An Abstract Machine for Implementing Connectionist and Hybrid Systems on Multi-processor Architectures*

Yannick Lallement[†] Thierry Cornu[‡] Stéphane Vialle[§]
lallemen@loria.fr cornu@di.epfl.ch steph@ese-metz.fr

September 1993

Abstract

The interest in new hybrid AI models, both symbolic and numeric, is currently increasing due to the complementary capabilities of these models. We present here the Cellular Abstract Machine (CAM), a tool for building such hybrid systems, taking the form of heterogeneous networks of cooperating agents (here called *cells*). Several AI applications have been written using the CAM, including a sample hybrid system. The CAM is implemented over a parallel architecture (a Transputer network). We give here the basic principles of the parallel implementation.

1 Introduction

This paper presents the Cellular Abstract Machine (CAM). The CAM is a parallel abstract machine dedicated to highly parallel artificial intelligence models like connectionist models and hybrid systems.

The paper covers:

- An introduction to hybrid models
- A description of the computational model underlying the CAM
- An example of hybrid system implementation on the CAM
- A brief overview of the implementation of the CAM on a multi-processor architecture.

*To appear in: *Parallel Processing for Artificial Intelligence, 2*, editor: H. Kitano,

[†]Crin-CNRS-Inria Lorraine, BP 239, F-54506 Vandoeuvre-les-Nancy

[‡]Swiss Fed. Inst. of Technology, EPFL DI, MANTRA Group,CH-1015 Lausanne

[§]Supélec, Campus de Metz, 2 rue Edouard Belin, F-57078 Metz cedex 3

2 Hybrid systems for new AI applications

AI world consists of two main paradigms: Symbolic AI *versus* Connectionist AI. These paradigms appear to be quite complementary, as demonstrated in many studies ([3] [11]): where the symbolic paradigm is weak, the connectionist paradigm is strong, and *vice-versa*. Symbolic models (SM) easily handle deductive reasoning (e.g., expert systems) and knowledge representation (facts and rules), have good explanation capabilities, can handle sequential, step-by-step processing. However, symbolic models have trouble with learning, with noisy inputs, with generalization. Connectionist models (CM), on the opposite, have inherent learning capabilities, do not care of noise, can easily generalize, and have a graceful performance degradation. Their weakness mainly concerns knowledge representation and rule manipulation. [16] and [2] give tracks about how neural nets can manipulate structured knowledge but these methods tend to reduce generalization and learning capabilities of these nets.

Given these complementary features, it seems interesting to combine SM and CM into hybrid symbolic-connectionist systems (HS). Such models are expected to exhibit the advantages of both the symbolic and connectionist ones, which would allow them to perform deductive operations, as well as recognition and learning operations. The building of a hybrid model is not trivial, given the gap between CM and SM, namely knowledge representation. Whereas, in a neural net, knowledge is distributed and cannot be localized on one neuron or another, SM manipulate symbols, which are well-known and well-localized entities. Neural net knowledge is low level, grounded in the real world by learning, whereas SM knowledge is a high level set of symbols and rules (a formal system) with an outside interpretation.

A variety of hybrid systems can be designed, as many parameters can change: knowledge representation type (or types), nature and size of symbolic and connectionist units, nature and size of communications between these units, etc. In fact, the current hybrid systems work divides in two main issues: weakly coupled *versus* tightly coupled HS.

- Weakly coupled HS associate one CM and one SM with each one keeping its identity and functionality. A control mechanism supervises both models. The underlying idea is to use the neural net as a backup when the expert system cannot solve the problem (because of a noisy input or a lack of rules.) This sort of HS tends to provide powerful expert systems with new features: the ability to learn from examples, the ability to increase their knowledge and performance with experience and the ability to deal with noisy or contradictory inputs. [3] and [12] describe such systems.

- The idea of a tightly coupled HS is to find a new reasoning model, by strongly associating symbols and connectionism. The research is in its infancy, but some models are starting to give interesting results. [15] designed a HS to handle common sense and similarity reasoning. Such hybrid systems are usually designed as networks with various types of nodes and communication between these nodes.

Hybrid systems are by essence constituted of several computing elements of different natures and different sizes. It is then possible and interesting to run such a system over a parallel architecture to increase the speed of the process. This is the aim of the Cellular Abstract Machine presented bellow.

3 The Cellular Abstract Machine (CAM)

The Cellular Abstract Machine (CAM) is intended to provide a generic development environment for exploring hybrid system implementation, especially for tightly coupled HS. One of the main features of the CAM is to enable users to execute their programs automatically and transparently on different types of hardware, namely sequential computers and multi-processor machines.

3.1 Computational model

The underlying computational model of the CAM is an agent based model, i.e., is part of the family of actor and object based models. The basic agents are called cells. The main features of the computational model are:

- The communication graph between cells is explicit. This means that a connection between two agents must be declared explicitly and lasts until explicitly destroyed. This is a first difference compared to classical actor models [1] in which messages are sent using logical addresses, thus providing abstraction for the communication pattern.
- The users declare cell types statically before run time. Thus cell types may be compiled in a simple way.
- The cells of a CAM program are dynamically created instances of the static pre-declared types. The pattern of connections between agents may also be created and modified dynamically.
- Finally, the main specificity of the CAM model is that it is globally synchronized. Every cell updates synchronously in a cyclic way, making the CAM model a synchronous MIMD model. This is probably the main difference with classical actor models of computation [1] where concurrency is asynchronous.

3.2 Components of cell type declarations

A cell may be seen as a basic abstract processor, with internal data storage, explicit connections to other cells, and an associated executable program responsible for periodically updating the output values of the connections to other cells.

In the current programming language of the CAM, a program consists of a list of cell type declarations of which the actual cells of a program will be instances. These declarations include for each cell type the following elements:

- The cell type data structures, including the private variables names and those of the input and output channels to connect to other cells. The same output channel may be connected to an unlimited number of input channels. Thus the communication pattern may be more general than just point to point connections.
- To each cell type is also associated a set of transition rules specifying the actions to perform during each update. A transition rule consists of a precondition (head of the rule) and an action (body of the rule). A precondition is a boolean expression testing the value of input channels and private variables. An action is a small sequential program. In the current programming language of the CAM, this program is expressed in a classical procedural style.

A cell update consists of a matching for all transition rules and a firing of at most one rule body. The corresponding action is described with a simple set of instructions in a procedural style.

3.3 Basic CAM cycle

Figure 1 gives the three successive phases of a CAM cycle. The first phase is a computation phase, during which the outputs of the cells are updated according to the former values of their inputs and private variables. During this phase cells may also issue requests for creating new cells thus establishing and modifying connections between cells. On creation each cell receives a unique identifier. This identifier serves as a logical name for the cell and will be used for further connection requests. The second phase is a management phase. During this phase the requests for dynamic changes in the cellular topology are taken into account. During the last phase (the communication phase) the newly computed output of the communication channels between cells are actually sent to the receiving cells.

When a CAM program runs only one cell of a type called *main* is initially created. It is up to the user to define this cell type. The initial cell will be the ancestor of every other cell of the program.

3.4 Example of programming style for the CAM

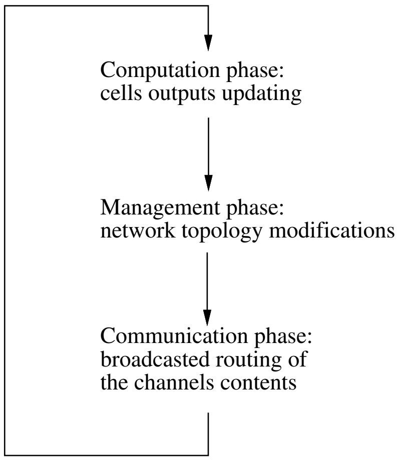

Figure 1: Main stages of a CAM cycle

As an example of the CAM programming style we will briefly explain how to create a feed-forward layered network of cells. This type of structure is very common in neural network applications.

On figure 2 dotted links represent the relations of filiation between cells and plain links correspond to the actual communication channels between cells. The basic idea in the program (fig. 3) is that the first created cell (labeled M on the figure) will create layer cells (labeled either L for the ordinary intermediate layers or I and F for the special case of initial and final layers) and connect them in the feedback direction (see figure 2.1). Each L cell (resp I, F) will first create during its first cycle (the second CAM cycle) the unit cells constituting the corresponding layer. Unit cells are labeled U for ordinary layers and I/O for user interface cell types (see figure 2.2). Then each layer cell will be responsible for connecting its unit cells to the unit cells of the next layer. To enable the layer cells to perform these connections, the identifiers of the unit cells in the next layers are sent back through the connections between layer cells. Figure 2.3 shows the network state after this connection phase.

The *initially* rule corresponds to the creation of the layer units during the first cycle of the L cell. Rule 1 then fires several times until all the units of the next layer are connected to the current layer and until all identifiers of the units of the current layer have been sent to the preceding layer. Eventually, after fulfilling all connection tasks, rule 2 fires in to eliminate the now useless L cell instance.

In the next section we give an example of a hybrid system implementation on the CAM.

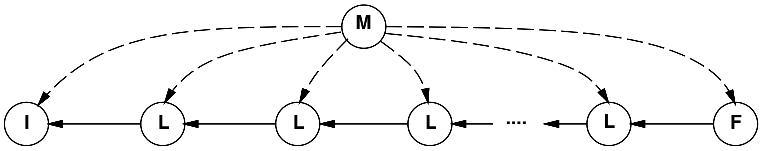

Fig. 2.1: Network state after CAM cycle 1

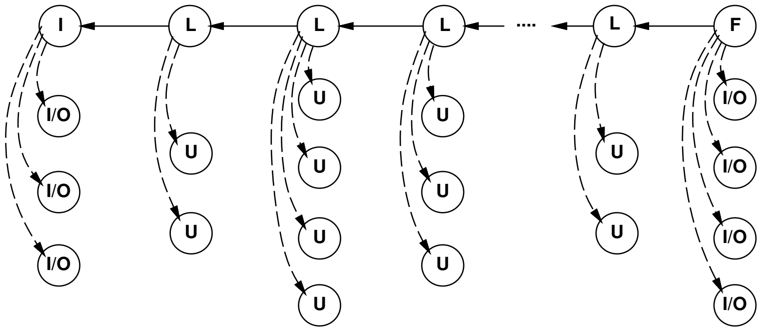

Fig. 2.2: Network state after CAM cycle 2

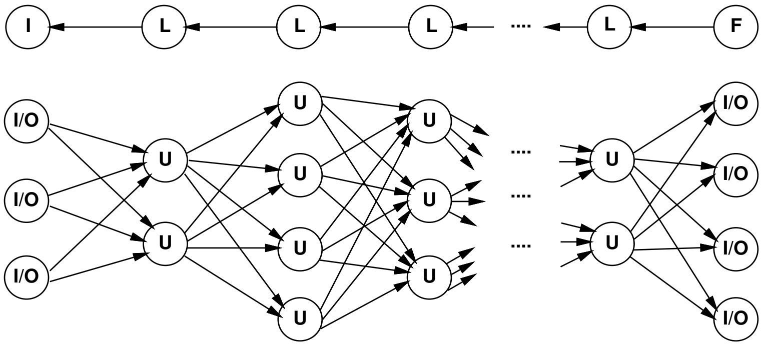

Fig. 2.3: Network state after all layer connections

Figure 2: Principle for creating a feed-forward layered structure

```
typecell L_type (int l, int n) {
/* - l is the layer number.                                          */
/* - n is number of units in the current layer.                      */
 in immat from_following_layer;/* Channel receiving unit cell identifiers */
                              /* from following layer.                */
 out immat to_prev_layer;     /* Channel sending unit cell identifiers of*/
                              /* the current layer to the precedent layer*/
 immat neuron[DIM_LAYER_MAX]; /* Unit cells of the current layer       */
                              /* identifiers table.                   */
 int i,j,k;
/* End of channels and variables declaration.                        */

/* Rule 'initially' is a special precondition automatically fired during */
/* the first cycle of the cell life.                                  */
 initially ==> {
   for (i = 0; i < n; i += 1)         /* Creates the n unit cells of the  */
     neuron[i] = create U_type (l,i); /* layer and stores their logical   */
                                      /* identifiers in the array 'neuron'*/
   to_prev_layer = neuron[0];         /* Sends the identifier of the first*/
                                      /* unit cell to the previous layer. */
   j = 1;                             /* Initialises the counter for      */
                                      /* identifiers sent.                */
 }

/* Rule 1 : fires until all identifiers of unit cells have been sent to  */
/* the preceding layer and until all cells of the following layer have   */
/* been connected to the unit cells.                                   */
 (j <= n) || (from_following_layer != NULL) ==> {
   if (j < n)                         /* If there are still cell identi-  */
     to_prev_layer = neuron[j];       /* fiers to send then send the      */
   else                               /* following cell identifier, else  */
     to_prev_layer = NULL;            /* send a void value.               */
   if (from_following_layer != NULL)  /* If there are still units of the  */
     for (k = 0; k < n; k += 1)       /* following layer to connect to    */
       connect chin[k]                /* the current layer then connect   */
             of from_following_layer/* the following unit of the        */
             oftype U_type            /* following layer to all units of  */
          to chout                    /* the current layer.               */
             of neuron[k]
             oftype U_type;
   j += 1;                            /* Increases counter for identi-    */
                                      /* fiers sent.                      */
 }

/* Rule 2 : fires when rule 1 does not and kills the cell.             */
 (j > n) && (from_following_layer == NULL) ==>
   kill self;
}
```

Figure 3: Declaration of the L cell type in a cellular language implemented on the Cellular Abstract Machine

4 Implementation of a hybrid model on the CAM: an example

Several types of AI programs, connectionist as well as symbolic, have been written using the CAM formalism: traditional connectionist algorithms like the Hopfield model and multi-layered perceptron [6], a small markers propagation network [5] inspired from NETL [8] and a hybrid system called Resyn described in this section (see also [13]).

4.1 The hybrid model

The Resyn network implements two different inference mechanisms, both founded on a localist representation of knowledge (one node for one concept). They are:

- A symbolic marker-passing mechanism to represent symbolic assertions such as *Apple is a Fruit* or *Dolphin is a Mammal*. The nodes in this network (called S-nodes) represent objects like *Fruit* or *Apple* or *Golden*. They are connected by symbolic links such as *is_a* or *is_not_a*.
- A numeric activation network where nodes (called N-nodes) represent features (*Big* or *White*) having a property name (*Size* or *Color*). The links between these nodes are purely numeric and have a weight: for example, there would be a strong link between *big* and *heavy*, meaning *a big object is often a heavy one*. The numeric activations of the features represent their relevance to the current context.

The property names (such as *Size* or *Color*) divide the numeric network in different clusters, used to realize mutual exclusion. When a node on a given cluster receives a sufficient activation, the cluster gets clamped: the first active node will prevent the other ones to become active. For example, when the node *small* in the cluster *size* is active enough, then the other nodes of the cluster (as *big*) cannot become active too.

The connections between nodes of different types are possible: an *elephant* can be connected to *big* and *heavy*, meaning that an elephant has big and heavy for properties, and that a big and heavy thing may be an elephant. The propagation from S-nodes to N-nodes will be done by converting the symbolic markers in a numeric activation. For instance, when *elephant* gets a marker, a numeric activation will be produced on *big* and *heavy*. The propagation will then be able to continue in the numeric part of the network. On the other side, if *big* and *heavy* are sufficiently active (i.e. if their activations are beyond a pre-defined threshold), then *elephant* will tend to get a marker, converted from numeric activations, which will allow propagation to go on in the symbolic part of the network.

The network is described by a set of primitives used to declare new S-nodes or new N-nodes, and to connect them (with S-S, N-N or S-N/N-S connections).

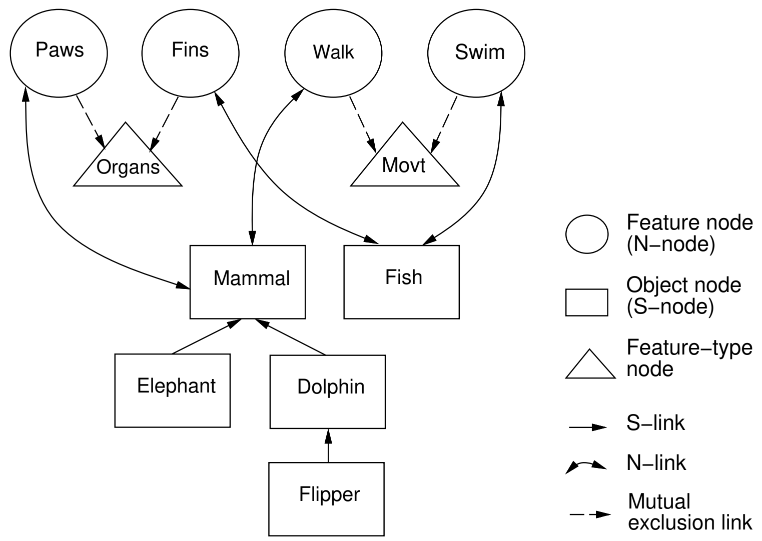

Figure 4: A small Resyn network

These low-level primitives allow to specify directly the architecture of the network. There is also a high-level primitive called *is-like* which allows to declare that two objects are similar even if they do not share the same ancestors. The network can be informed that a dolphin looks like a fish though it is a mammal. The network will then find the features associated to *fish*, and will add new necessary links between *dolphin* and these features (see figures 4 and 5). With the *is_like* primitive, the network topology can be modified without specifying each new node and each new link. The network in figure 5 is built using the following Resyn primitives:

```
new_object(dolphin)              connect(mammal,paws,1)
new_object(mammal)               connect(mammal,walk,1)
new_object(fish)                 connect(fish,fins,1)
new_object(elephant)             connect(fish,swim,1)
new_object(flipper)

new_feature(paws,organs)         is_a(dolphin,mammal)
new_feature(fins,organs)         is_a(flipper,dolphin)
new_feature(walk,movt)           is_a(elephant,mammal)
new_feature(swim,movt)           is_like(dolphin,fish)
```

Due to its hybrid nature, the model can represent an analogy between objects, not unlike [15]. If several objects share enough common features, the

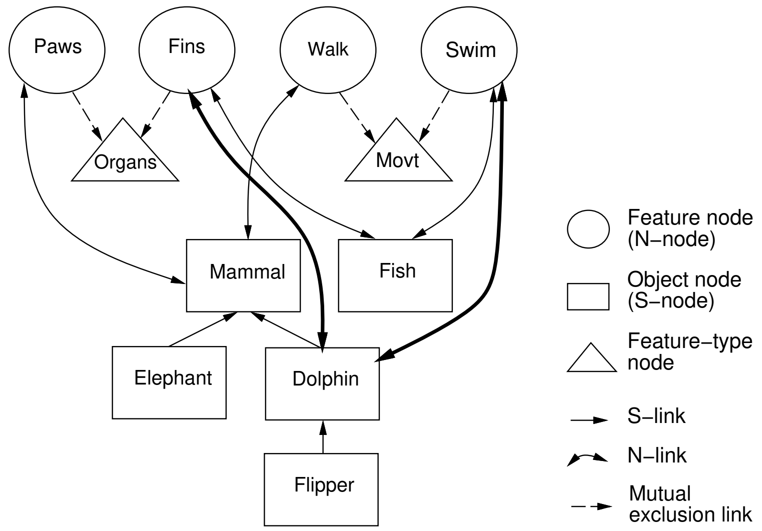

Figure 5: The same network, after execution of *is_like(dolphin,fish)*.

activation of one of them will cause the activation of the other ones. In a similar manner, when some features of an object are active enough, then this object will tend to become active too.

The network is queried by giving some original markers and activations (which can be interpreted as giving clues to the system) to the chosen nodes. Several propagation cycles are accorded to the network to spread activations and markers. The network answer is a list of activated S-nodes (the concepts relevant to the input) and a list of <feature, property name> pairs (the properties of the relevant concepts). For instance, asking about *flipper* (by issuing a symbolic marker on the *flipper* node) in the figure 5 network will give:

flipper is a dolphin
dolphin is a mammal
value of the organ property: fins
value of the movt property: swim

Thanks to the clusters mutual exclusion mechanism, *mammal*'s properties like *paws* will not be activated, as *fins* has been activated first. On the opposite, giving *mammal* a marker and *fins* an activation will make *dolphin* and *flipper* become active, indicating that being a mammal and having fins is sometimes possible, even if not the default case.

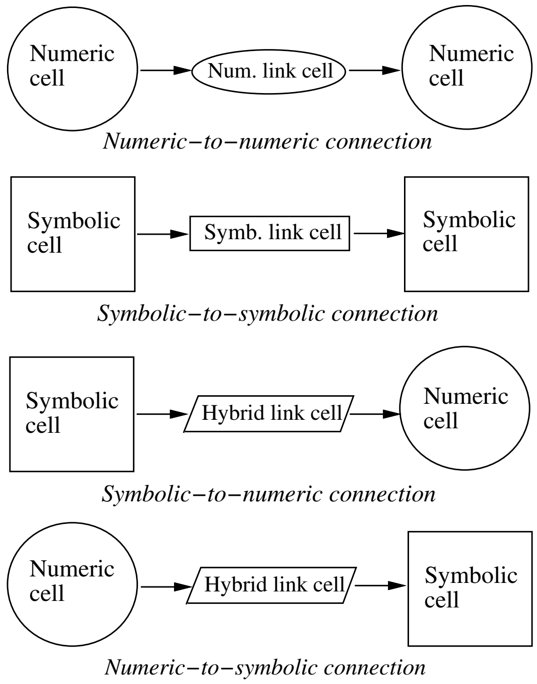

Figure 6: CAM implementation of Resyn: examples of connections between different types of nodes

4.2 Implementation on CAM

The CAM implementation of an heterogeneous network such as Resyn is rather straightforward: each node and each link of the network will be mapped on a cell in the program. One type of cell will correspond to one type of node or one type of link. The families of cell types used in Resyn implementation are the following:

- The object cells: one object (S-node or N-node) corresponds to one cell. The rules allow the computation of the new state of the object (what are its current markers, or what is its activation). The connections (numeric or symbolic links) between the objects are managed by a second class of cells: the link cells.
- The link cells: numeric links are weighted, and compute the activation to transmit from first cell to second cell. Symbolic links are not weighted, and decide which markers they transmit from a first cell to a second cell. Hybrid links decide to convert one type of activation in the other one.

(figure 6 shows examples of different types of cell connections.)

- The cluster cells: there is one cluster for each property name, and one cell for each cluster. The cluster cells are used to implement mutual exclusion: when a property (an N-node) becomes sufficiently active, the corresponding object cell connects itself to its associated cluster cell. The cluster cell can then prevent the other property values from being activated.
- The main cell: it is used to build the network (by creation of new cells and connections between each other) and to query it (by originating markers and activations, and getting the new network state after several propagation cycles).

Thus the CAM is a sufficiently general formalism to describe such tightly coupled hybrid systems, often implemented as dynamic networks of small computing elements. In the next section, we describe the implementation principles of the CAM on a parallel architecture with distributed memory (a Transputer network) which will allow a network like Resyn to run in parallel.

5 Multi-processor implementation of the CAM

The basic implementation principle of the CAM is to map a cell network on a network of processors (see figure 7). During program execution cells are created, connected and communicate with each other. Thus, their placement, connection and communications must be dynamic. Two sorts of data will be routed, using two different routings: the contents of the cell channels (users data), and the cellular network management requests (system data).

In the current version of CAM, a broadcast mechanism allows each processor to communicate to others a part of its memory. A virtual shared memory is thus established, which contains all the channels of each cell. This mechanism allows any cells to communicate together, no matter how far are their host processors from each other (see figure 8). During the communication phase n, each processor is given the contents of all other processors cell channels, and stores those channels with its own ones. Thus, during the computation phase $n+1$, it knows the results of cycle n, and uses them to compute the values of its own output channels. Finally, it broadcasts to all processors the new values of its channels.

A distributed garbage collector recycles the unused parts of the shared memory, to minimize the volume of data exchanged between the processors. This is an important mechanism to get an efficient CAM, as data passing between processors is still the most restricting operation.

Cell placement takes place at the moment of their creation. The target processor for a new cell is chosen by the processor issuing the creation request. For efficiency reasons, there is no cell migration. This would need the update

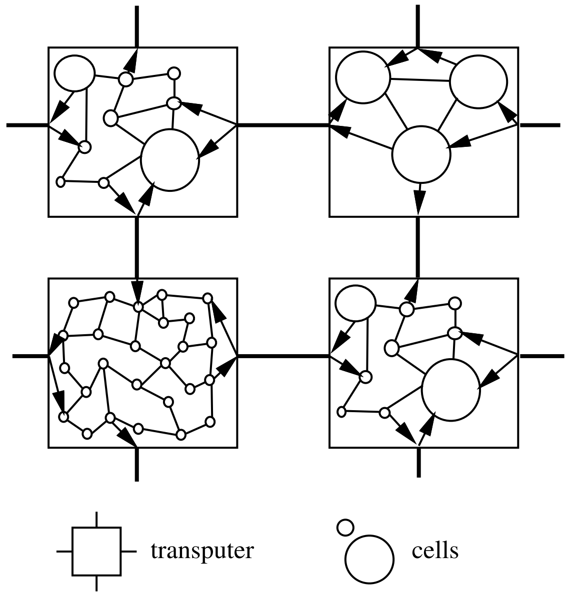

Figure 7: implementation principle on a coarse grain MIMD distributed architecture

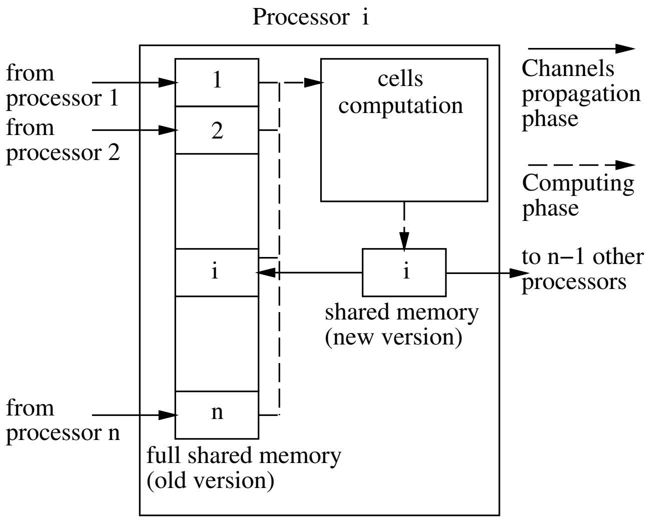

Figure 8: data structure in the local memory of a processor for a fully broadcasted communication

through the network of the cells and their channels, which would not be efficient on our distributed message-passing architecture. The cell mapping only has to balance the processor's load. It is better to tolerate a light load difference using a simple, fast mapping algorithm than to spend time searching for a perfect balance. The estimated load of a processor is based upon its number of cells and its number of output channels. At every cycle, each processor gets these two informations from the others, which allows it to decide which processors are most likely to receive new cells.

Input-outputs are managed by predefined cells, which can be created and connected like ordinary ones, except that those cells are automatically mapped onto the processors driving the *ad hoc* device.

The management mechanism of requests (create, connect, kill...) is robust: it keeps the coherence of the cellular network, even in case of impossible or contradictory requests. For instance, an attempt to connect a dead cell can cause a warning, but does not create a wrong connection, neither does it waste shared memory. Simultaneous requests for the connection of one input channel to different output channels eventually create only one connection.

Such a robustness is achieved by multiple system data routing between the involved processors. The requests execution phase includes three sub-phases, each one including routing and processing of sub-requests. This complex mechanism is necessary to prevent from bad requests and execute good ones.

On each processor a cell manager uses an identifier table to list its cells, and each sub-request is intended for a unique processor cell manager. Thus we use an addressed routing, that avoids sending data to processors which will not worry about them, and minimizes the routed data volume. This is a *store-and-forward* distributed routing [14].

Thus broadcast communication is used for user data whereas addressed routing is used for system data. Anyway, for small messages, routing control informations become bigger than messages themselves. Addressed routing issues more and smaller messages than broadcast, and thus becomes less efficient. Then, for performance reasons, system data can be broadcasted rather than addressed, depending on the data volume to send. An automatic switching between broadcast and addressed modes has been implemented, which minimizes the system data routing delays [9].

6 Advancement of the implementation and conclusion

The results achieved so far by the CAM project are promising:

- A first CAM prototype is running on sequential computer since 1990 [4]. This prototype was associated with a first rough and simple programming language. This sequential version has been used to implement sample programs.

- Several programming examples in different areas of parallel AI are available, demonstrating the interest of the CAM concepts for AI modeling [13].

- A complete theoretical study of expected performances on a Transputer network has been realized ([6], [7]) demonstrating the viability of the implementation method on a multi-processor architecture. Different families of optimal processor topologies have been defined and among them torus grids are currently used.

- The communication kernel (broadcasting, addressing, automatic switching between broadcast and addressed mode, distributed garbage collector, creation, placement, destruction and connections of cells) is running on a Transputer network since September 1992. Experiments have been conducted with this implementation using benchmarks of Transputers realized for this purpose [10]. For an architecture with nine Transputers, the theoretical performance prediction has thus been verified for the communication phase of the implementation. Performance prediction still remains to be validated for a variable size of the architecture.

- A complete operational version on multi-processor architecture is planned for June 1993. This new version includes an enhanced programming language, with technical improvements like the ability to declare and use functions, the ability to write and compile a program in separate files.

Several research directions should be investigated in the future:

- Extensive programming experiments including real size applications and evaluation of the parallel version of the CAM.

- Extensive benchmarking of the multi-processor implementation of the CAM, with a growing number of processors.

- Development of an improved programming language for the CAM with a higher level of data typing and abstraction.

Acknowledgements

We would like to thank H. Schneider and S. Monsallier and R. Potdevin for helpful discussions and comments.

References

[1] G.A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press Series in Artificial Intelligence, Cambridge, Mass., 1986.

[2] V. Ajjanagadde and L. Shastri. Rules and variables in neural nets. *Neural Computation*, (3):121–134, 1991.

[3] B. Amy. Les systèmes hybrides en intelligence artificielle. In *Ecole de Printemps NSI*, Villard de Lans (Isère, France), Mai 1991.

[4] T. Cornu and J.P. Haton. An actor language for connectionism based on cellular automata. In *Proc. INNC90*, Paris, July 1990.

[5] T. Cornu. Un langage d'acteurs pour le connexionnisme à base d'automates cellulaires. In *Workshop Symboliconnexionniste*, Paris, April 1990. ENST.

[6] T. Cornu. *Machine Cellulaire Virtuelle: Définition, Implantation et Exploitation.* PhD thesis, Université de Nancy I and Supélec Metz, 1992.

[7] T. Cornu and S. Vialle. A framework for implementing highly parallel applications on Transputer based architectures. In *Second Workshop on Abstract Machine Models for Parallel Computers.* University of Leeds, April 1993.

[8] S.E. Fahlman. *NETL: a System for Representing and Using Real-World Knowledge.* MIT Press, Cambridge, Mass., 1979.

[9] S. Gay. *Réalisation d'un Système de Communication Multiprocesseurs Optimisé.* DEA thesis, Université de Nancy I and Supélec Metz, France, 1992.

[10] S. Gay and S. Vialle. Analyse des divers parallélismes du Transputer. *Lettre du Transputer et des calculateurs distribués*, (15):27–42, sep 1992.

[11] M. Gutknecht and R. Pfeifer. An approach to integrating expert systems with connectionist networks. *AI Communications*, 3(3):116–127, September 1990.

[12] M. Gutknecht, R. Pfeifer, and M. Stolze. Cooperative hybrid systems. In *IJCAI Proceedings*, pages 824–829, 1991.

[13] Y. Lallement. *Machine Cellulaire Virtuelle et Intelligence Artificielle.* DEA thesis, Université de Nancy I and Supélec Metz, 1991.

[14] L.M. Ni and P.K. McKinley. A survey of wormhole routing techniques in direct networks. *Computer*, 26(2):62–76, February 1993.

[15] R. Sun. *Integrating Rules and Connectionism for Robust Reasonning: A Connectionist Architecture with Dual Representation.* PhD thesis, Brandeis University, Waltham, MA 02254, 1991.

[16] D.S. Touretzki. Boltzcons: Dynamic symbol structures in a connectionist network. *Artificial Intelligence*, 46(1-2), 1990.