primary> -> <identifier>

<lvalue> ::= <identifier> |
<primary> [<expression>] |
<lvalue> . <identifier> |
<primary> -> <identifier> |
* <expression> |
(<lvalue>)

<binop> ::= * | / | % | + | - | >> | << | < | > | <= | >= | == | != |
& | ^ | | | && | || | ?:

<asgnop> ::= = | += | -= | *= | /= | %= | >>= | <<= | &= | ^= | |=

Declarations

<declaration> ::= <decl-specifiers>[<init-declarator-list>] ;

<decl-specifiers> ::= <type-specifier>[<decl-specifiers>] |
<sc-specifier>[<decl-specifiers>]

<type-specifier> ::= char | short | int | long | unsigned | float | double |
<struct-or-union-specifier> | <typedef-name> |
netdef<goup-or-aggr-specifiers>

<struct-or-union-specifier> ::= struct { <struct-decl-list> } |
struct <identifier> { <struct-decl-list> } |
struct <identifier> |
union { <struct-decl-list> } |
union <identifier> { <struct-decl-list> } |
union <identifier>

<struct-decl-list> ::= <struct-declaration> |
<struct-declaration><struct-decl-list>

<struct-declaration> ::= <type-specifier><struct-declarator-list> ;

<struct-declarator-list> ::= <struct-declarator> |
<struct-declarator> , <struct-declarator-list>

<struct-declarator> ::= <declarator> |
<declarator> : <constant-expression> |
: <constant-expression>

<declarator> ::= <identifier> |
(<declarator>) |
* <declarator> |
<declarator> () |
<declarator> [[<constant-expression>]]

<constant-expression> Syntaxically defined

<typdef-name> ::= <identifier>

<group-or-aggr-specifier> ::= **aggregate { <aggregate-decl-list> }** |
aggregate <identifier> { <aggregate-decl-list> } |
aggregate <identifier> |
group { <group-decl-list> } |
group <identifier> { <group-decl-list> } |
group <identifier>

<aggregate-decl-list> ::= **Shape: <shape-spec>** ;
[**Paving:**][**<paving-spec>**][;]
Cells: <cells-spec> ;
Neiborhood: <neighborhood-spec> ;
Connectivity: <connectivity-spec> ;
[**?-set:**] [**<?-set-spec>**][;]
[**!-set:**] [**<!-set-spec>**][;]
Π-function: <Π-function-spec> ;
[**ΣoΠ-function:**] [**<ΣoΠ-function-spec>**][;]
Stop-condition: <stop-condition-spec> ;
Λ-program: <Λ-programm-spec> ;

<Shape-spec> ::= **<shape>of-size:**<expression> |
part-of<identifier>**of-size:**<expression> |
in-layers (<list-of-identifiers>) |
<same-as-statement>

<shape> ::= **line** | **plane** | **hyperplane** | **torus**

<list-of-identifiers> ::= <identifier> |
<identifier> , **<list-of-identifiers>**

<same-as-statement> ::= **same-as**<identifier>

<paving-spec> ::= **square** | **hexagonal** |
<same-as-statement>

<cells-spec> ::= <compound-statement>

<neighborhood-spec> ::= **all** | **none** | **8-nearest-neighbors** |
full-forward-from <identifier> **to** <identifier> |
<same-as-statement>

<connectivity-spec> ::= <compound-statement>

<?-set-spec> ::= <identifier>

<!-set-spec> ::= <identifier>

<Π-function-spec> ::= <compound-statement>

<stop-condition-spec> ::= **synchronous-mode** | <compound-statement>

<Λ-programm-spec> ::= <compound-statement>

<group-decl-list> ::= **Shape: <shape-spec> ;**
[**Paving:**][**<paving-spec>**][**;**]
Cells: <cells-spec> ;
Neiborhood: <neighborhood-spec> ;
Connectivity: <connectivity-spec> ;

<sc-specifier> ::= auto | static | extern | register | typedef

<init-declarator-list> ::= <init-declarator> |
<init-declarator> , <init-declarator-list>

<init-declrarator> ::= <declarator>[<initializer>]

<initializer> ::= = <expression> |
= { <initializer-list> } |
= { <initializer-list> , }

<initializer-list> ::= <expression> |
<initializer-list> , <initializer-list> |
{ <initializer-list> }

<type-name> ::= <type-specifier><abstract-declarator>

<abstract-declarator> ::= <empty> |
(<abstract-declarator>) |
* <abstract-declarator> |
<abstract-declarator> () |
<abstract-declarator> [[<constant-expression>]]

4.3. Game of life

4.3.1. Description of the problem

One of the most well-known examples of C.A. (Cellular Automata) is the game of life. This model is very simple. We choose to implement it as a torus of units (NxM) with a square paving (see figure 2.7.); each unit is connected to its eight nearest neighbors with fixed weights (equal to one). There is no learning phase.

The updating rules of the cells say that:

- if a cell has more (strictly) than three alive (output = 1) neighbors, it dies from over population;
- if a cell has less (strictly) than two alive neighbors, it dies from loneliness;
- if a cell has two alive neighbors, it will stay in its' present state at next generation;
- if a cell has three alive neighbors, it will be alive at next generation.

In order to program this model, we consider that the units form one aggregate S with fixed !-set and ?-set equal to S, because all the units are input and output at the same time. The cells' program describes the rules of the

game, while the control program activates the aggregate S the required number of times. The initial values of the units, given by the programmer, correspond to the state at first generation.

Two different implementations of the game of life are proposed in the next two sections.

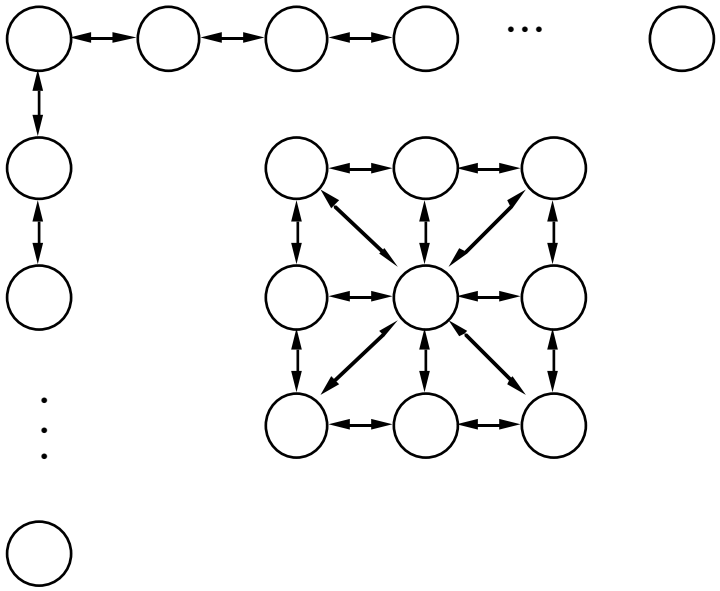

Figure 2.7.

4.3.2. MENTAL program (first instance)

In this first instance of the life game program all the cells are grouped within a single aggregate.

Remarks about the programs notations: In **BOLD** appears the special feature of MENTAL and in *ITALICS* appears the name of the predefined variables and access functions of MENTAL

```
##############################################################################################

/*Inclusions*/

#Include <MENTAL.h>
/*Necessary library for any MENTAL program*/

/*Definitions*/

#define     N     60
#define     M     N

/*AGGREGATES AND CELLS DESCRIPTIONS*/

netdef aggregate  S
/* Definition of aggregate S */
{
      Shape:          torus of-size: N, M;
      Paving:         square;
      Cells:          {
                        Cell (S, $x, $y) = Life-Cell(S);
                        /* This means that the cell of aggregate S are of type Life-Cell */
```

```
                        /*The $x and $y are variables that may be matched to any fitting indices.
                          Here "Cell (S, $x, $y) = Life-Cell(S);"
                          stands for: "for(h1=0;h1<N;h1++)
                                          for(h2=0;h2<M;h2++)
                                              Cell (S,h1,h2) = Life-Cell(S);"
                        */
                    }
    Neighborhood:   8-nearest-neighbors;
    Connectivity:   {
                        complete;
                        symetric;
                    }
    ?-set:          S;
    !-set:          S;
    Π-function:     {
                        Pi  ($!x, $?y) = O ($!x);
                        /*The function Pi takes two cells as its' arguments. The first one should be
                          a !-cell and the second one a ?-cell. The function Pi specifies how
                          information incoming from a ?-cell is transformed by the connector on
                          its' way to a !-cell. For this particular instance, there is no
                          transformation.
                        */
                    }
    ΣoΠ-function:   I(!) = W(!,?) X O(?);
                    /*The ΣoΠ-function specifies in matrix notation how information is modified by
                      connectors. This information is a faculative synthesis of the information given by
                      the Pi function and by part of the cells' program. This information is, however, a
                      usefull clue for the compiler toward SMART wich takes direct advantages of
                      matrices operations.
                    */
    Stop-condition: synchronous-mode  ;
    /* The synchronous-mode Stop conditions means that the aggregate activation is stopped after one cycle*/
}

cell  Life-Cell ( S )
    aggregate S; /* Definition of the type of cell "Life-Cell" */
{
    Respectivly-to-aggregate  S
        {
            int h1;
            int input = 0;
            for(h1=0;h1<Nber-of-inputs,h1++)
                input = input + Inputs [h1];
            /*Summation of the inputs*/
            if(input > 3)
                Output  = 0;
            else    if(input < 2)
                        Output  = 0;
                    else    if(input = 3)
                                Output  = 1;
            /*Usual game of life decision rule*/
            Broadcasting-time  = Broadcsating-time  + 1;
            Next-wake-up-time  = Next-wake-up-time  + 1;
```

```
                    /*Usual updates of Broadcasting-time and Next-wake-up-time varariables to simulate a
                      synchronous mode on an asynchronous machine.
                    */
            }
}

Control()
{
        int h1;
        Output (Cell (S, $x, $y)) = 0;
        /*Outputs of all cells are set to 0*/
        ….
        /*Initializations corresponding to state of first generation*/
        Weights (S, $x, $y) = 1;
        /*All weights are set to 1*/
        /*Weights are never used but, however, they are usefull to specify connection, non zero weight meaning
        that the connection exits*/
        for(h1=0,h1<1000;h1++)
                Activate (S);
        /*Execution of 1000 generation*/
}
##########################################################################################
```

4.3.3. MENTAL Program (second instance)

In this second instance of the life game program, the 3 600 cells are distributed over 16 aggregates. This has been done thinking that could be the good way to write the program for an execution on a 16 Transputers Super-Node, placing one aggregate on each Transputer.

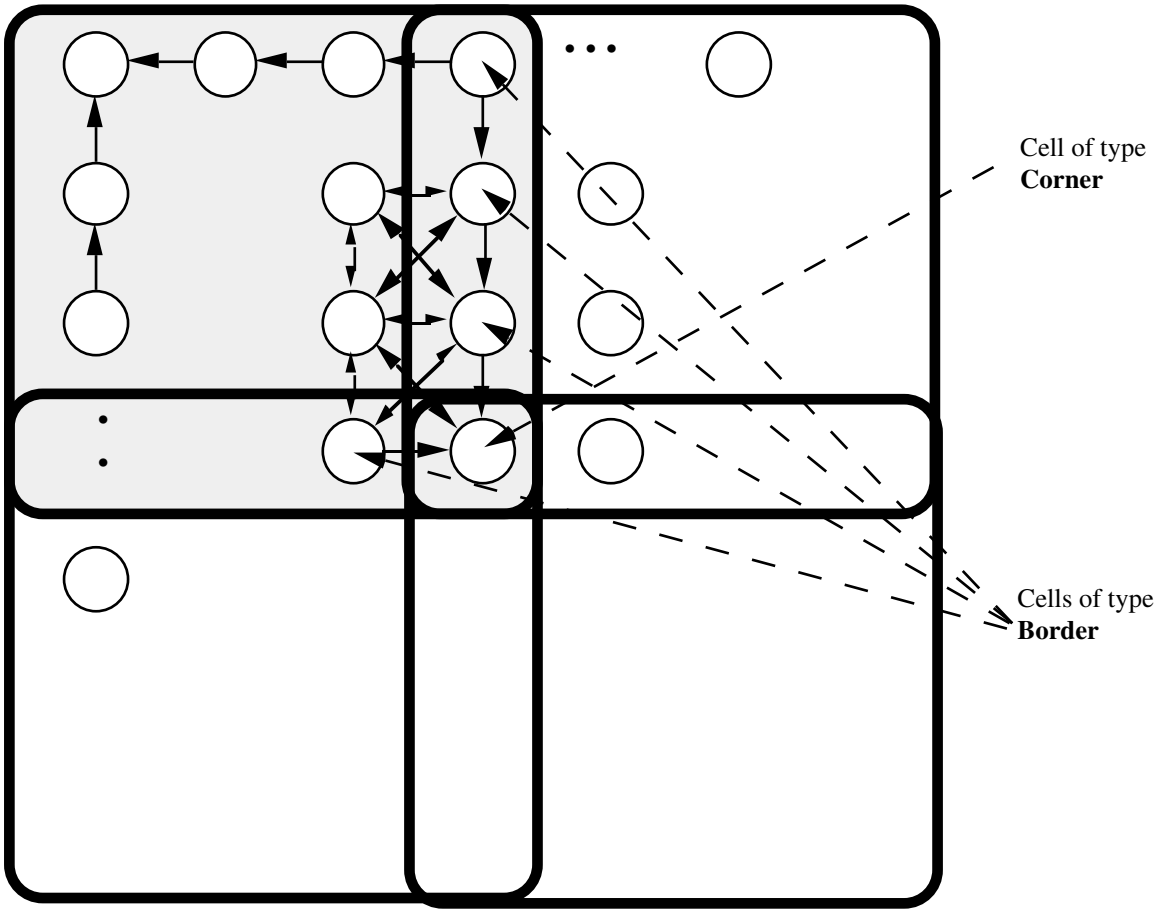

Figure 2.8.

```
##########################################################################################

/*Inclusions*/

#Include <MENTAL.h>

/*Definitions*/

#define     N     60
#define     M     N
#define     p     ((N / 4) + 1)

/*AGGREGATES AND CELLS DESCRIPTIONS*/

netdef group  S
{
      Shape:          torus  of-size:  N,  M;
      Paving:         square;
      Cells:          {
                            Cell (S, $x, $y) = Cell (C[$x / p][$y / p], $x % 4, $y % 4);
                            /*Mapping of cells of group S to cells of the array of aggregates C.*/
                      }
```

```
        Neighborhood:   8-nearest-neighbors;
}

netdef aggregate C[4][4] /* Array of aggregat */
{
        Shape:          part-of S of-size: N/4 + 1, M/4 + 1;
        Paving:         same-as S;
        Cells:          {
                                int h1, h2, h3, h4;
                                for(h1=0;h1<4;h1++)
                                        for(h2=0;h2<4;h2++)
                                                /*All cells are "standard" by default*/
                                                Cell (C[h1][h2], $x, $y) = Life-Cell-Standard(C[h1][h2]);
                                for(h1=0;h1<4;h1++)
                                        for(h2=0;h2<4;h2++)
                                                /*Bottom and right rows cells are of type "border"*/
                                                {
                                                        for(h3=0;h3<M/4;h3++)
                                                                /*But some of them are replaced */
                                                                /* by the other function Life-Cell-Border */
                                                                Cell (C[h1][h2], N/4, h3) =
                                                                Life-Cell-Border(C[h1][h2],C[h1+1][h2]);
                                                        for(h4=0;h4<N/4;h4++)
                                                                Cell (C[h1][h2], h4, M/4) =
                                                                Life-Cell-Border(C[h1][h2],C[h1][h2+1]);
                                                }
                                /*Bottom right cell is of type "corner"*/
                                for(h1=0;h1<4;h1++)
                                        for(h2=0;h2<4;h2++)
                                                Cell (C[h1][h2],N/4,M/4) =
                                                Life-Cell-Corner(C[h1][h2],C[h1][h2+1],
                                                                C[h1+1][h2],C[h1+1][h2+1]);
                        }
        Connectivity:   {
                                int h1, h2, h3, h4;
                                complete;
                                symetric;
                                for(h1=0;h1<4;h1++)
                                        for(h2=0;h2<4;h2++)
                                                {
                                                /* Destruction of some connections to prevent multiple
                                                connections between cells (see figure 2.8.)       */
                                                        for(h3=1;h3<M/4 + 1;h3++)
                                                                {
                                                                        disconnect(Cell (C[h1][h2], N/4, h3),
                                                                                        Cell (C[h1][h2], N/4, h3 - 1);
                                                                        disconnect(Cell (C[h1][h2], 0, h3),
                                                                                        Cell (C[h1][h2], 0, h3 + 1);
                                                                }
                                                        for(h4=1;h4<N/4 +1;h4++)
                                                                {
                                                                        disconnect(Cell (C[h1][h2], h4, M/4),
                                                                                        Cell (C[h1][h2], h4-1, M/4);
                                                                        disconnect(Cell (C[h1][h2], h4, 0),
```

```
                                                        Cell (C[h1][h2], h4+1, 0);
                                                }
                                        }

                        }
        ?-set:          C[$x / p][$y / p];
        !-set:          C[$x / p][$y / p];
        Π-function:     {
                                Pi  ($!x, $?y) = O ($!x);
                        }
        ΣoΠ-function:   I(!) = W(!,?) X O(?);
        Stop-condition: synchronous-mode;
}

cell    Life-Cell-Standard (A1)
        aggregates  A1;
{
        Respectivly-to-aggregate   A 1
                {
                        int h1;
                        int input = 0;
                        for(h1=0;h1<Nber-of-inputs,h1++)
                                input = input + Inputs [h1];
                        if(input > 3)
                                Output  = 0;
                        else    if(input < 2)
                                        Output  = 0;
                                else    if(input = 3)
                                                Output  = 1;
                        Broadcasting-time  = Broadcsating-time  + 1;
                        Next-wake-up-time  = Next-wake-up-time  + 1;
                }
}

cell    Life-Cell-Border (A1 , A2)
        aggegates  A1,  A2;
{
        int input;
        wait-for Active.A1   && Active.A2 ;
        /*A cell of type border will be activated only when its' inputs from both aggregates A1 and A2 are
          availabale.
        */
        Respectivly-to-aggregate   A 1
                {
                        int h1;
                        int input = 0;
                        for(h1=0;h1<Nber-of-inputs,h1++)
                                input = input + Inputs [h1];
                        Broadcasting-time  = Broadcsating-time  + 1;
                        Next-wake-up-time  = Next-wake-up-time  + 1;
                }
        Respectivly-to-aggregate   A 2
                {
                        int h1;
```

```
                int input = 0;
                for(h1=0;h1<Nber-of-inputs,h1++)
                        input = input + Inputs [h1];
                Broadcasting-time = Broadcsating-time + 1;
                Next-wake-up-time = Next-wake-up-time + 1;
        }
    input = input.A1 + input.A2;
    if(input > 3)
        Output = 0;
    else  if(input < 2)
                Output = 0;
        else  if(input = 3)
                        Output = 1;
}

cell  Life-Cell-Corner (A1, A2, A3, A4)
    aggegates A1, A2, A3, A4;
{
    int input;
    Respectivly-to-aggregate  A 1
        {
            int h1;
            int input = 0;
            for(h1=0;h1<Nber-of-inputs,h1++)
                    input = input + Inputs [h1];
            Broadcasting-time = Broadcsating-time + 1;
            Next-wake-up-time = Next-wake-up-time + 1;
        }
    Respectivly-to-aggregate  A 2
        {
            int h1;
            int input = 0;
            for(h1=0;h1<Nber-of-inputs,h1++)
                    input = input + Inputs [h1];
            Broadcasting-time = Broadcsating-time + 1;
            Next-wake-up-time = Next-wake-up-time + 1;
        }
    Respectivly-to-aggregate  A 3
        {
            int h1;
            int input = 0;
            for(h1=0;h1<Nber-of-inputs,h1++)
                    input = input + Inputs [h1];
            Broadcasting-time = Broadcsating-time + 1;
            Next-wake-up-time = Next-wake-up-time + 1;
        }
    Respectivly-to-aggregate  A 4
        {
            int h1;
            int input = 0;
            for(h1=0;h1<Nber-of-inputs,h1++)
                    input = input + Inputs [h1];
            Broadcasting-time = Broadcsating-time + 1;
            Next-wake-up-time = Next-wake-up-time + 1;
```

```
            }
        wait-for Active.A1  && Active.A2  && Active.A3 && Active.A4;
            {
                input = input.A1 + input.A2 + input.A3 + input.A4;
                if(input > 3)
                Output  = 0;
                else    if(input < 2)
                            Output  = 0;
                        else    if(input = 3)
                                    Output  = 1;
            }
}

Control()
{
        int h1,h2,h3;
        Output  (Cell  (S, $x, $y)) = 0;
        /*Outputs of all cells are set to 0*/
        ….
        /*Initializations corresponding to state of first generation*/
        Weights  (S, $x, $y) = 1;
        /*All weights are set to 1*/
        for(h1=0,h1<1000;h1++)
            par(h2=0;h2<N/4;h2++)
                par(h3=0;h3<M/4;h3++)
                    Activate  (C[h2][h3]);
            /*Parallel activation of the 16 aggregate of the array of aggregate C[4][4].*/
}
```

##

4.4. Hoppfield

4.4.1. Description of the problem

The Hopfield net is normally used with binary inputs. This net is most appropriate when exact binary representations are possible, but it is less appropriate when input values are actually continuous, because a fundamental representation problem must be addressed to convert the analog qualities to binary values .

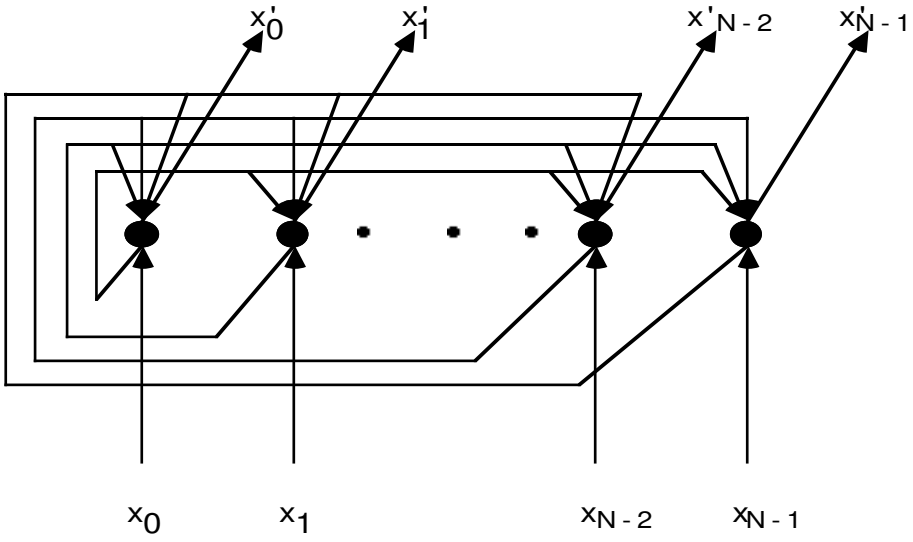

Figure 2.9.

This net, shown in Figure 2.9., can be used as an associative memory or to solve optimization's problems. It has N nodes containing hard limiting, non-linearities, and binary inputs and outputs taking on the values +1 and -1. The output of each node is fed back to all other nodes via weights denoted w_{ij}. The operation of this net is described in the following, first weights are set using the given recipe from exemplary patterns for all classes $X=\{ x^{(0)}, x^{(1)}, \ldots x^{(M-1)} \}$. For each class $x^{(\alpha)}$ we form the nxn matrix

$$W_\alpha = x^{(\alpha)}(x^{(\alpha)})^T - I_n$$

where I_n denotes n x n identity matrix. Now the Hopfield connection matrix for the set M classes is defined as

$$W = \sum_{\alpha=0}^{M-1} W_\alpha$$

$$W = \sum_{\alpha=0}^{M-1} (x^{(\alpha)})((x^{(\alpha)})^T - I_n)$$

or in other word

$$w_{ij} = \begin{cases} \sum_{k=0}^{M-1} x_i^k x_j^k, & i \neq j \\ 0, & i = j, 0 \leq i, j \leq M-1 \end{cases}$$

where w_{ij} is the connection weight from node i to node j, and x_i^k is element i of the examples for class k. We assume that once W has been calculated, all other information about $x^{(\alpha)}$ will be forgotten. Then an unknown pattern is imposed on the net at time zero by forcing the output of the net to match the unknown pattern. Following this initialization, the net iterates in discrete time steps using a given formula such that

$$X_{n+1} = f\,(W.X_n)$$

The pattern specified by the node outputs after convergence is the net output.

Hopfield (1982) has shown that some relaxation schemes have an associated "potential" or cost function and that the states to which the network converges are local minima of this function. This means that the networks are performing optimization's of a well-defined function. Unfortunately, there is no guarantee that the network will find the best minimum. One possibility is to redefine the problem as finding the local minimum which is closest to the initial state. This is useful if the minima are used to represent "items" in a memory, and the initial states are queries to memory which may contain missing or erroneous information. The network simply finds the minimum that fits the query. Hopfield showed that if the units are symmetrically connected (the weight from the unit *i* to the unit *j* equals to weight from unit *j* to unit *i*) and if the units are updated one at a time, each update reduces (or at worst does not increase) the value of a cost function which he called "energy" because of the analogy with physical systems. Consequently, repeated iterations are guarantied to find an energy minimum. the global energy of the system is defined as

$$E = -\sum_{i<j} w_{ij} s_i s_j + \sum_i \theta_i s_i$$

Where w_{ij} is the strength of connection (synaptic weight) from the *j*th unit to the *i*th unit, s_i the state of the *i*th unit, and θ_i is a threshold.

The updating rule is to switch each unit into whichever of its two states yield the lower total energy given the current states of the other units. Because of the connections are symmetrical, the difference between the energy of the whole system with the *k*th hypothesis -1 and its energy with the *k*th hypothesis +1 can be determined locally by the *k*th unit and is just

$$\Delta E = \sum_i w_{ki} s_i - \theta_k$$

Therefore, the rule for minimizing the energy contributed by a unit is to adopt the state +1 if its total input from the other units exceeds its threshold. This is the familiar rule for binary threshold units.

Hopfield [Hopfield82] and others have proven that this net converges when the weights are symmetric ($w_{ij}=w_{ji}$) and node outputs are updated asyncronously, because if the node outputs are updated syncronously, the net may not be converge to one of the classes or, if it is, the result may not be the nearest class.

4.4.2. MENTAL program

```
##########################################################################################
/*Inclusions*/
```

```
#Include <MENTAL.h>

/*Definitions*/

#define     Ninput              16
#define     time-equilibrium    100


/*Useful miscelanous functions*/

int integer-alea();
int h;

/*AGGREGATES AND CELLS DESCRIPTIONS*/

netdef group      INPUT
      Shape:            Line of-size: Ninput;
      Cells:            Cell (INPUT, $x) = Hopfield_Cell (INPUT)
      Neighborhood:     All;
      Connectivity:     {
                              Complete;
                              Symetric;
                        }


netdef aggregate        NET
      Shape:            In-Layers:
                              Layer 1: INPUT;

      Cells:            Cell (NET , $x) = Hopfield_Cell (INPUT) ;
      Neighborhood:     {
                              Full-forward from INPUT to INPUT;
                        }
      Connectivity:     {
                              Complete;
                              Symetric;
                        }
      Π-function:       Pi  ($x, $y) = Weight  (NET, $x, $y) * Output  ($x);
      Λ-function:       if ($x!=$y) {
                              Weight  (NET, $x, $y) = Weight  (NET, $x, $y) + Input  ($x)*Input  ($x);
                              }
      Stop-condition:   Time  (NET) > time-equilibrium;


Define      Hopfield_Cell       a type of cell
      Aggregates:       A1;
      Respectivly to aggregate A1:
            {
                  if(Input>=0)
                        Output  = 1;
                  else
                        Output  = -1;
                  Brodcasting-time  = Time ;
```

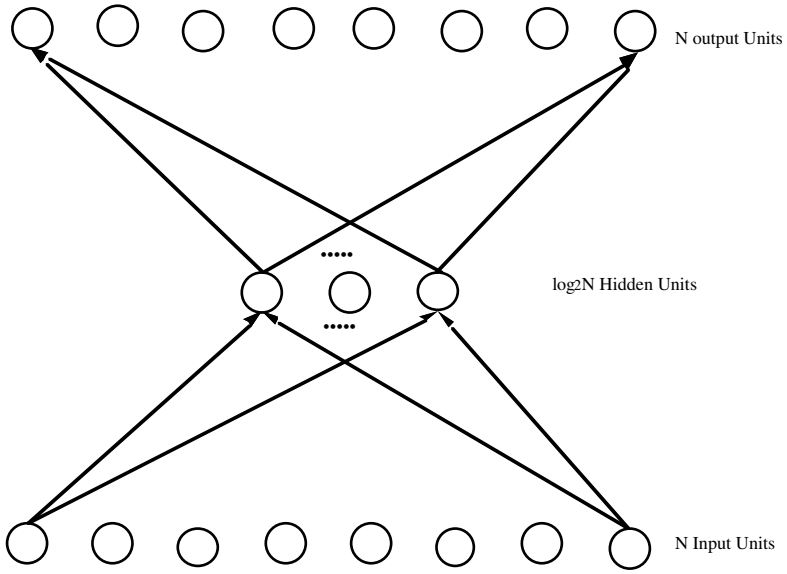

Figure 2.10.

```
                Next-wake-up-time = Time + integer-alea(1, M);
        }

Control()
{
    for(h1=0; h1<Number-of-prototypes; h1++)
        {
            Input (Cell (INPUT, .)) = data(.);
            Learn(NET);
            Activate(NET) ;
}
```

##

4.5. Back-propagation

4.5.1. Description of the problem: the "encoder" problem

As an example of the back-propagation; Ackley, Hinton, and Sejnowski [Ackley85] have posed a problem in which a set of orthogonal input patterns are mapped to a set of orthogonal output patterns through a small set of hidden units. In such cases the internal representations of the patterns on the hidden units must be rather efficient. suppose that we attempt to map N input patterns onto N output patterns. Suppose further that $\log_2 N$ hidden units are provided. In this case, we expect that the system will learn to use the hidden units to form a binary code with a distinct binary pattern for each of the N input patterns. Figure 2.10. illustrates the basic architecture for the encoder problem. Essentially, the problem is to learn an encoding of an N bit pattern into a $\log_2 N$ bit pattern and then learn to decode this representation into the output pattern.

The description of this model, using our programming language, is given in the following. The network *NET* is composed of three groups *INPUT*, *HIDDEN*, and *OUTPUT*, in which they form two aggregates I->H, and H-

>O. Each aggregate has its own definition of the units' shape, paving, neighborhood, !-set, ?-set, and two programs for the learning phase and the activation phase. Weights between units are initialized by random values and stored in matrices according to the aggregates.

The control part describes the two possible phases of the model *learning* and *recognition*. In the learning phase we clamp the input layer and the output layer with data, then we activate I->H and H->O in forward direction to calculate the state of the hidden units, and the state of the output units. The next step is to activate I->H and H->O in backward direction in order to propagate the errors, and to modify the weights by activating the learn procedure for each aggregate. These operations must be repeated many times to achieve the stability. The recognition phase clamps the input layer with data then activates I->H and H->O to calculate the state of the output layer.

4.5.2. MENTAL program

```
##########################################################################################

/*Inclusions*/

#Include <MENTAL.h>

/*Definitions*/

#define        Eta             0.5

/*Useful miscellanies functions*/

int Number-of-hidden-units()
/*Compute the integer immediately superior to the log2 of its argument*/

float alea()
/*Compute a random real value between its two arguments*/

/*AGGREGATES AND CELLS DESCRIPTIONS*/

netdef group INPUT
{
      Shape:          line of-size: Ninput;
      Cells:          {
                            Cell (INPUT, $x) = Input-Cell(I->H);
                      }
      Neighborhood:   none;
}

netdef group OUTPUT
{
      Shape:          line of-size: Noutput;
      Cells:          {
                            Cell (OUTPUT, $x) = Output-Cell(H->O);
                      }
      Neighborhood:   none;
}

netdef group HIDDEN
```

```
{
        Shape:              line of-size: Number-of-hidden-units(Ninput);
        Cells:              {
                                Cell (HIDDEN, $x) = Hidden-Cell(I->H, O->H);
                            }
        Neighborhood:       none;
}

netdef aggregate I->H
{
        Shape:              in-layers ( INPUT, HIDDEN );
        Neighborhood:       full-forward from INPUT to HIDDEN;
        Connectivity:       {
                                Complete;
                            }
        ?-set:              INPUT;
        !-set:              HIDDEN;
        Π-function:         {
                                Pi  ($!x, $?y) = Weight  (I->H, $!x, $?y) * Output  ($!x);
                            }
        ΣoΠ-function:       I (!) = W (!, ?) X O (!);
        Stop-condition:     synchronous-mode;
        Λ-program:          {
                                Weight  (I->H, $?x, $!y) = Weight  (I->H, $?x, $!y) +
                                                           (Eta * Output ($?x) * Output ($!y))
                            }
}

netdef aggregate H->I
{
        Shape:              in-layers ( HIDDEN, INPUT );
        Neighborhood:       Same-as I->H;
        Connectivity:       Same-as I->H;
        ?-set:              HIDDEN;
        !-set:              INPUT;
        Π-function:         {
                                Pi  ($!x, $?y) = Weight  (I->H, $!x, $?y) * Output  ($!x);
                            }
        ΣoΠ-function:       I (!) = W (!, ?) X O (!);
        Stop-condition:     Synchronous-Mode;
}

netdef aggregate H->O
{
        Shape:              in-layers ( HIDDEN, OUTPUT);
        Neighborhood:       full-forward-from HIDDEN to OUTPUT;
        Connectivity:       {
                                Complete;
                            }
        ?-set:              HIDDEN;
        !-set:              OUTPUT;
        Π-function:         {
                                Pi  ($?x, $!y) = Weight  (H->O, $?x, $!y) * Output  ($?x);
                            }
```

```
        ΣoΠ-function:     I (!) = W (!, ?) X O (!);
        Stop-condition:   synchronous-mode;
        Λ-program:        Weight  (H->O, $?x, $!y) = Weight  (H->O, $?x, $!y) +
                                                        (Eta * Output ($?x) * Output ($!y))
}

netdef aggregate O->H
{
        Shape:            in-layers ( OUTPUT, HIDDEN);
        Neighborhood:     same-as H->O;
        Connectivity:     same-as H->O;
        ?-set:            OUTPUT;
        !-set:            HIDDEN;
        Π-function:       {
                                Pi  ($!x, $?y) = Weight  (H->O, $!x, $?y) * Output  ($!x);
                  }
        ΣoΠ-function:     I (!) = W (!, ?) X O (!);
        Stop-condition:   synchronous-mode;
}

netdef group NET
{
        Shape:            in-layers ( INPUT, HIDDEN, OUTPUT};
}

cell Input-cell ();
/*Input-cell are only memory spots to stock incomming data.*/

cell Hidden-cell ( A1, A2)
        aggregates A1,  A2;
{
        real threshold, previous-output;
        Respectively-to-aggregate  A 1
                {
                        float input = 0.0;
                        int h1;
                        for(h1=0; h1 < Number-of-inputs ; h1++)
                                input = input + Inputs [h1];
                        Output  = 1 / (1 + exp(-input - threshold));
                        previous-output = Output ;
                        Brodcasting-time  = Time ;
                        Next-wake-up-time  = Time  + 4;
                }
        Respectively-to-aggregate  A 2
                {
                        real input = 0;
                        int h1;
                        for(h1=0; h1 < Number-of-inputs ; h1++)
                                input = input + Inputs [h1];
                        Output  = input * previous-output * (1 - previous-output);
                        Brodcasting-time  = Time ;
                        Next-wake-up-time  = Time  + 4;
                }
}
```

```
cell Output-Cell
        aggregates  A1;
{
        real goal;
        Respectively-to-aggregate   A1  :
                {
                        real input = 0;
                        real sate;
                        int h1;
                        for(h1=0; h1 < Number-of-inputs ; h1++)
                                input = input + Inputs [h1];
                        state = 1 / (1 + exp(-input - threshold));
                        Output  = (goal - state) * state * (1 - state);
                        Brodcasting-time  = Time ;
                        Next-wake-up-time  = Time  + 4;
                }
}

Control()
{
        int h1;
        Output (Cell (Net, .)) = 0;
        Weights (I->H, ., .) = alea(0, 1);
        Weights (H->O, ., .) = alea(0, 1);
        threshold(Cell (HIDDEN, .)) = Theta;
        threshold(Cell (OUTPUT, .)) = Theta;
        Time (I->H) = 0;
        Time (H->O) = 0;
        Time (O->H) = 0;
        Time (H->I) = 0;
        Time (Cell (NET, .)) = 0;
        Next-wake-up-time (Cell (HIDDEN, .), I->H) = 1;
        Next-wake-up-time (Cell (OUTPUT, .)) = 2;
        Next-wake-up-time (Cell (HIDDEN, .), O->H) = 3;
        for(h1=0; h1 < Number-of-prototypes; h1++)
                {
                        Output (Cell (INPUT, .)) = data(.);
                        goal(Cell (OUTPUT, .)) = data(.);
                        activate (I->H);
                        activate (H->O);
                        learn (H->O);
                        activate (O->H);
                        learn (I->H);
                }
}
##########################################################################################
```

4.6. Kohonen

4.6.1. Description of the problem

a - Neuromimetic architecture

Let us consider a network with n neurons and p inputs $\mathbf{x_i}$. Each input $\mathbf{x_i}$ arrives on all the neurons. An input $\mathbf{x_i}$ is weighted for the neuron k, with the synaptic coefficient m_{ki}. At the beginning, all the synaptic weights are randomly distributed. More, the network is itself connected through inhibition recursive connections. With these connections, each cell k distributes its signal s_k to cells in its neighbourhood. This distribution is done with positive weights in the closest neighbourhood and with negative weights far from.

These recursive connections process a convolution for the outputs with a particular kernel. The look of this kernel is the classical shape of the "mexican hat". This structure is shown at the figure VII.1. Through these recursive connections, clusters with high activities appear in the network.

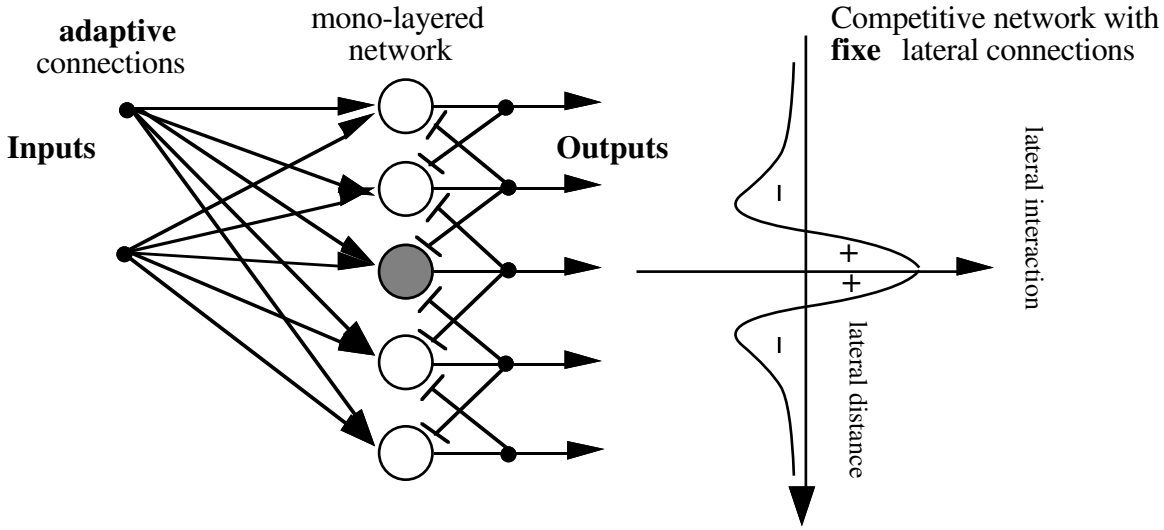

Figure 2.11.: Architecture of the Kohonen's self-organizing network. The weights w_{ij} can evaluate according a learning rule, and the weights of the recursive connections are fixed.

The learning rule for the weights w_{ij} is :

(7.1) $dw_{ij} / dt = \alpha.y_i.(x_j - w_{ij})$.

With this law, the updating is greater as the neuron is active. This is the action of the variable y_i. This updating concerns the neuron i and its neighbourhood whose the matching between the input $\mathbf{x}$ and the synaptic vector $\mathbf{w_i}$ is greater. In this case, the response of this target neuron is maximum.

Before the action of the recursive connections, the outputs are computed by $y_i = \Sigma\ w_{ij}.x_j$. This equation corresponds to the scalar product between the vector $\mathbf{x}$ and the synaptic vector $\mathbf{w_i}$. The vector $\mathbf{w_i}$ has p components w_{ij}. So the output y_i is maximum when the vectors $\mathbf{x}$ and $\mathbf{w_i}$ have the same direction, with a fixed norm.

Let us simplify. We consider that the norm of the vector $\mathbf{w_i}$ is 1(cf. figure VII.2). For each vector $\mathbf{w_i}$, the incremental modification is computed by :

(7.2) $d\mathbf{w_i}/dt = \alpha.y_i\ (\mathbf{x}-\mathbf{w_i})$

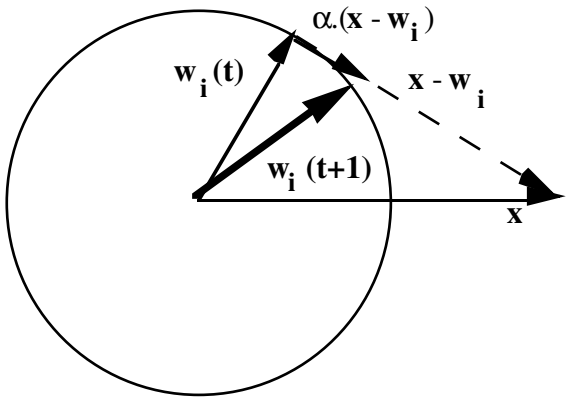

Figure 2.12.: Vectorial representation of the Kohonen's algorithm. The closest vector **wi** from **x** is updating. This updating is represented with a bold line

For a vector $\mathbf{w_i}$, a learning step is equivalent to add a correcting vector $\alpha.(\mathbf{x}-\mathbf{w_i})$, with α a positive adaptation gain. After a normalization phase, the new vector $\mathbf{w_i}$ is closest to **x** than before the learning step. This adjustment of the vector direction is shown at figure VII.2. So the maximum response for **x** is increased with this learning rule.

The convergence is reached when $\mathbf{x} = \mathbf{w_i}$ for the neuron i which better matches the vector **x**. In fact, it always exists a neuron i with the greater response for an input **x**. The vector $\mathbf{w_i}$ for this neuron i is updating in order to better match the vector **x**. And also because of the recurrent connections, this updating is realized in the positive neighbourhood of this neuron. When the convergence is reached, for each vector **x**, there is one active cluster of neurons in the network (corresponding to the positive connections in the "mexican hat" shape).

More, with the recursive connections, a topology organizes itself in the network. That is say : close vectors in the input space are projected in close neurons in the network.

b - Equivalent formulation

Kohonen has proposed a simplified version of this algorithm in which the solution at the convergence is the same as for the previous one. In fact, we consider here is only one active area of neurons when an input **x** is presented. This area covers the neighbourhood of the more active neuron. The synaptic weights of this neuron better match the input **x**. In this area, the outputs are supposed to be equal to 1 and outside this area, equal to 0.

So, this simplified algorithm can be described by :

1. Present an input **x**.

2. Process, for each neuron i, the scalar product $y_i = \mathbf{x}.\mathbf{w_i}$ and search the indices i^* for the neuron which has the maximum output y_i.

3. Apply the learning rule for the neurons which are in the neighbourhood of the target neuron i^* :

$d\mathbf{w_i}/dt = \alpha.y_i(\mathbf{x}-\mathbf{w_i})$

Outside this neighbourhood, the synaptic weights receive no change.

In practice, to reach the convergence, the following rules have to be respected :

1. the adaptation gain α decreases as the function 1/t.

2. an adaptive neighbourhood is initially large in order to structure the network. And with the learning process, this neighbourhood decreases as the topologic relations organize themselves in the network. At the end (at the convergence), the neighbourhood is null.

From a theoretical point of view, the convergence of the Kohonen's algorithm has been demonstrated by M. Cottrell and J.C. Fort [Cottrell89], in the case of a 1D-structure. For a 2D structured neural network, at present, the convergence is not proved. A variant of this algorithm with fixed borders has been proposed by these authors. The convergence of this modified algorithm is demonstrated.

4.6.2. MENTAL program

```
##########################################################################################

/*Inclusions*/

#Include <MENTAL.h>

/*Definitions*/

#define     Ninput              16
#define     time-equilibrium    100

/*Useful miscellanies functions*/

float alea();
int integer-alea();
int h;

/*AGGREGATES AND CELLS DESCRIPTIONS*/

Define      INPUT       a group
      Shape:            Line of-size: Ninput;
      Cells:            Cell (INPUT, .) = kohonen_Cell(INPUT) ;
      Neighborhood:     bi-lateral;
      Connectivity:     {
                              Complete;
                              Symetric;
                        }


Define      NET         an aggregate
      Shape:            In-Layers:
                              Layer 1: INPUT;

      Neighborhood:     {
                              bi-lateral;
                        }
      Connectivity:     {
                              Complete;
                              Symetric;
                        }
      Π-function:       Pi  ($x, $y) = Weight  (INPUT, $x, $y) * Output  ($x);
      Λ-function:       {
```

```
                        max=0;
                        for (h=1; h<=Ninput; h++)
                                if (Input (h)>Input (h-1) )  max=h;
                        for (h=max-1; h<=max+1; h++)
                                Weight  (NET, h, max) = Weight  (INPUT, h, max) +
                                                        alpha*Output  (max)*(Input  (h)-Weight  (INPUT, h,
max);
                                }

        Stop-condition:  Time  (NET) > time-equilibrium;


cell    Kohonen_Cell ( A1 )
        Aggregates:         A1;
        Respectivly to aggregate  A1 :
                {

                        Brodcasting-time  = Time ;
                        Next-wake-up-time  = Time  + 2
                }

Control()
{
        Weight (NET, ., .) = alea(0, 1);
        /*Initialization of weights*/
        for(h=0; h<Number-of-prototypes; h++)
                {
                        Input (Cell (INPUT, .)) = data(.);
                        Learn(net);
}
```

##

4.7. From A.N.N. description language to virtual machine

The compiler going from the A.N.N. description language to the virtual machine should be relatively simple to write.

In fact the only task it has to do is to fill the empty slot of the standard data structures. Those slots are nearly the same one than the ones defined in the Occam harness in § 3.4..

5. COMPARATIVE STUDY WITH OTHER NEURAL NETWORKS PROGRAMMING ENVIRONMENTS

Most of the existing neural networks simulators use different sequential programming language, but the appearance of parallel languages and neural network oriented ones offer more possibilities for a good exploitation. In this paragraph, we present some of the simulators and the languages.

5.1. Artificial Neural Network simulators

5.1.1. P3:

The general-purpose parallel simulator P3 was developed with PDP (Parallel Distributed Processing) group.

In P3 networks, each computing element, called a unit, contains a computer program that reads from connections and sets outputs on other connections, possibly also modifying some local state parameters. The major components of P3 are:

1 - The *plan language*, which describes the collection of units in a model and specifies the connections between them. This description is called a "plan".

2 - The *method language*, an extension to LISP, which implements the internal computational behaviors of the units in a model.

3 - The *constructor*, which transforms the plan and associated methods into a computer program and, when run, simulates the network.

4 - The *simulation environment*, which provides an interactive display-oriented facility for observing and testing P3 models.

The method programs that implement the internal functioning of units are written in the form of ordinary computer program in an appropriate language. The current implementation runs on the Symbolics 3600, write in the Lisp language. In order to allow the methods to use the values of inputs and parameters in their computations, a set of special access statements is incorporated into the system and is available to LISP program.

The job of the P3 plan language is to describe the units and connections that constitute the network being simulated. To do this, the language uses a small but rich set of statements that make it possible to succinctly describe large group of complex connected units. The three fundamental constituents of the plan language are the UNIT TYPE, UNIT, and CONNECT statements. The UNIT TYPE statement names and describes a kind of unit. The UNIT statement instantiates and names actual units. This statement can instantiate either a single unit or a whole array of units of the same type. The CONNECT statement makes connections. Since the statement can be used inside of loops, a single connect statement can make an arbitrarily large number of connections using the available array features.

The P3 system itself is completely synchronous and updates all units on each cycle. Since updating a unit involves invoking its method, the question of whether or not the outputs of a unit actually change on any P3 cycle can be decided by the method. For example, to model asynchronous updating, each unit can have an additional inputs that controls whether or not it is update on a cycle. Then the decision as to which units are to be updated can be given to a control unit that is connected by a separate line to the update inputs of all the other units.The method program inside this control unit decides which units in the network will be update on each cycle.

A typical P3 plan might contain a large number of simple neuron-like units forming the core of the network together with a few special purpose units to generate input to the core network and control its function. The master control unit, used above to implement asynchronous updating, is an example of this kind of special-purpose unit.

The dynamism managed by the interface update the intermediate text generated by the constructor. The generation of an intermediate code for the simulation impose:

1 - stopping the simulation,

2 - a complete edition of the code described the network, and

3 - a new compilation.

Example:

```
(unit type dipole
        parameters flag i1 j1 i2 j2
        ouputs (d array i j)
        method <routine écrite en Lisp> )

( unit type competitor
        parameters p g flag
        inputs (C array i j terminal parameters W)
```

```
            (i-A)
        outputs (o-A)
        method <routine écrite en Lisp> )
```

Instanciation of units. Units are distributed in an euclidian space of three dimension (@ x y z),where x,y,z ∈ $\mathbb{N}^3$.

```
(unit stimulus of type dipole
        at(@ 000)
        outputs (d array (i 0 5) (j 0 5))))

(unit cluster array (k 0- cluster-size 1) of type competitor
        at (@ (* k (+ cluster-size 4)) (+ cluster-size 10) 0)
        initialize (g = 0,05)
        inputs (C array (i 0 (- stimulus-size 1)) (j 0(- stimulus-size 1))))
```

Declaration of connection network and cells activation. Outputs of stimulus units are connected to each cluster units with a weight generated in a random fashion.

```
(for (k 0 (+1 k)) exit when (> k cluster-size) do
        (for (i 0 (+1 i)) exit when (> i stimulus-size) do
                (for (j 0 (+1 j)) exit when (> j stimulus-size) do
                        (connect unit stimulus output d i j
                                to unit (cluster k) input C i j
                                terminal initialise (W = si:random-in-range
                                        0,0 (//2,0 (expt (+ stimulus-size 1) 2))))))))

(for (k 0 (+1 k)) exit when (> k cluster-size) do
        (for (j 0 (+1 j)) exit when (= j k) do
        (connect unit cluster k output o-A to unit cluster j input i-A))
        (for (j (+ k 1) (+ 1 j)) exit when (> j cluster-size) do
        (connect unit cluster k output o-A to unit cluster j input i-A)))
```

Environment access are assumed by Lisp functions. Therefore obtain the value of a parameter or of an array element uses the following forms :

```
        (read-unit parameter flag) ou (red-unit (C i j))
```

It allows you to access the variables flag and (C i j).

5.1.2. Genesis

Genesis is a graphically oriented, general purpose simulation system to facilitate the modeling of neural networks, developed at San Diego California in 1989. The entire system has been designed to run under UNIX and X-windows.

The design of the simulator and interface is based on a "building-block" approach. Simulations are constructed of modules which receive inputs, perform calculations on them, and generate outputs. Network specification and control is done at a high level using graphical tools and a network specification language. The graphics interface consists of a number of tools which the user can configure to suit a particular simulation. The specification language consists of a set of simulator and interface functions that can be executed interactively from the keyboard or from text files storing command sequences. The language also provides for arithmetic operations and program control functions such as looping conditional statements, and subprograms.

Modules in Genesis are divided into computational modules, communication modules and graphical modules. All instance of computational modules called *elements*. These are the central components of simulations, performing all of the numerical calculations. Elements can communicate into two ways: via *links* and via *connections*, Links allow the passing of data between two elements with no time delay and with no computation being performed on the data. Thus, links serve to unify a large number of elements into a single computational unit. Connections, on the other hand, interconnect computational units via simulated communication channels which can incorporate time delays and perform transformations on data being transmitted.

Example:

Create different types of elements and assign them names.

```
create                          neuron1
create      active_compartment  cell_body
create      passive_compartment dendrite
create      synapse             dendrite/synapse
```

Establish functional links between the elements.

```
link        dendrite            to      cell_body
link        dendrite/synapse    to      dendrite
```

Set parameters associated with the elements.

```
set         dendrite    capacitance     1.0e-6
```

Make copies of entire element subtrees.

```
copy        neuron1         to      neuron2
```

Establish connections between two elements. Notice that the hierarchical representation is maintain during manipulation of elements.

```
connect     neuron1/axone   to      neuron2/dendrite/synapse
```

Set up a graph to monitor an element variable.

```
graph       neuron1/cell_body       potential
```

Make a control panel with several control widgets.

```
xform       control
xdialog     nstep       set-nstep   -default 200
xdialog     dt          set-dt      -default 0.5
xtoglle     Euler       set_euler.
```

5.1.3. Nnsim

Nnsim is a simulation system of a.n.n. built in 1988 at Stuttgart University, Germany. It is a collection of functions write in C [Nijhuis89].

Internal behaviour of a cell is described in C, it is sequential and deterministic. Functions allowed programmer to maintain the abstraction level as a.n.n. are cells running in concurrency which exchange messages. Units are located in the Euclidien space of one dimension, as in Genesis. Nnsim has adopted an Object oriented approach. Therefore cells are viewed like passive object waiting activation messages.

A communication library is available. It offers communications : point à point, one element toward a group of elements (diffusion) and, many elements towards many others. The execution correspond to a synchronous model like in P3.

As the simulator system is ineractive, it possesses primitives allowing to stop simulation and to mofify the network without all to recompile. Dynamical management is controled by the 'Network handler'. It control network construction, network modifications and network execution. Only the 'Network handler' communicates directly with the internal representation model such it is stored in Nnsim.

5.1.4. Nacre, Slogan et N

The simulator environment Nacre comes from the Pygmalion project. It is an Esprit project from the European community 2059 started in 1988. Its main difference from the previous systems is the elaboration of a dedicated asynchronous a.n.n. language. The language Slogan incorporate directly characteristics mainly devoted to a.n.n. simulator systems. There is a prototype of Slogan running on a Lisp machine. Slogan was used to define the N language. N is conceptually nearness Actor model [Angeniol88, Derot89, Moulin.89, Actor88]. We constraint our presentation to Slogan ; we are mainly interested by the characteristics introduced in order to built an a.n.n. language.

Slogan is an object oriented language. It offers a functional description and a structural description of neural networks. Cell behaviour is described by a class and activate by a method. A class is composed of parameters, of local variables and of sub-classes. A method is a function associated to a class which can send messages towards sub-classes and where the parallel execution is described.

The system is entirely asynchronous ; concurrency is manage by parallelism. It is possible in Slogan to specify that an instruction have 80% of chance to be executed. Therefore Slogan covers models describing cells activity in a probabilistic way. For example, an instruction which specify a complete connectivity with a coefficient of 0.5 will be interpreted establishing 50% of connections. Activation of a method in a repetition structure followed by a coefficient of 0.7 will be activated in 70 per cent of cases.

Such a temporal management is basically local. On the other hand parallelism is a fair parallelism, methods do not synchronize.

Communications among methods are realised through specific typed variables : plug-in type and plug-out type. communications are unidirectionnal, non-symetric and strongly typed. Messages are set in a list which is going to be read and set up. An other aspect of communications between tow methods comes from the fact that synaptic weight between tow cells are frequently symmetrics (Wij = Wji). Following this consideration Slogan uses shared variables. There is three types of shared variables:

- plugged variables for input or output. They ensure communications with models at level n-1 or n+1;
- accessed variables whose value must be given by a higher level model;
- share-up variables are common at this level of class and upper classes.

Primitives allow to create and to destroy links (plugs) during execution. Only methods are affected by dynamic management ; classes stay the basic model. It is not possible to change the global structure ; only little modifications concerning connections are possible.

5.1.5. The Rochester Connectionnist Simulator

The RCS is not restrain to artificial neural networks but includes a wide variety of connectionnist networks. Its main design criterion has been flexibility . Each unit can compute a different function, any amount of data may be associated with each unit, and an arbitrary connection pattern may be specified.

int MakeUnit (type, func, init_pot, potential, data, output, init-state, state)

type is a pointer to character string, and is simply used for display purposes. *func* is a pointer to the function used to simulate the unit's action. *potential* is the activation level for the unit. *data* is a value for the unit data field described above. *output* is the initial output of the unit, like *state* is the initial state value. *init-pot* and *init-state* are the values to set the unit potential and state when the network is reset.

NameUnit (name , type , index , length , depth)

This function allows to declare array units as simple as single unit. The *name* is a pointer to character string, may be used during simulation from the command interface. *Type* is SCALAR, VECTOR or ARRAY. *Length* and *depth* are the dimensions of the unit.

Each unit has a number of sites at which the incoming links arrive. The provision of sites allows differential treatment of inputs, since the links themselves do not indicate their origin at the destination unit.

Site * AddSite (index , name , function data)

Index is the index of the unit to which the site is to be attached. *Name* is a pointer to a character string which will be the name of the site. *Function* is a pointer to the function to be called to simulate the action of the site. *Data* is the value described above. Simulation may be run synchronously or asynchronously.

Link * MakeLink (souce-unit,destination-unit,site-name,weight,data,function)

MakeLink creates unidirectional point to point link which supports a weight and a function. Therefore a link can transform messages in a more complicated way than a simple multiplication by a weight.

5.1.6. Nestor Development System (NDS)

NDS was developed by Nestor company. Its objective is to resolve classification problems find in industrial, scientific, medical and financial environments. Its force is its aptitude to develop applications in less time than with classical software technologies. Like all a.n.n. no specific algorithm is needed. NDS offers a single learning rule RCE, presented by Reilly and Cooper. The architecture of the RCE network is constituted of three layers : an input layer, an output layer and an hidden layer. Each node of input layer stores value of examples to recognize. Output cells correspond and code the categories, the classes of which depend examples. Hidden cells define a surface in the space of examples. This sub-space is a sphere centre on an example and of radius lambda. All examples of the same class which fall into the sphere is recognize, if it's not the case, an other sphere is created. If an example does not imply the same class, the radius is reduced and a new sphere dedicated to this class is created [].

5.1.7. Mirrors/ II

Mirrors/II is a general purpose connectionnist simulation environment for developing connectionnist models based on Mirror/ I experiences. Mirrors/II is currently in use at University of Maryland, since 1988. Mirrors/II is distinguished by its support of a very high-level non procedural language along with user and system libraries for he accumulation of system resources. The paradigm of Mirrors/II is those of "massively" parallel systems. Massively parallel systems are often referred to as connectionnist models because they consist of a set of nodes which are connected together and communicate via links.

Mirrors/II is not itself a connectionnist simulation environment; it is a general-purpose software environment with an indexed library of system components. Its libraries contain predefined connectionnist models facilitating user starts in connectionnist models implementation. There is spreading activation methods, learning methods and control strategies.

The Mirrors/II system is currently written in Lisp. This Lisp implementation allows conventional artificial intelligence software to be easily integrated into Mirrors environment.

5.1.8. GN lami simulator

Lami simulator is a part of NERVES project. It is a useful tool for teaching basic artificial neural networks models. Since elaborated a general purpose simulator implements Kohonen network, Hopfield network, Hérault-Jutten network, backpropagation network and a specialised network to control an inverted pendulum. With this simulator, the user can discover the main aspects of networks behaviour and the differencies between the learning rules. He can also understand the meaning of the weigths, thanks to the visualization possibilities.

a - Kohonen neural network

Learning in this network is an auto-organizing process which, starting from an initial disordered configuration of the neuron weights tends by adapting these weights to produces a configuration in which the topological relationships between input vectors and the corresponding neuron activations are preserved, and the probability distribution of the input vectors is approximated by the distribution of neurons. The simulator visualizes the network state during learning, and allows interactive modification of a number of parameters, such as network size and input distribution. It is also possible to fix the precision of the numerical values. Therefore rounding errors can be revealed.

b - Hopfield network

This simulator uses Hopfield network model to recognize characters. The size of the network can be chosen at program start. Two types of iteration are available : parallel or Monte-Carlo (random sequential iteration).

c - Hérault-Jutten network

This simulator allows user to experiment recursive network. The source signals can be visualised and modified by user during simulation. Size of the network, mixing matrix an learning coefficients can also be changed.

d - Backpropagation network

This simulator deals with approximate transfer functions (continous outputs). It provides a graphical function display, a supervisor selector and a network state viewer.

e - A specialised network to control an inverted pendulum

This simulator shows an application of learning through experimentation and generalization principles. The neural part is a multi-layer network that is trained by the gradient backpropagation algorithm an then optimized by a random search method. It uses the same software that Backpropagation network.

5.1.9. Cognitron

Cognitron is a neural network prototyping and delivery system for the Macintosh. It includes examples as Back-propagation, Pattern association using Delta rule, … , pattern recognition and signal processing applications. It offers many tools directly accessible by the interface window like *excite* which raises the activation of a unit by defined a percentage, *clamp* which holds the activation of a unit constant or *new* used to create new units. It is also possible to program new unit in a specific Lisp, nearby P3 Lisp, and to control the grain size of parallelism like in Occam.

5.1.10. Ecn neurocomputing GmbH

We briefly present the Ecn neurocomputing simulator.

Ecn neurocomputing offers hardware and software systems and maintenance. The simulator environment is essentially constituted by the neural network simulation package SAIC-ANS015. It contains libraries and a high level language ANSpec. ANSpec is an actor oriented language which has been developed by the company to model massively parallel distributed systems such as artificial neural networks.

5.1.11. DESIRE/NEUNET

DESIRE/NEUNET is derived from the well tested DESIRE (Direct Executing SImulation in REal time) package for dynamic systems such as servomechanisms and physiological models. DESIRE/NEUNET interprets only experiment-protocol program which sets parameters and calls a simulation run for network training or decision making. The interpreter program may then inspect or process simulation results, decide to change parameters, and make additional simulation runs. It executes directly modifications without the usual annoying wait for program translation.

A true compiler is built into the powerful interpreter and compiles all time-critical operations directly into memory. A reasonable matrix language makes it easy to split large neuron interconnection matrices onto smaller, intuitively meaningful pieces.

VECTOR net = alpha*A*a + beta*B*b - gamma*C*c - delta*d + bias

where a,b,c,d are various activations, and A, B, C are connection weight matrices.

5.2. Artificial Neural Network Languages

As there exist a lot of languages there is not many artificial neural networks languages. Condela, Nesila, Slogan, Mental and Cellsim claims to be dedicated to a.n.n. Slogan was presented with its simulation environment, Nacre. Mental and Neural was presented as part of this report. We are going to present in this section Condela, Nesila and CELIP.

5.2.1. Condela

Condela is a high level language specifically write for the development of neural networks programs. It is a procedural language where parallelism is introduced by the concept of unit selection. A type units library and an access functions library make the programming of PDP models, like backpropagation or Boltzmann machine, easier. Condela offers primitives in order to manipulate a.n.n. as layered networks.

Condela is under development, since 1988, at the Technical University of Vienne in Austria. Actually the Condela III (release 1.0) is available via ftp. This version is executable under Unix system. It is mainly a C preprocessor.

The internal behaviour of a cell is described in Lisp, in C or in any other classical sequential language. Programmer can think in parallel as cells are viewed as independent processes. Programmer specifies behaviour without being preoccupied by the underlying architecture of computer.

The main idea in Condela is to structure neural networks in layered network. A layer is instanciated by a name which represent a set of units. It is therefore possible to connect two layers with a link. Each link supports a weight. Default values for initial weight may be specified, by default it is set up to zero. the activation and the execution of Condela neural networks are synchronous.

Example:

We have choose to present the NOR function. It is implemented with the backpropagation model. firstly we define the connections between layers

```
NETWORK xor CONSISTS OF

    -- firstly we define the layers.

    LAYER input OF [2] UNITS OF TYPE back_prop END LAYER
    LAYER hidden OF
```

```
            LAYER compute1 of [2] UNITS OF TYPE back_prop END LAYER
            LAYER dummy1 of [1] UNITS OF TYPE back_prop END LAYER
        END LAYER
        LAYER output OF [1] UNITS OF TYPE back_prop END LAYER
            LAYER compute2 of [2] UNITS OF TYPE back_prop END LAYER
            LAYER dummy2 of [1] UNITS OF TYPE back_prop END LAYER
        END LAYER
END NETWORK;

-- secondly we define the connections between layers.

CONNECT input TO hidden INIT VALUE random*2-1 ;
CONNECT hidden TO output INIT VALUE random*2-1 ;

-- and then the control structure of execution.

DECLARE i: INTEGER;
DECLARE target : VECTOR;
        dummy1 = 1 ;
        dummy2 = 1 ;
        LOOP 2000 TIMES;
            i = truncate(random*4);
            input[1] = i DIV 2; input[2] = i MOD 2;
            target[1] = (i DIV 2 + i MOD 2) MOD 2;
            EVAL hidden;
            EVAL output;
            CALL show_error (input,ouput,target);
            CALL backprop (output,target);
            CALL backprop (hidden);
        END LOOP;
END
```

The dynamic aspect of Condela relies on the primitives CREATE and CONNECT. Notice that dynamic creation of cells implies dynamic creation of links : an isolated cell carries no information [Almassy90]. It is possible in Condela to defined entirely any a.n.n. dynamically.

5.2.2. Nesila

Nesila (Network Simulation Language) is a declarative language for the specification of a.n.n. build at the Stuttgart University. It is mainly dedicated to implement medium to large sized networks, between 10^2 to 10^4 units. Nesila has to be viewed as a kernel of a simulator environment actually in development [Korb89].

Each cell is entirely defined by an unique number, use at activation, a name, a type, and the connections.

UNIT(No,Name,Type,Activ,Output)

There is three predefined types of cells in Nesila: INPUT, OUPUT and HIDDEN. The INPUT and OUTPUT types are used to define cells which interface hidden cells between environment. You can notice that the programmer must manage when to activate rules of learning. The simulation environment no more than Nesila is able to distinguish when learning rules or recognizing rules are to be activated.

Cells, like in Condela, can be group together to build subsets more easy to manipulate.

Communications are established by links. Links are defined by an output unit, an input unit and a weight :

SourceUnit --> TargetUnit:ConnStrength;

Weight can be omitted. It is therefore set to zero by default. one interesting characteristic of Nesila is the possibility to describe a set of connections in a conditional way.

```
UnitSet = [ Unit(i,Name,HIDDEN,0,0) WHERE 0<= i < 100] ;
ConnSet = [x-->y:inhibitory WHERE x,y IN UnitSet AND x<>y];
inhibitory = -100;
```

The behaviour of network described in Nesila can be activated from many ways :

- in a synchronous way, all cells are up date at the same time;
- in a random order of activation or
- in a fixed order of activation.

Presented as a main objective of Nesila construction is the elaboration of a dynamical internal representation. Dynamicity is to understand in a classical sense : The internal representation is only limited by the computer capacity, the activation mode of a.n.n. and the variations of weights are dynamic, like variables are.

5.2.3. Cellsim and CELIP

Cellular automata are specific neural networks. They have regular space structures and regular time structures. Cellular automata have a finite number of elements, they are homogeneous and they evolve in a synchronous way [DP86].

As Cellsim simulator and CELIP language present very similar characteristics, we choose to present just CELIP language.

Cellsim is a simulator oriented towards prototyping cellular algorithms for image processing. It is basically an extension of the standard C. Cellsim introduces simultaneous evaluation of all cells. It allows to fix the shape and the structure of cell network.

In the same way, CELIP (CEllular Language for Image Processing) was developed for prototyping cellular algorithms for image processing. it is also an extension of standard C.

Cells are C functions. Among cells variables some of them are dedicated to communication. They are specified in the following way :

```
cell
{
        byte    reg,            /* mémorise le gris du pixel = l'état de la cellule   */
                temp;           /* sert à échanger deux gris                          */
        byte regarr[9] ;        /* pour stocker les gris des voisins                  */
}
```

Connections are entirely defined by the *neightbourghood* table

```
neightbourghood
{
```

```
        MOORE     [
                  4       3       2
                  5       $0      1
                  6       7       8
                  ]
}
```

and allow cell to consult the different variables dedicated to communications.

regarr[i] = MOORE[i].reg

The $ symbol separates the current cell from its neightbourghoods. Therefore MOORE[0].reg is in our example the variable reg of the current cell itself.

PART 3

DEVELOPMENTS AROUND SMART

1. SYNOPSIS

Artificial Neural Networks often rhyme with massive parallelism. We need parallelism to simulate neural nets and fully benefit from their promise, but we do need an efficient way to implement them. So we propose a dedicated approach, general enough to encompass a large class of models. It consists in performing arithmetics oriented algorithms which leads to the joint development of hardware and software tools: the *SMART* machine. **SMART stands for Sparse Matrix Adaptive Recursive Transforms, the general framework we use to describe ANN**. In order to handle networks with a sparse or dynamic connection topology we propose an original architecture. Using high speed CMOS technology this architecture will deliver up to 300 MFlops in a deck side format, supporting up to one million connections: from 1,000 fully interconnected *neuronlikes* to 10,000 sparsely interconnected ones. With the user-friendly UNIX Workstation host and integrated high speed VME channels we create a ***Neurostation*** exhibiting a power to price ratio many times greater than that of a super-computer. In this part, we discuss first the SMART approach and its appeal. Second, we briefly present a model of hardware architecture for SMART sparse calculus. Third, we address the programming issue and display preliminary results.

2. THE SMART APPEAL

A connections matrix is the most shared formalism in papers about ANNs. As a matter of fact it is the regular way of setting coupled equations, hence we can find very general models for which theoretical proofs are derived. However for further studies and practical implementations, this straightforward view seems computationally unmanageable. More and more, different networks are linked together; each of them is assigned to a particular task such as visual perception, association, planning or motor control… And it does not sound well to melt all these neurons in the same pot, with the same dynamics and any individual one fully connected to all others. Sure enough, there would be a lot of learning and stability problems provided we can afford the processing and memory size required by this connective mess. It is more appealing therefore to have a dynamic topology, deleting unused connection after a first learning, thus freezing basic concepts, then adding new connections to relate these rough shapes while further learning transpires.

As far as ANN software simulations are concerned, an oriented description for *objects* (actors…) is the answer of the moment . Any instance of a *neuron object* stores its weighted links and an activation program, enabling whatever hybrid network description we may dream of. Then it is easy to derive one task per *neuron*; and as we have a great number of *neurons*, we obtain the required number of tasks for massive parallelism. But what about the implementation? The dynamic load balancing upon a large set of processors of such a huge number of asynchronous tasks is still an issue. Since we have very few a priori constraints on the neurons -different link numbers, adaptation rules, activation laws…- a static balancing is difficult as well. We get the feeling there is a great overhead load for the sake of synchronizing a lot of a priori asynchronous tasks which perform more or less the same program. Obviously for any probable implementation, we will need only a small set of objects to refer to. Most likely we shall let each neuron vary from the model through learning rather than customize it.

The SMART approach infers matrices from the object oriented description. It simply groups synapses in clusters as large as possible, according to the various dynamics and adaptation rules. Hence we get a small set of connection matrices, presenting of course a sparse topology. Then neurons are grouped according to their activation law. If we get too many matrices or groups, there is a design problem and some reductions are required. For instance, in order to achieve different neural responses let us set a function with one extra argument to select the suitable response. Thus a set of a small number of very large tasks is defined. We might call them sequentially, but each task is easy to parallelize since it consists in the same operation to be performed many times on independent data.

We note there are many ways to perform one operation, e.g. the information weighting through a link. The plainest one is to multiply the neuron *output* with a weighting factor. This generally assumes multivalued responses which account for time and space averaging of spikes over a cluster of real neurons. *SMART* adopts this side since an efficient use of the information coded by pulse sequences would require very complex links and might lead to the same average result. As arithmetics become cheaper and the Amdhal law is still enforced, **the point is to perform arithmetics rather than conditional operations** and a lot of steps of a simple algorithm rather than few steps of a complex one which leads to the same result.

This general view defined, we can devise an efficient architecture for a static load balancing, with no contention or synchronization problems. Sparse matrix handling will allow powerful implementations of complex networks:

1 - The dot product can be extended to a convolution with a matrix to compute each term, hence the weighting will become a recursive filtering.

2 - A few matrices with various dynamics will lead to a multi-synchronous network.

3 - Networks with dynamic topology will be valid too, provided we support quasi-static load balancing.

3. AN EFFICIENT HARDWARE ARCHITECTURE

Many techniques have been suggested for the efficient multiplication of sparse matrices by vectors; they involve often highly irregular storage schemes and algorithms for handling non-zero elements in the matrix. Although these techniques lead to very powerful sequential implementations; they are, in general, not very suitable for parallel architectures. Systolic architectures were implemented using some of these techniques, but they were not designed to support matrix updating. Meanwhile, *CRASY*, another systolic architecture dedicated to ANN simulations exactly solved the last problem. Unfortunately it did not deal with sparse matrix computing which imposes an advance addressing to avoid the storage of zero elements and to limit the calculus cycle number.

Using *CRASY* architecture, **we are designing *SMART* to carry on sparse matrix multiplication by vector as well as matrix updating**. It is composed of a *SPARC®* microprocessor as control unit, and an array co-processor as processing unit (see Figure 3.1.).

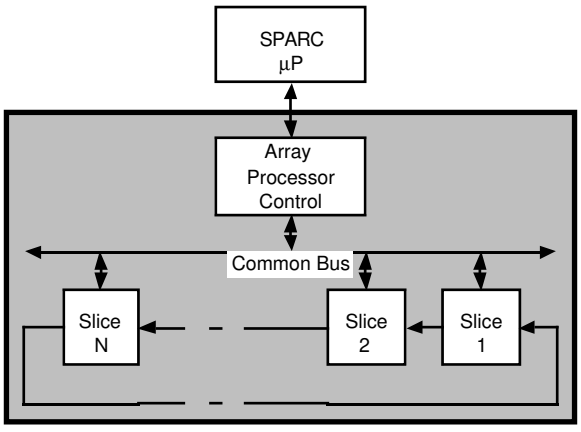

Figure 3.1.

Many identical blocks, called slices, form the array co-processor. Each slice has its own local memory and two links to its neighbourhood, while broadcasting between the slices is performed by a common bus (see Figure 3.2.).

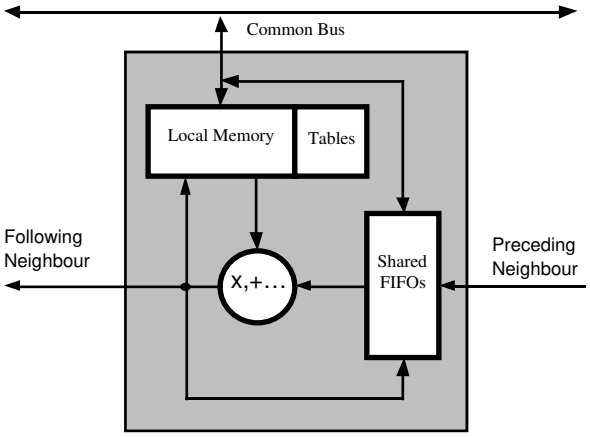

Figure 3.2.

The originality of this architecture comes from three major points:

1 - The way we control slices permits us to achieve both parallel and systolic operating modes and the potential to mix the two modes.

2 - In order to resynchronize the executed instructions, FIFOs are placed between slices. Therefore parallel and systolic instructions can share the same pipe-line to get the best performance.

3 - The method to avoid zero multiplications, when we multiply a sparse matrix by a vector, is to construct some tables that point to a vector component location for any given non-zero coefficient in the matrix. In other word these tables represent the result of compacting sparse matrices: zero coefficients which represent unused synapses are not stored. Part of the hardware is designed to support these tables with no extra load.

The preliminary simulation for this architecture yields encouraging results, some of these are illustrated below:

1 - The maximum efficiency per slice can reach up to 20 MFLOPS affected by compacting sparse matrices performance.

2 - The number of connections assigned per slice can reach up to 64000 connections. This number of connections between neurons is affected by compacting sparse matrices performance.

3 - The maximum number of neurons assigned per slice is about 1000 neurons according to the simulated network architecture.

Figure 3.3. displays simulations of the compacting performance mentioned above. Synapses are randomly distributed on a network of 1000 neurons for various average numbers of synapses per neuron. Then the corresponding matrices are squeezed and efficiency factor is deduced from slice memory allocation on an eight-slice machine configuration.

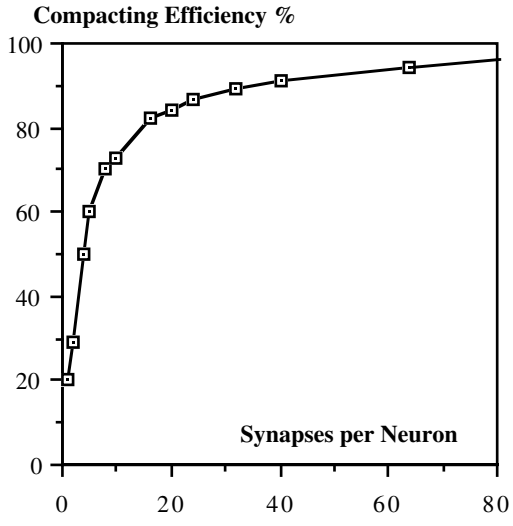

Figure 3.3.: Performance of the allocating algorithm

4. SOFTWARE TOOLS

The design of *SMART* hardware is comprehensively performed with some software tools, since such a machine cannot be efficient unless a suitable software environment is provided. So, SMART will be linked to a workstation to get the benefit of a standard friendly environment (see Figure 3.4.). The combination (workstation + *SMART*) forms a parallel system in which each machine is based on the same *SPARC*®processor model.

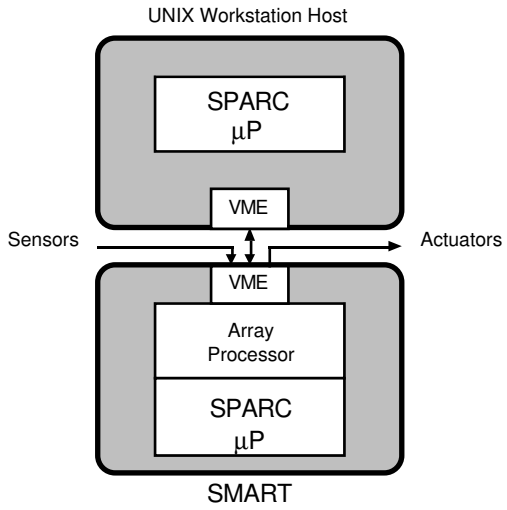

Figure 3.4.: System combination

This point of view allows us to define a simple model for the combination.

A new language programming the combination appears necessary to achieve the best implementation of ANN simulations. This language should have certain features, such as the following:

1 - The ability to implement algorithms other than the ANN ones, for instance signal processing, image processing, etc..

2 - The ability to manipulate matrices and vectors, in other words, the language should have limited instructions to define matrices and vectors, and to apply all the possible operations between them. Sparse matrices will be treated as a special type.

3 - The ability to describe parallel tasks. In fact, we impose parallel notions on this language for tasks have to be shared between the two machines. *SMART* executes all vectorial instructions concerning matrices and vectors, whereas the host performs monitoring. Thus a data flow is established between the machines. It is controlled by a special protocol introduced into the language. The user may assign the tasks for each machine in his program, therefore the synchronization is achieved using rendez-vous set up by either the user or the compiler.

A good choice is made for the language by extending the C language to a vectorial one. This way, we use all the software available on the host by adding a simple interface to each. This language will be the first step towards an implementation of a high level specification language for ANN on *SMART*.

An advanced compiler should be developed for the extended C according to *SMART* architecture. This means, the compiler should handle the following points:

1 - All the classical obstacles caused by the parallelism like communication, synchronization … should be considered.

2 - The memory management for matrices, especially sparse ones, is an important task to perform. As we mentioned before, in describing hardware architecture, matrices should be stored in memory in the form of lists which provide information about matrix structures. Then the compiler should deliver these lists for each matrix declared in a program, using some strategies to optimize memory allocation and load balancing upon the slices.

3 - The various number of slices allowed by SMART has to be transparent. A recompilation is required for programs when a change in the slice number takes place.

An assembly language is developed for SMART as first step toward the software tools, mentioned above. This language is composed of the assembly language of SPARC plus a set of instructions for the array processor, which is accommodated as coprocessor-operate instructions(CPop#). So a pre-processor is designed to interpret the additional set of instructions into CPop1 (see Figure 3.5.).

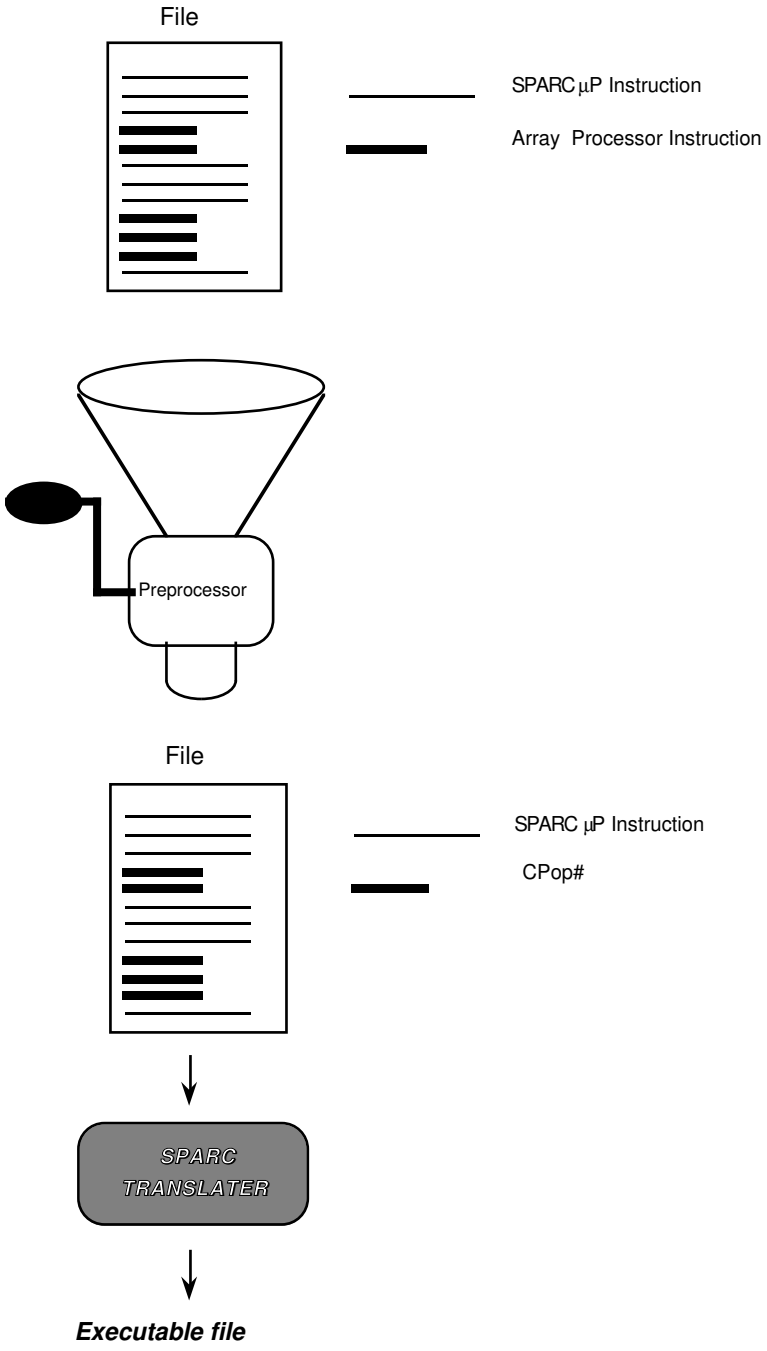

Figure 3.5.

The array processor instructions consist of generalized instructions of SPARC, like Vector add instructions, vector subtract instructions, vector multiply instructions, vector maximum and minimum instructions, co-processor pipe-line control instructions, vector logical instructions, . . etc (see Table 3.1.). These instruction achieve all the calculus deal with vector and matrix.

instruction	code	Description
CVFADD	1	vector floating-point add
CVFAADD	2	vector floating-point absolute add
CVFADDA	3	vector floating-point add absolute
CVFNADD	65	vector floating-point minus add
CVFNAADD	4	vector floating-point minus abs add
CVFNADDA	5	vector floating-point minus add abs
CVFADDAB	6	vector floating-point hybrid add abs
CVFAADDAB	7	vector floating-point absolute hyb add
CVFNADDAB	8	vector floating-point minus hyb add
CVFNAADDAB	9	vector floating-point minus abs hyb add
CVIADD	10	vector integer add
CVIAADD	11	vector integer absolute add
CVFSUB	12	vector floating-point subtract
CVFASUB	13	vector floating-point absolute subtract
CVFSUBA	14	vector floating-point sub absolute
CVFASUBA	66	vector floating-point absolute sub abs
CVFNASUB	15	vector floating-point minus abs sub
CVFNASUBA	16	vector floating-point minus abs sub abs
CVFSUBAB	17	vector floating-point hybrid sub abs
CVFASUBAB	18	vector floating-point absolute hyb sub
CVFNSUBAB	19	vector floating-point minus hyb sub
CVFNASUBAB	20	vector floating-point minus abs hyb sub
CVISUB	21	vector integer subtract
CVIASUB	22	vector integer absolute subtract
CVFMUL	23	vector floating-point multiply
CVFAMUL	24	vector floating-point absolute multiply
CVFSGN	25	vector floating-point sign times
CVIMUL	26	vector integer multiply
CVISGN	27	vector integer sign times
CVFMAC	28	vector floating-point multiply accu
CVFACM	29	vector floating-point accumulate mul
CVFMACZ	30	vector floating-point multiply
CVFACMU	31	vector floating-point accumulate
CVFRCP	32	vector floating-point reciprocal seed
CVFDST	33	vector floating-point division step
CVFMAX	34	vector floating-point maximum
CVFMAXA	35	vector floating-point maximum
CVIMAX	36	vector integer maximum
CVFMIN	37	vector floating-point minimum
CVFMINA	38	vector floating-point minimum
CVIMIN	39	vector integer minimum
CVFSAT	40	vector floating-point saturate
CVFEQ	41	vector floating-point equality
CVFNEQ	42	vector floating-point non-equality

CVFGT	43	vector floating-point superiority
CVFLT	45	vector floating-point inferiority
CVIEQ	46	vector integer equality
CVINEQ	47	vector integer non-equality
CVIGT	48	vector integer superiority
CVILT	49	vector integer inferiority
CVFRND	50	vector floating-point round
CVFSTI	51	vector floating-point scale to integer
CVISTF	52	vector integer scale to floating-point
CVIAND	53	vector logical and
CVIOR	54	vector logical or
CVIXOR	55	vector logical xor
CVINOT	64	vector logical not
CVILSH	56	vector logical shift
CVIASH	57	vector arithmetic shift
CMOV	58	coprocessor move
CTST	59	coprocessor register test
CRST	60	coprocessor register reset
CPLOCK	61	coprocessor pipe-line lock
CPULOCK	62	coprocessor pipe-line unlock
CPSTUFF	63	coprocessor pipe-line stuff

Table 3.1.

5. STATE OF DEVELOPMENT:

SMART is composed of several boards delivered in a deck side format. The first prototype should be realized for the coming summer. Meanwhile, the conception of the software environment is done, and a great part of it is under construction.

For the moment, we are working on utilities that smart needs like monitor, debugger, .. etc, using some standard software available for SPARC processor (Sparc Architecture Simulator, Sparc Monitor, Remote debugger .. etc); in the same time we develop some applications for SMART like Kohonen algorithm, source separation ..etc, using the assembly language and a basic library for the operations deal with matrix.

In the future an engineer will join our group during 12 month to continue the applications development.

PART 4

DEVELOPMENTS AROUND SUPERNODE

In this paragraph we present briefly the SuperNode, a Transputer based reconfigurable parallel architecture of super-computers which can be used in both M.I.M.D. (Multiple Instructions Multiple Data) and S.I.M.D. (Single Instruction Multiple Data) modes or, rather, S.P.M.D. (Single Program Multiple Data), to be precise.

1. THE MACHINE

The SuperNode machine is constructed from Transputers and communication processors.

1.1. The Transputer

Each Transputer is based on a 32 or 16 bits processing unit that includes link communication devices, a floating point unit and a private memory extending from (today) 2kbytes fast internal memory on chip to 4 Gbytes of external memory.

Four serial bi-directional links are available for external communications. These links can be directly connected to other Transputer links or through link interface adaptors to other hardware devices.

The link interface implies that Transputers networks are not classical local area networks but genuine multi-processor networks. There is no shared memory, information exchange is explicitly done using messages sent through links. Links throughput is 20 Mbits/s.

The Transputer has hardware support for concurrent execution. It implements and manages a process abstraction. Each process has a process descriptor which records process status and hardware resources usage (timers).

The Transputer improves context switching by doing micro-coded process scheduling. The context of a Transputer process is a few words (five 32-bits words on a T800, which brings process context switch to 600 ns). There are two levels of priority. Both priorities process scheduling is communication driven. Furthermore, low priority processes are time-sliced while high priority processes are not.

More recently Transputer family extends with the H1. The H1 Transputer increases Transputer performances: Links throughput is 80 Mbits/s. H1 Transputer offers enhanced process model communications. The limitation of four links on Transputers is overshoot by the use of virtual links. Close to Occam channels H1 communication assume multiplexage and routing through network.

1.2. SuperNode architectural model

Three keywords are at the basis of the SuperNode architectural model: modularity, homogeneity and expandability (see Figure 4.1.).

A SuperNode is a collection of lower level machines which can be either SuperNodes or Transputers at the lowest level.

Within a n-level SuperNode, each (n-1)-level SuperNode has a number of physical links dedicated to external communication. A n-level switch is responsible for establishing link to link correspondences between (n-1)-level links. It is capable of performing any possible link to link interconnection. The switch is dynamically reconfigurable under the control of a control processor attached to each level. Each module has the same regular structure: a set of sub-modules interconnected via a switch. Extending the machine is just a matter of adding another level of switching.

The basic module is a collection of working processors each having its own local memory (from 256k of fast access memory to 4M of DRAM per Transputer), interconnected through a switch.

The switch also allows server processors to be added at each level in order to provide system services such as memory, disks servers or I/O dedicated controllers.

The basic module is controlled by a processor that masters a control bus where working processors are connected.

To help providing fine grain parallelism, the Transputer hardware manages a basic form of process abstraction.

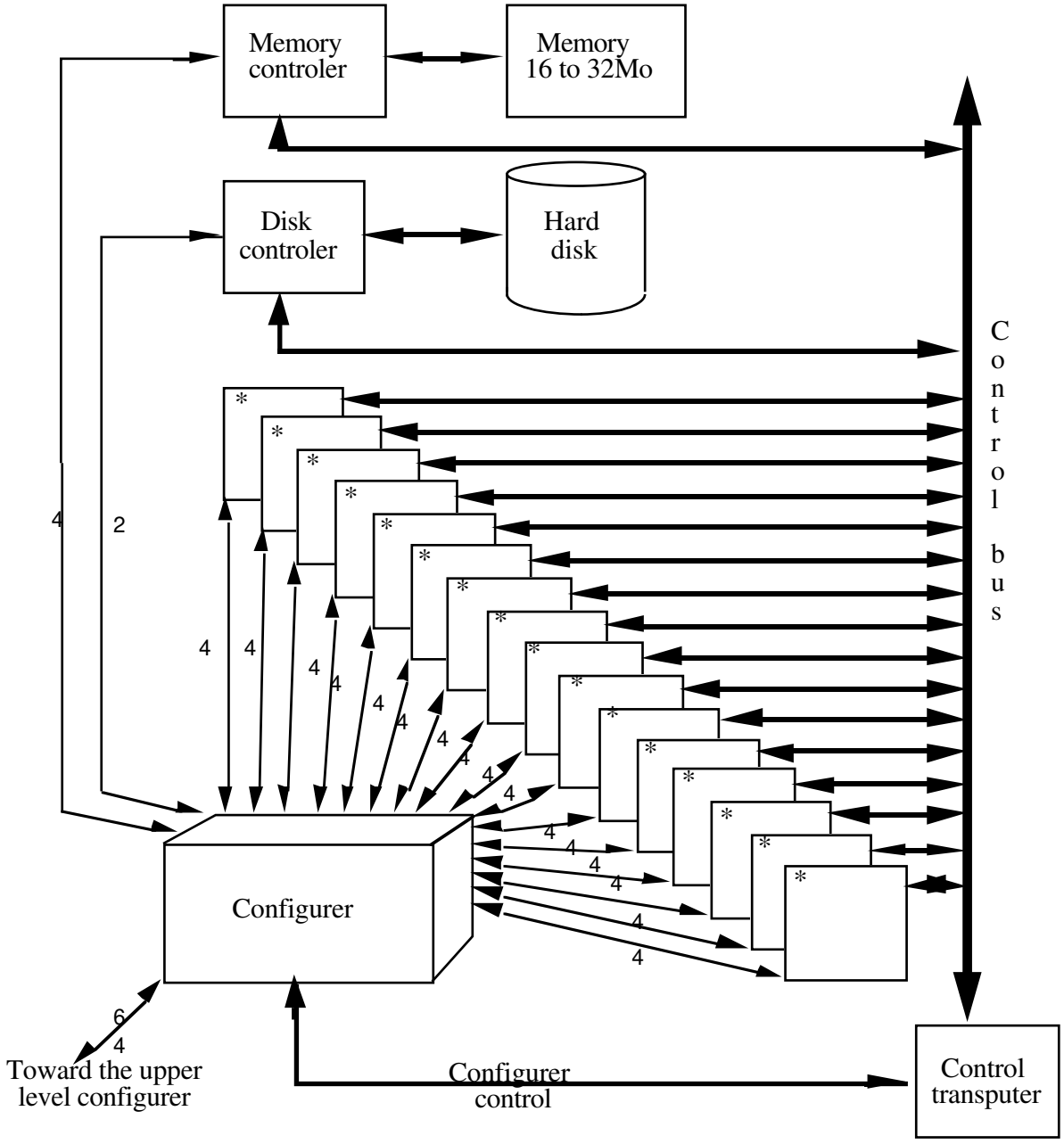

Figure 4.1.

Moreover, processors physical interconnection network can be changed dynamically. These two features can be used to provide a refined mapping between physical machine and user conceptual machines and to reduce the cost associated with interprocess communication.

Processes within a virtual machine can be directly mapped into hardware processes. As target application programs include such fine grain parallelism that instructions are considered to be processes, interprocess communication grain should be tuned accordingly. The limited number of external communication links is a bottle-neck to remote processes communication.

Reconfigurability allows physical links to be adjusted when needed. The SuperNode can then be adapted to any topological problem configuration. Dynamically adjusting network topologies involves a cost that could be prohibitive compared to introducing routing mechanisms.

The reconfigurability of the SuperNode allows any possible topological configuration of the machine while executing parallel programs. Dynamic configuration is achieved with a programmable switch device which allows modular construction of higher level SuperNode machines and ensures proper interprocessor communication within any configuration.

SuperNode machines have been developed in the ESPRIT project 1085 between 1986 and 1988.

For more details on the description of SuperNode's architecture see [Harp86] and [Muntean89].

2. OCCAM AND PARALLEL C

The underlying programming paradigm of Occam (and SuperNode) is T.C.S.P. (Theoretical Communicating Sequential Processes) [Hoare85], however the C parallel comes from C and could therefore be viewed as an extension of C.

2.1. Occam

Occam uses processes as a basic concept. There is simple processes and structured processes. Simple processes are skip, stop, affectations and communications. The skip process express a process which do nothing and terminate correctly. The stop process express a process which begin no action. It represent the deadlock. Affectation processes are the classical affectations of imperative languages. Communication processes are the transmission of a value or the reception of a value in a synchronous way.

Each process evolves independently from other processes in an asynchronous way, using its own resources. It has its own set of names of communications which can be used for interactions with its environment, with the system or with other processes. Any communication in a given process is view as instantaneous, which means that it is considered to have no duration.

The structured processes are processes combined in a sequential, in a parallel, in a condition, in an alternative or in a repetitive manner. The sequential combination of processes allow to execute the first process and if it terminate correctly then to execute the following processes. The conditional process allow to execute a process if a condition is true, if not it passes to the next condition and if there is no other condition it generates an error. The repetitive process is a process which is executed while a condition is true.

Less classical are the non-deterministic and the parallel processes. Occam intrinsic non-determinism corresponds to processes' internal choices that can not be controlled by the environment when they occur. This allows to ignore deliberately facts which influence the selection of actions and therefore preserve a high level of abstraction in the description of processes' behaviors. In the same way, Occam parallelism allows us to focus our attention on the interface between processes without being preoccupied by common variables.

2.2. IMAG C

The C parallel developed in 1988 at the IMAG institute is system development oriented. The need of dynamical structures and recursivity motivate this extension. As a C extension the concept of process is not a basic concept of parallel C. Parallelism, non-determinism and communications are C functions.

Parallelism is more dynamic than in Occam. It is possible to define a tree of processes without subterfuge. Non-determinism separates more precisely the test and the realisation of communications. Communications have the same semantic as Occam communications but it is possible to interfere between processes through common variables.

The ANSI C which is presented in the next section follow the same basic ideas of the IMAG C. It's presentation in more details is motivated by the fact that it is an available product with basic tools for parallel programming on parallel machine.

2.3. ANSI C (Parallel)

The ANSI C was developed by Inmos. It is based on the classical sequential ANSI C language plus some specific extensions inspired by the old C 3L parallel language. The parallel ANSI C tool-set is a software cross-development system for Transputers, hosted on PC/MS-DOS, Sun 3, Sun 4/SunOS or VAX/VMS systems. It consists of a full ANSI C compiler with concurency support, a multi-language linker, a configurer for mapping programs onto Transputer networks and host server tools.

Concurrent processing in parallel ANSI C conforms to the CSP model. Any C function can be defined as a concurrent process. A special set of functions is provided in the runtime library to handle concurency. Processes are instantiated by a call to ProcAlloc, using the function name as the link to process structure. This allow dynamic creation of processes.

… x = ProcAlloc (newproc, 0,3,pa1,pa2,pa3) …

Once allocated, the process is started using Procrun, ProcPar or ProPriPar :

… ProcRun (x)

Channels are a specific type Channel, which is manipulated by special dedicated functions. ChanIn and ChanOut are used to establish communication. *ChanAlloc, ChanReset and ChanInit allows, respectively, dynamical allocation of channels, reset a channel to its quiescent (non-communicating) state and initialization of a channel (no communications between processes). This allow a fine grain control of communications.

Non determinism is provided by special dedicated functions. Due to the possible presence of channels, Booleans and timers in guards (in contrast with Occam) several new control functions (compared to Occam) had to be provided.

3. NEURAL

3.1. Main characteristics of Neural.

In this paragraph we present briefly the main characteristics of Neural. The purpose of this presentation is to point out the interest of using Neural as a high level programming language for A.N.N. simulations on massively parallel machine as SuperNode. We present a simplified version of Neural.

Three main aspects are considered:

- the static characteristics, essentially related to the net's topology, concern the description of an A.N.N. as a static object;
- the dynamic characteristics concern the description of an A.N.N. as an object evolving in time;
- the execution on massively parallel machine concern the distribution of cells on processors.

3.2. Static characteristics

Artificial Neural Networks may be seen statically as a set of numerous simple processes (usually called cells) exchanging information through a huge number of connections. Neural processes can appear in first approximation as Occam processes.

Cells get the simplicity and the structuration of processes. Therefore a cell can be the description of a cluster of cells in using the parallel constructor. The behaviour of a cell can be non-deterministic.

Neural non-determinism allows to use output as input communications in a guard and supports few communications in a guard. The guard is validated in the same manner as in Occam except it could be more complicated to determine it's statue.

Processes can not interact through common variable, they must use explicit communications : connectors. The connectors are more sophisticated channels. Neural connectors offer rendez-vous, diffusion, multiple reception and group interactions. This correspond to the main interactions encountered in A.N.N. The rendez-vous connector realize a synchronous communication between two processes, and only two. The diffusion connector realize the synchronous diffusion of a message from a transmitter to many receiver. The multiple reception realize the synchronization of the receiver with one of its' transmitter. The group connectors realize the synchronous interaction between two groups of processes as in layered networks.

3.3. Dynamic characteristics

Different types of A.N.N. have very different kinds of dynamic features ranging from full synchronicity to completely asynchronous ones. An other kind of dynamic characteristics, useful in simulation, concern the modifications of the network topology.

Neural offers two type of processes. The asynchronous processes which evolve independantly from each other and the synchronous processes which are cycle processes repeating their cycle at each top of a timer. This allow to describe synchronous and asynchronous A.N.N. in the same language. Therefore all processes are able to cooperate through connectors.

Pointers introduce local dynamicity in processes. However recursivity is not allowed in the present version of Neural. Furthermore it is possible to create processes.

The creation of asynchronous processes is controled by the replicator structure. Combined with the parallel construct the replicated structure create a number of processes unknown at compilation time. The asynchronous processes could not be stopped by there environment. They can only stop correctly by there own decision, as in Occam.

Synchronous processes can be modified at any top of the timer. The timing assume the existence of a coherent global state of the synchronous processes network. The transformations laws assume the evolution of a free deadlock synchronous processes network to a free deadlock synchronous processes network.

3.4. Execution on massively parallel machine characteristics

The use of massive parallelism in order to increase the speed of simulations appears more and more as a need and a challenge to a languages such Neural, dedicated to A.N.N.. Our approach is to offer a general model through T.C.S.P. with a specific allocation language. Whenever the body of a program specifies only the behaviour of its processes the execution of all the processes is emulated on the same processor. However the body of a program can also specify physical allocation constraints.

The use of reconfigurability on a machine, like the SuperNode machine, could not actually be expressed in Neural. Reconfigurability appears very attractive on programs cut in different steps where it is possible to determine for each step the best configuration.

The variety of A.N.N. and their combinations like puzzle justify the introduction of knowledge in the allocation of processes on processors. This knowledge is mainly taken into account through four ways :

- the user can use the **standard** allocation;
- the user specifies a **criteria** per each class of processes, he has a coarse view on how to allocate his (her) processes on processors;
- the user **constraints** each processes to be on a specific processor, he wants to control precisely the allocation.

• the user can describe the processors **architecture**, he has to specified an undirected graph of processors connected by links.

Standard allocation, criteria, constraints and architecture are four aspects of the same problem and therefore are combined in a specific allocation language, associated to NEURAL We will show on the following example how to express the allocation of processes "in" and "out" on processors "1 H1", "firtslayer" on an other processor "2 H1" and n "cell" processes on two processors "3 & 4 H1" with the criteria that the n "cell" processors are structured with a grill topology.

```
PLACED PAR
        PROCESSOR 1 H1 :
         in
         out
        PROCESSOR 2 H1 :
         firtslayer
        PROCESSOR 3,4 H1,H1 grill :    -- grill is a criteria.
        PAR i = 1 FOR n
                cell ( i )

ARCHI [1..4] H1 T :
        LINKUP( 1 , host )
        LINKUP( 1 , 2 )
        LINKUP( 1 , 3 )
        LINKUP( 2 , 4 )
        LINKUP( 3 , 4 )
```

4. PRESENT STATE OF DEVELOPMENT

We have available computers, languages and programs.

The available computers are Transputers which allow to construct massive parallel machines like SuperNode since the end of the Esprit Project 1045 - SuperNode I.

The Occam 2 language, its' Transputer Development System and its' Stand-Alone development system are trade mark of Inmos and commercially available.

More recently in the Esprit Project 2528 - SuperNode II, Inmos has developed the parallel ANSI C. IMAG/LGI is working, in this context, on an operational system for parallel machine.

The NEURAL language's operational semantic has been defined (Philippe Chol's Thesis under redaction). A syntaxical analyser and a prototype compiler of Neural will be soon available on Sun 3/SunOs.

PART 5

APPLICATIONS

1. APPLICATIONS USING SMART

Since the machine SMART is still under construction, the number of applications developed for SMART is limited. For the moment, a student from Germany has been started to develop a kernel of a library in the assembly language to implement Kohonen's model and the sources separation algorithm. This library should be used to develop more applications on SMART, because it contains the basic operations deal with matrix and vectors. The major problem for these application is the unvailability of SMART.

2. APPLICATIONS USING SUPERNODE

2.1. Parallel genetic algorithm and combinatorial optimization problems

2.1.1. Synopsis

Genetic algorithms are stochastic search and optimization techniques which can be used for a wide range of applications. This section addresses the application of genetic algorithms to the graph partitioning problem. Standard genetic algorithms with large populations suffer from lack of efficiency (quite high execution time). A massively parallel genetic algorithm is proposed, an implementation on a SuperNode® of Transputers® and results of various bench-marks are given.

The parallel algorithm shows a superlinear speed-up, in the sense that when multiplying the number of processors by p, the time spent to reach a solution with a given score, is divided by kp (k>1).

A comparative analysis of our approach with hill-climbing algorithms and simulated annealing is also presented. The experimental measures show that our algorithm gives better results concerning both the quality of the solution and the time needed to reach it.

2.1.2. Introduction

Given a graph, the "graph partitioning problem" searches for a partition of its nodes which optimizes a given cost function.

There are numerous practical applications of this problem, for instance:

- design of V.L.S.I. (**V**ery **L**arge **S**cale **I**ntegration) circuits, where, given a set of components and a set of modules, we want to place the components in order to minimize the number of connections between modules, yet preserving some balance concerning the number of components on each module [Lawler69][Russo71];
- routing in distributed systems, where the considered problem is to subdivide the computer network into smaller clusters so that the control overhead for routing is minimized [Bouloutas89];
- image segmentation in the field of computer vision, where segmented images are represented as graphs in which each vertex represents a segment and each weighted edge between two vertices represents a topological relationship between two segments of the image [Hérault89];
- virtual memory paging systems, where we want to distribute the different objects on memory pages in order to minimize the number of references between objects stored on different pages [MacGregor78];
- mapping parallel programs on parallel architectures.

In our laboratory, we are especially interested in this last application, namely, the placement of communicating processes on processors of a distributed memory parallel machine. A survey of the different methods proposed in the literature to deal with this problem may be found in [Talbi90]. The parallel program is modeled as a graph where the vertices represent the processes, the vertices' weights represent known or estimated computation costs of these processes, the edges represent communication links required between them and the edges' weights estimate the relative amount of communication necessary along those links. When the number of processes exceeds the number of available processing elements, as it is usually the case in massively parallel programming, the mapping problem includes the contraction problem [Berman87] which is equivalent to the graph partitioning problem treated in this part.

The graph partitioning problem is NP-complete.

Consequently, heuristic methods should be used to deal with it. They may find solutions that are only approximations of the optimum, but they will do it in a reasonable amount of time. The different approaches that have been proposed for this problem may be divided in two main classes. On one hand, the general purpose optimization algorithms independent of the given application and, on the other hand, the heuristic approaches especially designed for a unique problem. As we want to avoid the intrinsic disadvantage of the algorithms of this second class (their limited applicability due to the problem dependence) our concern in this paper, is only the first class of algorithms.

Two widely used optimization techniques are the hill-climbing algorithm [Haden88] and simulated annealing [Sheild87]. Hill-climbing is sure to find the global minimum only in convex spaces. Otherwise, most often it is a local rather than a global minimum which is found. Simulated annealing offers a way to overcome this major drawback of hill-climbing but the price to pay to do so is an important computation time. Worst, simulated annealing algorithm is rather of a sequential nature, its parallelization is quite a difficult task [Baiardi89][Savage90].

More distributed optimization techniques may also be considered. Some of them are closely related to neural networks algorithms (see [Ackley87] & [Peretto90]). Others, namely, genetic algorithms (GAs) are considered in this paper. They are stochastic search techniques, introduced by Holland twenty years ago [Holland75], inspired by biological evolution of species. Development of massively parallel architectures made them very popular in the very last years. They have recently been applied to combinatorial optimization problems in various fields, such as, for instance, the travelling salesman problem [Grefenstette87], the optimization of connections and connectivity of neural networks [Whitley90], and classifier systems [Robertson87].

The purpose of this section is to prove that the graph partitioning problem may be solved quite efficiently by a parallel genetic algorithm.

The structure of the section is as follows:

- in § 2.1.3. we give a mathematical formalization of the graph partitioning problem and discuss few instances of classical cost functions.
- in § 2.1.4. we extensively present the genetic algorithm approach to the graph partitioning problem. After a recall of the genetic algorithms principles, we show how they can be used to deal with the graph partitioning problem, we discuss the matter of parallel genetic algorithms and finally we expose the proposed solution.
- the final sub-section (§ 2.1.5.), after a detailed presentation of the SuperNode implementation, presents the results of different bench-marks comparing either speed or quality of results of this algorithm given different sizes of population or given different numbers of processors. A comparative analysis of the genetic algorithm solution with hill-climbing and simulated annealing ones is also presented.

Finally, concluding remarks and possible extensions to this work are proposed.

2.1.3. Mathematical formalization of the graph partitioning problem

Given:

- an undirected graph $G = (V,E)$;
- an application Ω_1 from V into $\mathbf{Z}^+$, such that $\Omega_1(v_i) = w_{1i}$ is the weight of vertex v_i;
- an application Ω_2 from E into $\mathbf{Z}^+$, such that $\Omega_2(e_j) = w_{2j}$ is the weight of edge e_j;
- and a set of numerical constraints $\Phi = \{\phi_1, \phi_2, ..., \phi_m\}$ on these weights;

the graph partitioning problem has to find a partition Π of V ($\Pi = \{\pi_1, \pi_2, ..., \pi_n\}$) satisfying the constraints Φ.

A classical and well studied set $\Phi 1$ of constraints expresses that:

- for each sub-set π_i of V belonging to the partition Π, the sum of the weights of its vertices must be inferior to a given value B

$$\forall\ \pi_i \in \Pi, \qquad \sum_{v \in \pi_i} \Omega_1(v) \leq B$$

- the sum of the weights of edges going from one node of π_i to one node of some other π_j must be inferior to some given value C

$$\sum_{e\ \in\ \varepsilon} \Omega_2(e) \leq C$$

$$\text{with } \varepsilon = \{\ (x,y)\ /\ (x,y) \in E\ \&\ x \in \pi_i\ \&\ y \in \pi_j\ \&\ i \neq j\ \}$$

The graph partitioning problem under constraints $\Phi 1$ has been proved NP-complete (see [Garey79]).

Most applications correspond to the following set $\Phi 2$ of constraints where the weights of all nodes are set to 1 (see [Kernighan70] & [Feo90]):

- for each sub-set π_i of V belonging to the partition Π, the number of nodes in π_i is equal to a given value B_i

$$\forall\ \pi_i \in \Pi, \qquad \sum_{v \in \pi_i} \Omega_1(v) = B_i$$

$$\text{with } \forall\ v \in V,\ \ \Omega_1(v)=1$$

- the total cost of the edges going from one π_i to another π_j should be minimum

$$\text{MIN}\left(\sum_{e \in \varepsilon} \Omega_2(e)\right)$$

The graph partitioning problem under constraints $\Phi 2$ has also been proved NP-complete (see [Hyafil73]).

For our application, the mapping of parallel programs on parallel architectures, we have to consider the following set $\Phi 3$ of constraints:

- minimize the sum of the total communication costs between processors (total cost of the edges going from one π_i to another π_j) and of the variance of the loads of the different processors (variance of cost of vertices belonging to a given π_i):

$$\text{MIN}\left(\sum_{e \in \varepsilon} \Omega_2(e) + (K * \left(\frac{\sum_{\pi_i \in \Pi} \left(\sum_{v \in \pi_i} \Omega_1(v)\right)^2}{|\Pi|^2} - \left(\frac{\sum_{v \in V} \Omega_1(v)}{|\Pi|}\right)^2\right))\right)$$

With K=0 the set of constraints $\Phi 3$ reduces to $\Phi 2$. This proves that the partitioning problem under constraints $\Phi 3$ is NP-complete.

For the mapping problem, K is the weight of the contribution of the communication cost relative to the computational load balance across the system. Choosing a suitable value for K depends on knowledge about characteristics of the parallel architecture. Very small values of K would suggest a uniprocessor solution, and very large values would reduce the problem to one of multi-processor scheduling without communication costs. The parallel architecture used was a network of Transputers and K=2 has been chosen.

2.1.4. Genetic algorithm solution to the graph partitioning problem

a - Genetic algorithms principles and their application to the graph partitioning problem

Genetic algorithms compose a very interesting family of optimization algorithms. Their basic principle is quite simple.

Given a search space Σ of size M^N, given M symbols, any point of this space may be represented by a vector of N of these M symbols.

Given a fitness function F from Σ into $\Re$ associating a real value to any point of Σ.

Given an initial set of vectors, called the initial population.

Some genetic operators are used to generate new points of Σ given some old ones in a phase of the process called "reproduction". During this phase, some points of Σ are replaced keeping the size of the population fixed. The fundamental principle of GA is: "the fitter a vector, the most probable its reproduction". In mathematical terms it means that the probability P of reproduction is increasing as F increased:

$$\forall\ \sigma_1,\sigma_2 \in \Sigma, \qquad F(\sigma_1)>F(\sigma_2) \ \Rightarrow\ P(\sigma_1)>P(\sigma_2)$$

The standard genetic algorithm is:

Generate a population of random individuals.
While number_of_generations ≤ max_number_of_generations Do
Evaluation - assign a fitness value to each individual.
Selection - make a list of pairs of individuals likely to mate, with fitter individuals listed more frequently.

Reproduction - apply genetic operators to the selected pairs. New individuals produced constitute the new population.

The most common genetic operators used during reproduction are cross-over and mutation. Cross-over, given two vectors, cut them both at the same random point and exchange the two portions thus cut out (fig. 5.1a). Mutation is simply flitting a bit (fig. 5.1b). Two parameters need to be defined: Pc and Pm. They represent respectively the probability of application of the cross-over and mutations operators. Other genetic operators may be find in the literature, for instance, the inversion operator [Cohoon87] and many variants of the cross-over operator designed for specific problem domains [Shahookar90].

(a) Cross-over (b) Mutation

Figure 5.1. Genetic operators.

To use genetic algorithm for the graph partitioning problem, the following formalism is used:

Let us suppose that we have a graph of N nodes to divide into M sub-sets. Each of these sub-sets is labelled by a symbol (for instance an integer between 0 and M-1). A given partition is represented by a N vector of those symbols; where symbol p in position q means that node q of the graph is in sub-set p.

The fitness function F is the last cost function described in section II.

We use the usual version of cross-over, but mutation is a random trial of one of the M possible symbols.

b - A parallel genetic algorithm

Two approaches to parallel genetic algorithms have been considered so far:

- standard parallel approach: In this approach, the evaluation and the reproduction are done in parallel. However, the selection is still done sequentially, because parallel selection would require a fully connected graph of individuals as any two individuals in the population may be mated [Macfarlane90].

- decomposition approach: This approach consists in dividing the population into equal size sub-populations. Each processor runs the genetic algorithm on its own sub-population, periodically selecting good individuals to send to its neighbors and periodically receiving copies of its neighbors' good individuals to replace bad ones in its own sub-population [Pettey87][Tanese87]. The processor neighborhood, the frequency of exchange and the number of individuals exchanged are adjustable parameters.

The standard parallel model is not flexible in the sense that the communication overhead grows as the square of population's size. Therefore, this approach is not adapted to distributed memory architectures, where the cost of communication has a great impact on the performance of parallel programs. In the decomposition model, the inherent parallelism is not fully exploited as treatment of sub-populations may be further decomposed. This approach should be considered only when the number of available processors is less than the required size of the population.

Considering massively parallel architectures with numerous processors, we chose a fine-grained model, where the population is mapped on a connected processor graph like a grid, one individual per processor. We have a bijection between the individual set and the processor set. The selection is done locally in a neighborhood of each

individual. Another version to this approach has been already proposed in [Mühlenbein89], where at each generation a hill-climbing algorithm is executed for each individual in the population.

The choice of the neighborhood is the adjustable parameter. To avoid overhead and complexity of routing algorithms in parallel distributed machines, a good choice may be to restrict neighborhood to only directly connected individuals.

The parallel genetic algorithm proposed is:

Generate *in parallel* a population of random individuals.
While number_of_generations ≤ max_number_of_generations Do
- Evaluate *in parallel* each individual.
- Each individual reproduces *in parallel* with its neighbors.
- Do *in parallel* a selection of best local offsprings.

It is important to notice that these modifications of the standard model do not cause a degradation in the search efficiency of the standard genetic algorithm as shown in [Anderson90] and [Mühlenbein88].

2.1.5. SuperNode implementation and benchmarking

a - SuperNode Implementation

The SuperNode is a loosely coupled, highly parallel machine based on Transputers (fig. 5.2.). One of its most important characteristics is its ability to dynamically reconfigure the network topology by using a programmable VLSI switch device. This architecture offers a range of 16 to 1024 processors, delivering from 24 to 1500 Mflops performance. To achieve these performance, a hierarchical structure has been adopted [Jesshope86]. The basic component is a T800 Transputer. It is a 32-bit microprocessor, with on-chip memory and F.P.U. (**F**loating **P**oint **U**nit), delivering 10Mips and 1.5Mflops peak performance. Communication between Transputers is supported by 4 bidirectional, serial, asynchronous, point-to-point connection links. A Sun host station is used to provide the connection between the root processor and the external world.

The programming environment used in our experiments is Parallel C 3L. A configurator of the physical network, developed in our laboratory has been used to obtain the desired topology of the architecture.

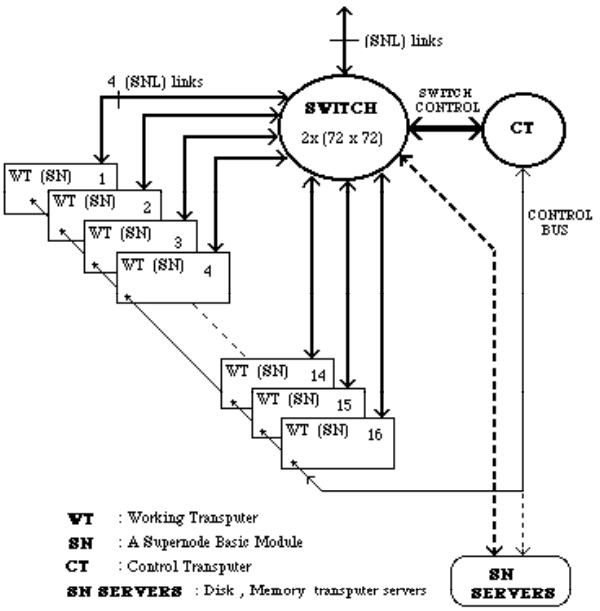

Figure 2 Supernode Parallel Architecture.

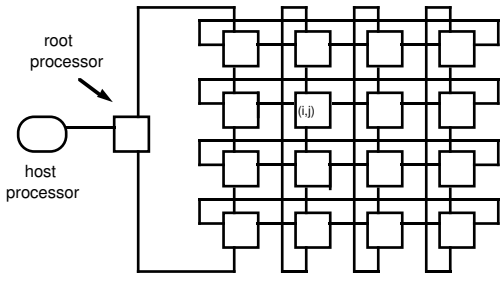

Figure 3 A torus of 16 processors.

We assume that each individual in the population resides on a processor and communication is carried out by message passing. The following is a pseudo-Occam description of the process executed in parallel by each processor:

```
SEQ
Generate (local_individual)
Evaluate (local_individual)
While  number_of_generations ≤ max_number_of_generations
    SEQ
       PAR  i=0  FOR number_of_neighbors
           PAR
              neighbor[i] ? neighbor_individual[i]
              neighbor[i] ! local_individual
       PAR i=0 FOR number_of_neighbors
          Reproduction (local_individual, neighbor_individual[i])
       Replacement
```

Each reproduction produces two offsprings. Our strategy is to choose randomly one of the offsprings. The replacement phase consists in replacing the current local individual with the best local offspring produced in the reproduction phase.

The population is placed on a torus. Given the four links of the Transputer, each individual has four neighbors. No routing is needed in the processor network because only directly connected processors have to exchange information.

We do not consider the best solution found globally since the communication involved in determining this solution would be considerable. We only pick up the best solution routing through a "spy process" placed on the "root processor" (see fig. 5.3.).

b - Varying the number of processors

The purpose of this bench-mark is to measure the speed-up when running the genetic parallel algorithm (for a given population size) on different sizes of torus of processors.

We use the speed-up ratio as a metric for the performance of the parallel genetic algorithm. The speed-up ratio S is defined as S=Ts/Tp where Ts is the execution time on a single processor and Tp corresponds to execution time for a p processors implementation. Figure 5.4. shows the obtained results.

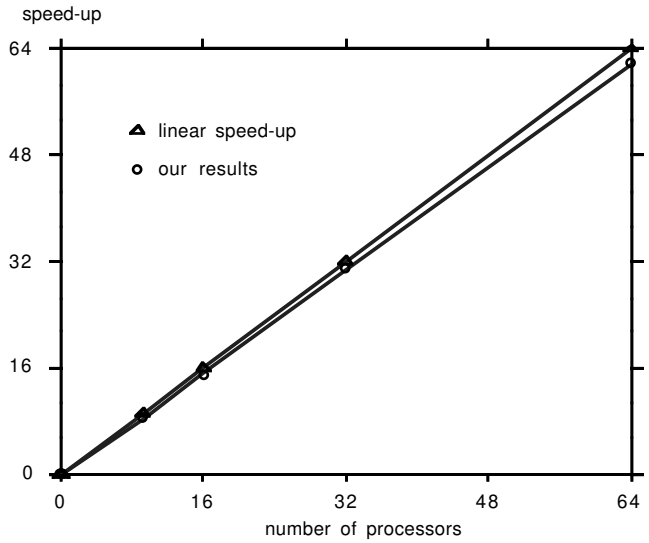

Figure 5.4. Speed-up of the parallel algorithm.

The algorithm has a near-linear speed-up. This is due to the fact that the communication cost between processes is relatively small compared with the computation cost, and is independent of the size of the architecture.

c - Varying the population size

The purpose of this bench-mark is to measure the evolution of solution's quality when running the parallel genetic algorithm with different sizes of population.

Figure 5.5. shows the obtained results. Notice that given the specific graph partitioning problem used (a pipeline of 32 vertices to be partitioned in 8 sub-sets) for this bench-mark, the best possible solution scores 7.

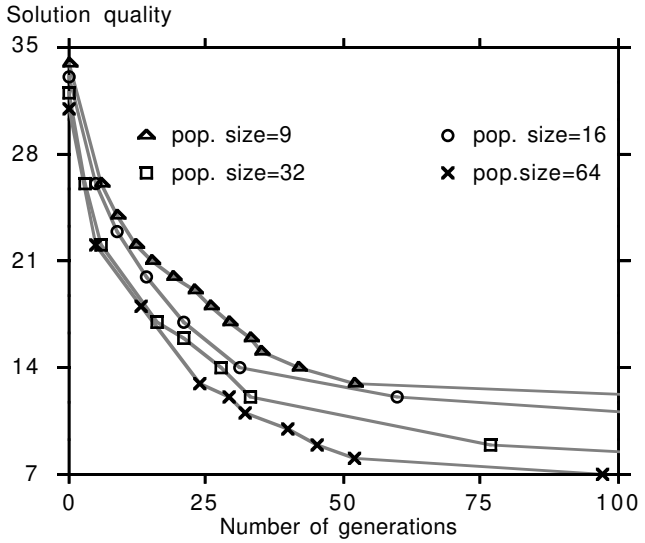

Figure 5.5. Solution quality function of population size.

As expected, for a given number of generations, the quality improves with an increase of population's size.

It may even happen that for a too small population a premature convergence occurs and that the optimal solution will not be ever reached.

The figure 5 shows also that the greatest reduction in the cost of the partitioning occurs at the beginning. Thus a moderate quality partitioning can be obtained very quickly.

d - Time to reach a given solution

The purpose of this bench-mark is to study the speed-up for a given solution's quality when running the genetic parallel algorithm on different sizes of torus of processors and with different sizes of populations (both being equal given that there is one individual per processor).

Figure 5.6. shows the influence of the number of processors (and population size) on the time needed to reach a solution scoring 8.

number of processors	CPU time
9	304
16	164
32	71
64	33

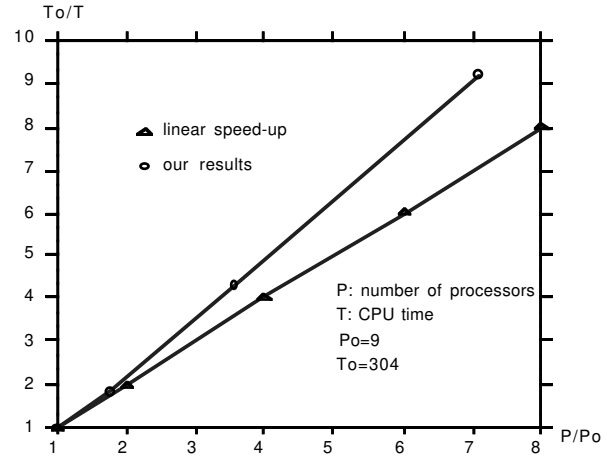

Figure 5.6. Execution times on different sizes of parallel architectures.

We have a "superlinear" speed-up of the parallel genetic algorithm, in the sense that when multiplying the number of processors by p the execution time is divided by kp (k>1).

e - Comparison with hill-climbing and simulated annealing algorithms

In this section, the performance of the proposed algorithm is compared with hill-climbing and simulated annealing algorithms.

Hill-climbing algorithm

The hill-climbing algorithm starts with a random configuration, and tries to improve it [Johnson85]. The improvement is carried out in small steps consisting of moving a vertex from one sub-set to another. A move is selected randomly, the cost change of the move is evaluated, and if the change is for the better the move is accepted and a new configuration is generated. Otherwise, the old configuration is kept. This process is repeated until there are no changes to the configuration that will reduce the cost function further. When this occurs a local minimum has usually been found, rather than the required global minimum. Figure 5.7. shows a version of the hill-climbing algorithm.

- Generate a random initial state S0 (S:=S0).
- Repeat
  - compute at random a neighboring state S'.
  - if cost(S') < cost(S) then S:=S'

  Until there is no better neighbor.

Figure 5.7. The hill-climbing algorithm.

Simulated annealing algorithm

The principle of the simulated annealing algorithm is the following [Kirkpatrick83] (fig. 5.8.): the system is put in a high temperature environment. At this temperature is applied a sufficiently long sequence of random elementary transformations (Markov chain) to reach the equilibrium at this temperature. Then, the temperature is slightly decreased and a new sequence of random move is applied. At each temperature the permitted energy states are governed by the metropolis criterion, which allows the configuration to accept a state with a probability $P(\Delta E,T)$. The search terminates when the system stabilizes.

There is a large amount of literature on this topic [Laarhoven87], and the basic algorithm allows for considerable variation and tuning of parameters. The version we considered is a straightforward version, which can probably

be greatly improved. However, as the parallel genetic algorithm is also a "naive", "out-of-the-shell" version, we think that the following bench-marks give, at least, interesting order of magnitudes.

The number of available changes to the configuration, denoted by L, when moving one vertex to a different sub-set, is given by L=N*(M-1), where N is the number of vertices of the graph to be partitioned and M the number of sub-sets of the partition. This value gives a measure of the size of the problem and is used as a parameter in the annealing schedule.

1. (Initialization step)
   - start with a random initial configuration S0 (S:=S0);
   - T := Tmax;
2. (Stochastic hill-climb)
   - generate and compute a random neighboring state S'; ΔE:=cost(S')-cost(S)
   - select the new configuration (S:=S') with probability P(ΔE,T)=min(1,exp(-ΔE/T));
   - repeat this step X*N*(M-1) times; /* length of the Markov chain spent at each T */
3. (Anneal/Convergence test)
   - set T:=aT;
   - if T≥Tmin goto step2.

Figure 5.8. The simulated annealing algorithm.

Experimental protocol and results

Each algorithm was run 10 times to obtain an average performance estimate. Experiments were performed on two different problems:

- a pipeline of 32 vertices to be partitioned in 8 sub-sets;
- and a grid of 64 vertices to be partitioned in 4 sub-sets.

For the genetic algorithm, we use a population of 64 configurations running on a 8 by 8 torus of processors. The annealing schedule that have been used during our experiments are given by table 5.1..

Symbol	Value	Description
Tmax	10	Starting temperature
Tmin	0.1	Minimum temperature
a	0.9	Temperature decay rate
X	2	Lenght of the markov chain

Table 5.1. The annealing schedule.

The tables below show the minimum, maximum, average value and the variance of the obtained solutions. The results for the hill-climbing and the simulated annealing algorithms are based on an implementation on a single T800 Transputer.

Algorithm	Solution				CPU time
	min	max	mean	deviation	(sec)
Hill-climbing	7.5	12.5	9.9	1.94	353
Simulated annealing	7.5	9	8.05	0.37	2296
Genetic algorithm	7	8	7.5	0.15	64

Table 5.2. Bench-marking with a pipeline of 32 vertices and a partition of 8 sub-sets.

Algorithm	Solution				CPU time
	min	max	mean	deviation	(sec)
Hill-climbing	24	47	32.9	42.49	643
Simulated annealing	21	27	24	4.00	3358
Genetic algorithm	17	25	21	6.25	93

Table 5.3. Bench-marking with a grid of 64 vertices and a partition of 4 sub-sets.

It can be observed from tables 2 and 3 that the genetic algorithm outperforms hill-climbing and simulated annealing algorithms both in the quality of the solution and the time used in the search.

2.1.6. Conclusions and future directions

A parallel genetic algorithm to solve the graph partitioning problem has been proposed and evaluated.

The main results are the following:

- the algorithm shows a superlinear speed-up;
- it is easy to program;
- it is simple to implement on massively parallel distributed memory architectures;
- it outperforms hill-climbing and simulated annealing algorithms both in the quality of the solution and the time used to reach it.

An important characteristic of genetic algorithms is that they may be used to solve a great variety of combinatorial optimization problems. We are using them to solve such optimization problems in the field of robot control and neural networks.

We are also studying an important improvement of the algorithm, namely, the dynamic variation of its parameters and particularly mutation probability. The cross-over operator becomes less effective over time as the strings in the population become more similar. One way to avoid the premature convergence and to sustain genetic diversity is by using adaptive mutation. During the first generations when there is ample diversity in the population, mutation must occur at very low rates. However, as diversity decreases in the population, the mutation rate must increase.

More theoretical work is planned: an cellular automata based model will be used to study the influence of the algorithm's parameters on its convergence

2.2. A parallel genetic Boltzmann machine

2.2.1. Synopsis

We proposed in a previous work [Bessière90e] **Probabilistic Inference** as a possible underlying theory or mathematical metaphor for numerous works in the field of formal neural networks.

Probabilistic Inference suggests that any cognitive problem may be split in two **optimization problems**. The first one called the "dynamic inference problem" is an abstraction of "learning", the second one, namely, the "static inference problem", being a mathematical metaphor of "pattern association".

In this previous paper, Boltzmann machines have been shown to be a special case of probabilistic inference, where the two optimization problems are dealt with using simulated annealing [Kirckpatrick83] for the pattern association part and using simple gradient descent for the learning one.

It was, then, suggested that other optimization techniques should be considered in that context and especially **genetic algorithms**. **The purpose of this section is to describe the investigations made about that question using a parallel genetic algorithm.**

We will first recall the principles of probabilistic inference, then, we will present briefly the parallel genetic algorithm and the ways it is used to deal with both optimization problems, to finally conclude about future extensions of this work.

2.2.2. Probabilistic inference

Probabilistic inference may be seen as the present state of the art resulting of numerous works aim to use probability theory as a model of inference and decision making in an incomplete and uncertain universe.

In probabilistic inference theory, knowledge is represented using the usual formalism of probability theory. Knowledge is encoded using a set $\Xi = \{X_1, ..., X_n\}$ of variables. The value space of Ξ is called Ω. The basic assumption is that **a knowledge state of a cognitive system is a probability distribution P over Ω.**

Probabilistic inference has to deal with two different problems:

1 - given a knowledge state (a probability distribution P) and some new information (a set of constraints Φ on Ξ), how to infer a new knowledge state (a probability distribution Q) taking into account the new information;

2 - given a knowledge state (a probability distribution P) and values of some of the variables of the set Ξ, how to infer the most probable values (according to P) of the other variables.

The first problem may be called the "dynamic inference problem" because it concerns how the knowledge state changes in order to take into account new information, while the second problem, the "static inference problem", concerns the consistency conditions of a knowledge state at a given time (see [Hunter86]).

a - Dynamic inference problem

Given a set of variables $\Xi = \{X_1, ..., X_n\}$, given Ω its value space, given a prior knowledge state (a probability distribution P) and given some new information (a set of constraints Φ on Ξ); the dynamic inference process has to find a posterior knowledge state (a probability distribution Q).

In the general case, there is an infinity of probability distributions which are potential solutions of this problem. However, all the probability distributions are not equivalent. Some appear to be more "coherent", more "probable", more "interesting" than some others. The function H(Q,P) (called Kulbach entropy, relative entropy

and Φ,

or cross entropy) is a way of measuring the "interest" of a given probability distribution Q relative to P the smaller H the better Q. H is defined by: $H(Q,P) = + \int_{\Omega} Q(\omega) \log \frac{Q(\omega)}{P(\omega)} d\omega$ [f.1].

According to this, the dynamic inference problem may be restated as follow: given P and Φ, find the probability distribution Q which **minimizes** H. It can be shown that if Φ is a consistent set of constraints, there is one and only one solution Q* to this problem. Finding Q* is not a trivial mathematical problem.

However, for a very important class of problems, where Φ takes the form of a set of real functions ($\Phi = \{f_1, ..., f_m\}$) such that the mean value a_i ($A = \{a_1, a_2, ..., a_m\}$) of every function f_i is known, then it can be shown that the solution take the following form: $q^*(\omega) = \frac{1}{Z^*} e^{-\sum_{i=1}^{m} \lambda_i f_i(\omega)}$ [f.2] where q* is the density of Q*, λi are the Lagrange multipliers and Z* is a normalizing constant.

Let us take an example: given two variables A (the Age of the captain) and L (the Length of the boat), the problem we want to solve is find A given L or find L given A.

We have: Ξ = {A,L}; Ω = [7,77] x [4,444].

Starting from scratch (no prior information), P_0, the initial knowledge state, is a uniform distribution over Ω (figure 1). Let us suppose that we first learn the mean value of A ($E(A) = m_A$) and the variance of A ($E((A - m_A)^2) = \sigma_A^2$). An infinity of probability distributions over Ω have this mean value and this variance. However one and only one minimizes H: the normal distribution P_1 having this mean value and this variance as parameters (figure 2). If we then learn the mean value and variance of L, by the same process, we get the probability distribution P_2 (figure 3). Iterating this process for all the data we can get about our problem, and especially for information expressing correlations between A and L we will finally get a probability distribution Q which will sum up all the previously acquired information (figure 4).*

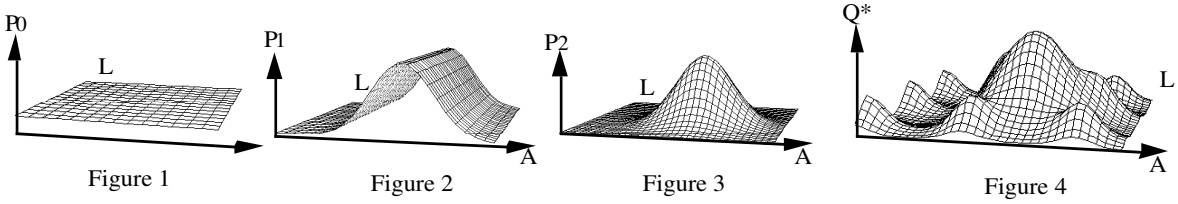

Figure 1 Figure 2 Figure 3 Figure 4

b - Static inference problem

Given a set of variables $\Xi = \{X_1, ..., X_n\}$, given Ω its value space, given P a probability distribution over Ω and given values of some of the variables of the set Ξ; the static inference process has to find the most probable values of the unspecified variables of Ξ.

According to the previous paragraph, the interesting cases to consider are those cases where P takes form [f.2]. Therefore, finding the most probable values of the non specified variables (i.e. maximizing P) corresponds to the **minimization** of the function: $U(\omega) = \sum_{i=1}^{m} \lambda_i f_i(\omega)$ [f.3] over the sub-space of Ω defined by the given values on Ξ.

Getting back to our example, this means that for a given value a of the variable A we want to find the most probable value l of the variable L. This process may be visualized by looking for the maximum of the curve defined by the intersection of Q and the vertical plane corresponding to A = a (this curve is in bold on figure 5).*

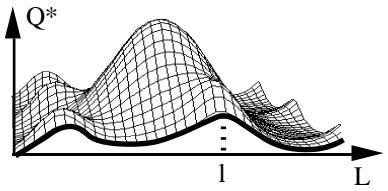

Figure 5

c - Boltzmann machine

A usual simplification of probabilistic inference consist in restricting the problem to cases where the X_i are binary variables and where the f_i depend on no more than two variables: $f_i(X_1, ..., X_n) = \alpha_i X_k X_l + \beta_i X_k + \gamma_i X_l + \delta_i$ [f.4]. In that case we get: $Q^*(\omega) = \frac{1}{Z^*} e^{-\sum_{i<j} W_{ij} X_i X_j - \sum_i W_i X_i}$ [f.5] where Q* is a 1-Gibbs distribution. Such distributions have well known dynamics with nice simple properties. This is the choice made, for instance, for Hopfield nets [Hopfield82] and for Boltzmann machines [Ackley88].

The dynamic inference problem has new parameters, the "weights" W_{ij}, in place of the old ones, the Lagrange multipliers λ_i. The search space, which was the space of all probability distributions over Ω, is approximated by the space of weights' values. The dynamic inference process is replaced by a gradient descent dynamic in the space of the weights' values with H as objective function. One of the remarkable properties of the Gibbsian neural nets is that this gradient descent may be done, without any explicit computation of H, using simply classical local adaptation rules (for instance, Widrow-Hoff's rule for Boltzmann machines) on the system at "thermodynamic" equilibrium (see [Ackley88]). Reaching thermodynamic equilibrium itself is strictly equivalent to treating the static inference problem and is done using simulated annealing. U is the objective function but for Gibbsian neural nets, U has the exact form of an energy function.

2.2.3. Parallel genetic algorithm

Genetic algorithms compose a very interesting family of optimization algorithms. Neither their principles, nor their application can be presented here but a very complete description may be found in [Goldberg89]. However, let us recall that the standard genetic algorithm has the following form:

Generate a population of random individuals.
While number_of_generations ≤ max_number_of_generations Do
Evaluation - assign a fitness value to each individual.
Selection - make a list of pairs of individuals likely to mate, with fittest individuals listed more frequently.
Reproduction - apply genetic operators to the selected pairs. New individuals produced constitute the new population.

Genetic algorithms are quite easy to implement on parallel machines. Two main approaches, to do so, have been considered so far:

- **standard parallel approach**: In this approach, the evaluation and the reproduction are done in parallel. However, the selection is still done sequentially, because parallel selection would require a fully connected graph of individuals as any two individuals in the population may be mated [Macfarlane90].
- **decomposition approach**: This approach consists in dividing the population into equal size sub-populations. Each processor runs the genetic algorithm on its own sub-population, periodically selecting good individuals to send to its neighbors and periodically receiving copies of its neighbors' good individuals to replace bad ones in its own sub-population [Pettey87][Tanese87]. The processor neighborhood, the frequency of exchange and the number of individuals exchanged are adjustable parameters.

Considering massively parallel architectures with numerous processors, namely, a SuperNode of Transputers, we chose a fine-grained model, where the population is mapped on a connected processor graph like a grid, one individual per processor. We have a bijection between the individual set and the processor set. The selection is done locally in a neighborhood of each individual. The choice of the neighborhood is the adjustable parameter. To avoid overhead and complexity of routing algorithms in parallel distributed machines, we chose to restrict neighborhood to the only four directly connected individuals. The parallel genetic algorithm proposed is:

Generate *in parallel* a population of random individuals.
While number_of_generations ≤ max_number_of_generations Do
Evaluation - Evaluate *in parallel* each individual.
Reproduction - Each individual reproduces *in parallel* with its four neighbors.
Selection - Do *in parallel* a selection of best local offsprings.

Another version to this approach has been already proposed in [Mühlenbein89], where, furthermore, at each generation a hill-climbing algorithm is executed for each individual in the population.

Detailed description of this work and various bench-marks may be found in [Talbi91a], [Talbi91b] and [Talbi91c]. This parallel genetic algorithm implemented on a SuperNode of Transputers shows a remarkable **"superlinear" speed-up**, in the sense that when multiplying both the number of processors and the size of the population by p, the execution time to reach a given quality of solution, is divided by kp (with k>1).

2.2.4. A parallel genetic Boltzmann machine

To deal with the static inference problem we replace the usual simulated annealing of Boltzmann machines by our parallel genetic algorithm. We work with a population of 64 individuals, each one being a vector of bits X_i representing the activities of the cells of the network. The search space is 2^N, where N is the number of cells in the network. The fitness function is the global energy function of the network $E = -\sum_{i<j} W_{ij}X_iX_j$. When the Boltzmann machine algorithm searches in parallel on the state of each cell, our algorithm searches in parallel on 64 global states of the network.

The very first comparisons between the two algorithms, made with single processor versions, show same run times' order of magnitude. It should however be noticed that the used version of the parallel genetic algorithm is a very simple and preliminary one, that may certainly be greatly improved. The same super-linear speed-up than for other applications was measured when running the algorithm on more processors. **A very remarkable result is that both algorithms found the same attractors or, at least, different attractors but with the same fitness values. This may be considered as an indirect "clue" that both algorithms converge on global optima** and do not get stuck in local optima because, in that last case, there would be no reasons for both of them to find two optima with the same fitness values.

To deal with the dynamic inference problem we replace the usual gradient descent of Boltzmann machines by our parallel genetic algorithm. We work with a population of 64 individuals, each one being a vector of integers representing the weights of the network. The search space is p^M, where M is the number of connections in the network and p the range of the weights. The fitness function depends on the specific application treated. It evaluates how good is a certain vector of weights, by averaging results obtain when treating the static inference problem on a set of different cases. When the Boltzmann machine algorithm searches sequentially the space of weights, our algorithm searches in parallel on 64 sets of possible weights.

The work is not advanced enough to draw any conclusions on our preliminary bench-marks on the dynamic inference problem. However, we may say that **the genetic parallel Boltzmann machine did learn** simple encoder problems [Ackley88].

2.2.5. Conclusion and perspectives

One aim of this work was to prove that formal neural networks could work using genetic algorithms to deal with the two optimization problems they have to face. This aim has been reached. **It suggests a totally new way to implement neural networks on parallel machines.** This seems to be a very promising

research track and will certainly be an important concern for us in the near future. We especially intend to improve our parallel genetic algorithm.

Much more work has to be done to really answer the question: "What optimization techniques should be used for the different optimization problems of the various formal neural networks models?". We expect to do more benchmarking to compare the different parts of the two versions of the Boltzmann machine and we also plan to try to apply our parallel genetic algorithm to other neural network models' optimization problems.

Finally, we hope that we will be able to propose **a cognitive algorithmic model inspired by probabilistic inference and using genetic optimization techniques**. We intend to test such a model on action-perception problems for "simple" (insect like) mobile robots.

PART 6

CONCLUSIONS AND PERSPECTIVES

The development of this task reached a point where all the preliminary work has been done in order to realize a high level specification language for Artificial Neural Networks.

The feasibility of such a language has been proved.

However, It appears obvious that in order to get a real implementation of such a language, susceptible to be either distributed or commercialize a considerable effort should still be done. This kind of work is a matter for a precompetitive industrial type ESPRIT project and not any longer for a B.R.A. type project.