A bridge between two paradigms for parallelism : Neural Networks and general purpose MIMD computers.

Yann Boniface, Frédéric Alexandre, Stéphane Vialle

A bridge between two paradigms for parallelism: Neural Networks and general purpose MIMD computers.

Yann Boniface[a] , Frédéric Alexandre[a] and Stéphane Vialle[b]

(a) boniface@loria.fr, falex@loria.fr LORIA, BP 239, 54506 Vandœuvre-lès-Nancy cedex, France
(b) Stephane.Vialle@supelec.fr Supelec, 2, rue Edouard Belin, 57078 Metz, France

I. Abstract

Recent hardware developments have led to use shared memory as an efficient parallel programming way. The main goals of the work reported here are to speed up executions and to decrease development time of parallel neural networks implementations. To allow for such implementations, a library has been defined, as a bridge between neural networks and general purpose MIMD computer parallelisms.

II. Introduction

Artificial Neural Networks are time consuming, especially during the learning phase. This cost together with the weak computation performance of a computer with regard to a human brain [4] make it difficult to test some complex large connectionist models, like some inspired from biological reality or some with a high dimensional input space or a large number of neurons. Another property of neural networks is that they include a large amount of natural parallelism. Consequently, it seems natural to use hardware parallelism technology to implement connectionist models. Parallel implementation can deeply exploit neural networks inherent parallelism (with a distributed set of neurons computing their activity simultaneously) and synchronism (a neural activity is computed from the outputs of all the connected neurons) and thus speed network execution. Furthermore, parallel machines are nowadays more widespread and easier to use and one can thus more and more seriously think to implement artificial neural networks to benefit from their natural parallelism [7].

Numerous attempts have been already done in that direction [6]. Because neural network parallelism is very different from modern and general purpose parallel computer parallelism, it is difficult to map artificial neural network models directly onto a parallel machine (see figure 1). Consequently, most parallel implementations are concerned only by a specific neural model. Beyond dedicated implementations, this paper proposes a general approach which consists of a library for adapting neural network parallelism onto a general purpose computer parallelism.

Making a set of programming tools available for neural networks developers would allow them to efficiently use parallel hardware resources without the need for specific knowledge. These tools must offer sufficient flexibility so that a wide range of neural networks with diverse interconnection structures and computational needs can be efficiently processed [10].

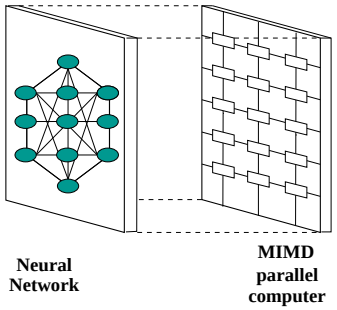

Fig. 1. Mapping a neural network onto a parallel machine

III. Hardware and biological parallelisms

A. Neuronal parallelism

The brain is often described as a massively parallel and distributed machine. It is composed with billions of elementary processors, neurons, who send their activity through unidirectional channels, connections. Artificial neural networks have been built on the same principles, according to the formal neuron model [5], a simple calculus automaton with a set of weighted inputs and an output. Simple distributed activities together with the principle of information widespreading, within a neural network, lead to the characterisation of a fine grain parallelism and a message passing communication paradigm for this kind of structure.

B. Hardware parallelism

Parallel computers are classified into two main classes : SIMD[1] and MIMD[2] machines [1]. SIMD machines use a great number (up to 65536) of specific processors, that execute small computations and a lot of data exchanges. An inspiration from natural neural networks can lead to a mapping of one neuron per processor. Thus SIMD machines seem well adapted to neural computation. Some tools are available, like Cupit [8], but the conception and production of these machines were stopped several years ago for technical and financial reasons.

MIMD parallel machines compound up to several hundreds very powerful processors, such as workstation ones. This kind of computer supports only coarse and medium grain parallel applications. At the origin, they were distributed memory computers and communication between processors used message-passing paradigm.

If this communication protocol is the same as neural network one, these parallel properties are different because, contrarily to these machines, neurons exchange short messages a lot (large connectivity) and often (simple calculus at each time). It is thus hard to get good performances if the application needs frequent communications [10].

To use this parallel hardware technology, connectionist community had to adapt its models. Commonly, they are modified to reduce communication between processors : communication between neurons is reduced and the topology of the networks is changed accordingly.

Therefore, with a distributed memory MIMD computer, a connectionist programmer must know parallel programming and parallel architecture to obtain an efficient implementation. Moreover a specific solution must be developed for each different model of network to parallelise [2].

Since a few years, a new kind of MIMD parallel computer has appeared : Distributed Shared Memory (DSM) computers [9]. Presently, DSM technology goes up to 128 processors. It supports various parallel paradigms, like message passing and memory sharing (see figure 2), and sharing memory is less time consuming than sending messages, when shared memory is well managed [11]. Consequently, it seems possible to communicate between the processors without mapping neural communication protocol to MIMD message passing protocol. Then the main restrictive factor to

[1]Single Instruction Multiple Data
[2]Multiple Instructions Multiple Data

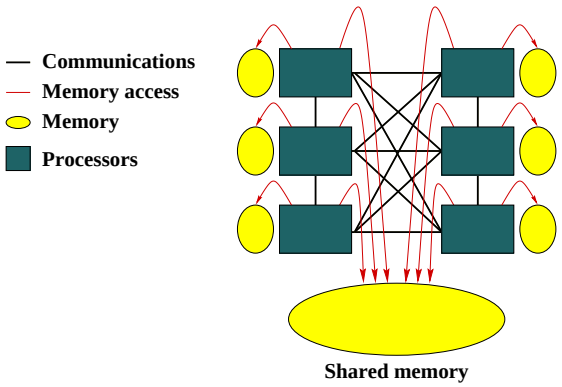

Fig. 2. Shared Memory MIMD parallelism

the building of a general interface between neural networks and that parallel computer seems to be solved.

IV. Our approach

The objective of a parallel implementation is usually to increase the speed of program execution and to deal with larger problems. But this is not our only aim. Specific objectives of our work are linked to the fact that it is intended to provide help to connectionist developers, who are rarely specialists in parallel architectures.

As a consequence, the mapping from fine grain neuronal parallelism to coarse grain architectural parallelism must be transparent for the user. This transparency can be obtained if only a few simple functions are proposed to the user (that is the reason why we chose to develop a library and not a complete new language with its own syntax like ParCel [12]). To increase friendliness, we also require that the same code using our library be compilable and efficient on both sequential and MIMD parallel shared memory machines.

In order to allow for the implementation on a DSM MIMD machine, with coarse grain parallelism and shared memory, of neural networks, rather characterised by fine grain parallelism and message passing communication, we chose to develop a library, as an interface between these paradigms. As "C" is the usual language used by connectionist community, our library is implemented with this language.

Another objective of our work is to facilitate the implementation of biologically inspired neural networks, including such an interesting distributed characteristic as topology. To implement biologically inspired neural networks, developers have to manage many synchronisation problems. As the brain can be seen as asynchronous, each neuron is activated when it receives a signal through its connections. To simulate this behaviour, it is necessary to synchronise all the neurons

and to manage the updating of their outputs.

We propose to develop neural networks with two levels of parallelism. First, the library allows connectionist programmers to develop their model with respect to the fine grain of the biological inspiration, while building the neurons. Second, network building with our library can be executed in general purpose MIMD shared memory computers.

V. Library Overview

Our library allows the building of networks using the smallest grain of the connectionist formalism, the neuron. A neural network is a set of autonomous and synchronous neurons communicating with message-passing through beforehand declared connections. A network is implicitly created while defining a set of neurons and their connections.

A. Neurons

A neuron is an autonomous entity which is activated when it receives a signal from at least one of the neurons to which it is connected, its inputs. Its own activation is sent through its output. This activation is determined from its last activation, the signals it receives from its inputs and the state of its environment.

In the same way, with our library, each neuron has a single output and receives its inputs by connecting itself to the output of any other neuron. One neuron has no limits in terms of number of connections, and each connection is unidirectional . Through these links, for each cycle, it receives the value of the output of the origin neuron evaluated at the end of the previous cycle.

In the simulation, time is divided into cycles. At the end of each cycle, outputs are updated. In the same cycle, a neuron can change its output, and every neurons can read this variable, but the real value will be the value defined at the preceding cycle.

Designed with our tools, a neuron is a function processing its output with respect to its local variables, its input variables and the global variables of the program(cf figure 3). The code of each type of neuron is built like a sequential function. This code describes, for each cycle of the execution, the tasks executed by the neuron. To simplify the implementation, the code that the user has to program is divided into three subfunctions that we call the *characteristic functions* of the neuron :

- **The initialization function** This function refers to the first cycle of the neuron. It contains tasks to be executed at the creation of the neuron : local variables creation and allocation, connections creation and output definition.

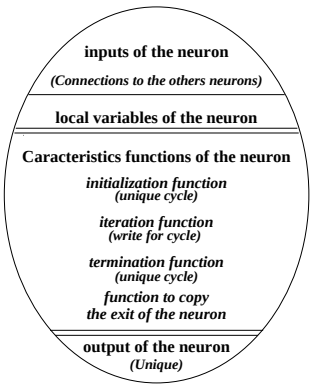

Fig. 3. A model of neuron

- **The iteration function** This function is executed from the second to the penultimate cycle. It usually describes a method to compute the output as a function of its inputs.

- **The termination function** This task is run at the last cycle before the death of the neuron. It is useful for setting local variables free.

If the neuron includes an output, another characteristic function may be declared :

- **The CopyOutput function** This function determines a method to return a complete copy (including its allocation structure) of the output. This function allows to update the neuron output at the end of each cycle. Building this function allows us to use any available type in the "C" language, structured or not.

In terms of variables, each neuron disposes of local variables, global variables and inputs. Each *characteristic function* has a pointer on the local variables. Type of local variable is free. Global variables are the global variables of the program. Each neuron can use them, but our library doesn't manage global variable mutual exclusion. So, if one neuron modifies a global variable during a cycle, two different neurons could read, in this cycle, this variable and obtain two different values. These variables would be only used in reading.

Example of neuron implementation is presented in Section VI..

B. Definition of some library functions

The following functions, provided by the library, are used to initialize and run the network :

- **make_net**(*nb_pc*, *nb_neuron*)
  Allows to declare the number of processes *nb_pc* and neurons *nb_neuron* used to run the network.

- **neuron**(*init_fct*, *iter_fct*, *term_fct*, *registration*)
  Allows to define a neuron. *init_fct iter_fct term_fct* are the characteristic functions described above, *registration* is the neuron identity.

- **execute_net()** This function executes the network composed with neurons defined with the *neuron()* function.

Other functions, provided by the library, allow the programmer to build the code of any type of neuron used in the network :

- **its_output**(*myoutput*, *CopyOutput*)
  This function is used to declare the output of the neuron *myoutput* and the function which copies this output *CopyOutput*.

- **connect_in**(*target*)
  It is used to create a connection between the neuron and the output of the neuron with *target* identity. This function returns a channel.

- **entry**(*channel*) This function returns the content of the input corresponding to the connection *channel*.

- **kill_me()**
  It is used to kill a neuron. When a neuron calls this function, it executes its *termination function* at the following cycle. Then it dies.

These functions are the main ones provided by our library. But other functions are also defined in order to kill neurons (one neuron can kill other neurons) or to manage layer representations of neural networks (e.g building a layer, defining fully connected layers, etc...).

VI. An example of implementation : a Kohonen Map

We present here one example of implementation with our library : the learning phase of a kohonen map [3]. In order to use parallel properties of this kind of neural model, we design our network with two types of neurons. Each neuron represents one prototype. One particular neuron, which is called *master* (neuron 0 in our implementation), has an additional role : for each cycle, it determines the winner and the neighborhood of the winner. Each cycle is divided into two steps. Tables I and II describe the steps of the execution of each neuron. Each neuron is connected to the output

	Initialization function	Iteration function	Termination function
Neurons and Master	Create connections, declare local variable and initialize weights	compute output according to inputs and local variables (two steps, see table II)	free local variable

TABLE I

Work of each type of neuron during the different phases of execution.

	Iteration function	
	Step 1	Step 2
neuron	modifies its weights if it is the winner or in the neighborhood of the winner and computes distance to the current example	no operation (sequential part)
master	same as above plus determination of the next example	Determination of the winner and computation of the neighborhood for the next example

TABLE II

Details of the two steps of the iteration phase for the two types of neuron.

of the master. This output contains the next example to treat, the registration of the last winner and the neighborhood of the winner. The master is connected to all the others, which send through their output their distance from the last example.

Concerning implementation, we now present the most specific steps. First, to define, for each type of neuron, its four characteristic functions (init, iter, term and copy_output), we need to define the various types of the local variable of the neuron (if it is structured). Definition of the type of the master is presented in figure 4.

```
typedef struct {
  int   example;   /*   next example    */
  int    winner;   /*  previous winner  */
  int neighbor;    /*     neighbor      */
  int      iter;   /*    iterations     */
  char     step;   /*     next step     */
}ExitMast;

typedef struct {
  int        last; /*   last example    */
  float   *weight; /* array of weights  */
  canal_in  *link; /*  array of inputs  */
  float     activ; /* previous distance */
  ExitMast   exit; /*      output       */
}LocalMas;
```

Fig. 4. Data type of master neuron

```
ExitMast *CopyMaster(ExitMast *output)
{
  ExitMast *tmp;

  tmp = (ExitMast *) malloc(sizeof(ExitMast));
  *tmp = *output;

  return tmp;
}
```

Fig. 5. Function to copy the output of the master

As master defines one output, we need to write the function to copy it, as figure 5 shows.

The initialization function of each type of neuron is built (see figure 6 for the master). Three important actions are distinguished : (1). The local variables are allocated and defined, (2). The output is defined and (3). Inputs of the master are declared, each link being a connection to one neuron (here link i correspond to neuron i). When each neuron has created its connections, the topology of the network is built.

```
void init_master(void **VarLoc)
{
  int i;
  LocalMas *MesVarLoc;

  /* (1) */
  MesVarLoc=(LocalMas*)malloc(
                sizeof(LocalMas));
  *VarLoc = MesVarLoc;

  MesVarLoc->weight = init_weights(INPUTS);
  MesVarLoc->exit.iter = 0;
  MesVarLoc->exit.step = 1;

  /* (2) */
  its_output(&(MesVarLoc->exit), CopyMaster);

  /* (3) */
  MesVarLoc->link = (canal_in *) malloc(
                   nb_neur*sizeof(canal_in));
  for (i = 0; i < nb_neur; i ++)
    MesVarLoc->link[i] = connect_in(i);
}
```

Fig. 6. Important stage in the initialization function of the master.

Afterwards, the iteration function of each type of neuron is defined. It describes the work of the neuron for each cycle. Figure 7 illustrates several characteristic actions. In (1) the neuron recovers its local variables, according to which, it chooses its action (2). In (3), the master recovers the input corresponding to its link i, for each i, and it uses this input in (4) (neurons just declare a float, their activation, as output). In (5), learning is ending, the master dies.

```
void iter_master(void *VarLoc)
{
  LocalMas  *MesVarLoc;
  SortMait     *Master;
  float       smallest;
  int                i;
  float            *in;

  /* (1) */
  MesVarLoc = VarLoc;

  if (Master->iterations <= AppIter)
    switch (Master->etape) {
  /* (2) */
    case 1 : {
      UpdateWeights();
      ComputeDistanceToExample();
      MesVarLoc->exit.example = my_random();
      MesVarLoc->exit.step = 2;
    } break;
    case 2 : {
      /* looking for the winner */
      smallest  = MesVarLoc->activ;
      MesVarLoc->exit.winner = 0;
      for(i = 1; i < nb_neurones; i++) {
          /* (3) */
          in = entry(MesVarLoc->link[i]);
          /*(4) */
          if(*in < smallest) {
              smallest  = *in;
              MesVarLoc->exit.winner = i;
            } }

      MesVarLoc->exit.neighbor=CompuNeighb();
      MesVarLoc->exit.etape = 1;
      MesVarLoc->exit.iter ++;
 } break;
    }
  else
    /* (5) */
    kill_me();
  }
```

Fig. 7. Important stage for the iteration function of the master.

Finally, the termination functions are defined, like the master's one (figure 8). The main aim of these functions is to make the memory free. In addition, saving weights is also possible.

```
void term_master(void *VarLoc)
{
 LocalMas *MesVarLoc;

 MesVarLoc = VarLoc;
 SaveWeights(MesVarLoc->weight);
 free(MesVarLoc->weight);
 free(MesVarLoc->link);
 free(MesVarLoc);
}
```

Fig. 8. Example of termination function : master

VII. Performances

We presents here the performances of the kohonen map, built like presented above. Figure 9 presents results obtained with a 100x100 grid for 100,000 learning iterations.

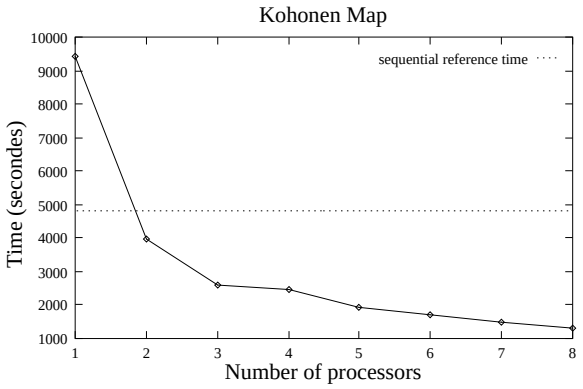

Fig. 9. Computation time as a function of the number of processors

As a reference, we show the sequential runtime of the same kohonen map, implemented in simple "C" language, compiled and executed on the same machine. The cost for using the first version of the library is thus indicated for one processor. Using the library is advantageous beyond one processor. These benchmarks have been executed on an Origin 2000 in multi-users mode.

This result reveals two stages in term of speedup. First a good acceleration is due to the increase of the number of processors and of the cache memory. The size of the available cache memory increases with the number of processors. Then, speedup decreases, due to increasing communications and load balancing difficulties .

This first result is satisfactory and has to be improved. It has been obtained with a relatively good example of network, nearby biological models. Networks with backprogation learning algorithm obtain worse results, due to the sequential (by layer) characteristic of this algorithm.

VIII. Conclusion and perspectives

This library makes a bridge between the natural parallel semantics of neural networks and the parallel architecture of modern and general purpose computers. It facilitates the design of neural networks, and allows their execution onto shared memory MIMD parallel computers. Obtained speedups are interesting for a general purpose tool. Nowadays, we are working to decrease the sequential cost of our library and we are completing its dynamic version.

Acknowledgements. Y. Boniface is supported by a grant from the high performance computing centre Charles Hermite (CCH), which also furnished parallel computation facilities.

References

[1] G. S. Almasi and A. Gottlieb. *Highly parallel computing*. The Benjamin/Cummings publishing company, Inc, 1989.

[2] d'Acierno A. Back-propagation learning algorithm and parallel computers: The clepsydra mapping scheme. *Neurocomputing*, To appear.

[3] T. Kohonen. *Self Organisation and Associative Memory*. Springer Verlag, Berlin, 3rd edition, 1989.

[4] Y. Lallement. Intégration neuro-symbolique et intelligence artificielle, applications et implantation parallèle. *PhD. Thesis*, 1996.

[5] W.S McCulloch and W.P Pitts. A logical calculus in the ideas immanent in nerveous activity. *Bulletin of Matematical Biophysics*, 1943.

[6] M. Misra. Parallel environments for implementing neural network. *Neural Computing Survey*, 1:48–60, 1996.

[7] H. Paugam-Moisy. Multiprocessor simulation of neural networks. In *The Handbook of Brain Theory and Neural Network.*, pages 605–608. The MIT Press., 1995.

[8] L. Precheld. Cupit-a parallel language for neural algorithms : Language reference and tutorial. *Tecnical Report, Univ. Karlsruhe, Allemagne*, 1994.

[9] J. Protić, M. Tomasević, and V. Milutinović. Distributed shared memory: concepts and systems. *IEEE Parallel & Distributed Technology*, 1996.

[10] S. Shams and J.L. Gaudiot. Parallel implementations of neural networks. *International Journal on Artificial Intelligence Tools*, 2(4):557–581, 1993.

[11] Silicon Graphics, Inc. *Performance Tuning Optimization for Origin2000 and Onyx2*. http://techpubs.sgi.com/library/manuals/3000/007-3511-001/html/O2000Tuning.0.html.

[12] S. Vialle, Y. Lallement, and T. Cornu. Design and implementation of a parallel cellular language for mimd architectures. *Computer languages*, 1998.