Membrane Computing in Connex Environment

Gheorghe Ştefan

Politehnica University of Bucharest (Romania) & BrightScale, Inc. (CA)
gstefan@brightscale.com
Web page: http://arh.pub.ro/gstefan/

Summary. The Connex technology is presented as a possible way to implement efficiently membrane computations in Silicon environment. The opportunity is offered by the recent trend of promoting the parallel computation as a real competitor on the consumer market. The Connex environment has an integral parallel architecture, which is introduced and its main performances are presented. Some suggestions are provided about how to use the Connex environment as accelerator for membrane computation.

1 Introduction

The computation model of membrane computing can be supported by a specific physical environment or by non-specific, on-Silicon parallel architectures. The second way is investigated from the view point of the Connex technology: a highly integrated parallel machine.

Membrane computing summary:

The membrane computing model is based on multi-set rewriting rules applied on a membrane structure populated with objects belonging to a finite alphabet. The potential degree of parallelism is very high in P systems because in each step all possible rules are applied (see details in [Păun '02]).

Connex environment summary:

The Connex technology has an intensive integral parallel architecture [Ştefan '06d]. The first embodiment of this technology (see [Ştefan '06c]) targets the high definition TV market, but the chip CA1024 can be used also as a general purpose machine for data intensive computing. Application in graphics, data mining, neural network [Andonie '07], and communication are efficiently supported by the Connex technology. Then, why not for membrane computing!

The Intel study:

Because, since 2002 the clock speed of the processor has improved less than 20%/year, after a long period characterized by around 50%/year, the promise of parallel computing starts to fascinate in a special way. Intel published seminal studies (see [Dubey '05], [Borkar '05]) about the next generation of parallel computers. The future processors will contain multi- or maybe many–processors optimized for the magic triad of **Recognition – Mining – Synthesis** (RMS). The main problem for this promised development is to find the way to program efficiently the next generation of parallel machines. New programming languages or more sophisticated computation models are needed to fructify the opportunities offered by the new coming parallel computation technologies. In this context membrane computing could play a very promising role.

The Berkeley study:

Rather than starting from the market opportunities, as Intel did with the RMS domains, the Berkeley approach [Asanovic '06] starts from their "13 dwarfs" (dense linear algebra, sparse linear algebra, spectral methods, ... finite state machines) identified as **parallel computational patterns** able to cover almost all the applications for the next few decades. While Intel takes into consideration a continuous transition from multi- to many-processors, the Berkeley approach is oriented from the start toward the many-processor systems working on data-intensive computation applications. Here is also the place for membrane computation if a good representation will be developed.

Application oriented vs. functionally oriented parallel architectures:

A complex, intense and general purpose application means usually a multi-threaded approach. In contrast with it, there are functions involving data intense computations. By the rule, multi-processors are involved in the first case (because they are able to exploit thread-level parallelism), and many-processors are needed in the second case. A multi-processor has usually a MIMD architecture, rarely a SIMD architecture, and never a MISD type one. For a many-processor machine the architecture must be shaped starting from a functional approach, and by the rule involves all the special forms of possible parallelism.

Our functional approach:

The **integral parallel architecture** (IPA) is a parallel architecture derived starting from the computational model of *partial recursive functions* [Kleene '36]. The Turing machine model has been successfully used to ground various sequential computing architectures. Because the functional approach of Kleene is more related with circuits (which are intrinsic parallel structures) we consider there is a

better fit between the functional recursive model and the parallel computation. The composition rule provides the best starting point to develop parallel architectures able to support efficiently the other two rules: the primitive recursion and the minimalization. If the *13 dwarfs* will be able to cover the RMS domains, maybe then an IPA will be enough powerful to cover efficiently the 13 computational patterns emphasized by the seminal work done at Berkeley. *A three level hierarchy results. It is topped by application domains (RMS), mediated by computational patterns (the 13 dwarfs), and grounded on various IPAs.*

In the following sections the idea of IPA and the Connex environment are introduced in order to offer various suggestions for a *membrane computing accelerator*. Membrane computing being an intrinsic parallel computational model has the chance to open new ways toward the efficient use of parallel machines.

2 Integral Parallel Architecture (IPA)

Various taxonomies were proposed for parallel computations (see [Flynn '72] [Xavier & Iyengar '98]). All of them tell us about different forms of parallelism. We can discuss about many forms only when we use the parallel approach to accelerate specific computations. But, when a real complex and intensive computation must pe done, sometimes we can not use only one form of parallelism. Actual computations involve usually all possible forms. For example, using Flynn's taxonomy, MIMD or SIMD machines can be defined, but it is not so easy to define MIMD or SIMD application domains.

General purpose or even application domain oriented parallel machines must be able to perform all the forms of parallelism, no matter how these forms are segregated. We propose in the following a new taxonomy and a way to put together all the resulting forms of parallelism in order to solve efficiently data intensive computations.

2.1 Parallelism and Partial Recursiveness

We claim that the most suggestive classic computational model for defining parallel architectures is the model of partial recursive functions, because the rules defining it have direct correspondences in circuits – the intrinsic parallel support for computation.

Composition & basic parallel structures

The first rule, of composition, provides the basic parallel structures to be used in defining all the forms of parallelism. Let be m n-ary functions $h_i(x_0, \dots x_{n-1})$, for $i = 0, 1, \dots m-1$, and an m-ary function $g(y_0, \dots y_{m-1})$. Using them the the composition rule is defined as computing the following function:

$$f(x_0, \dots x_{n-1}) = g(h_0(x_0, \dots x_{n-1}), \dots h_{m-1}(x_0, \dots x_{n-1}))$$

The associated physical structure (containing simple circuits or simple programmable machines) is represented in Figure 1.

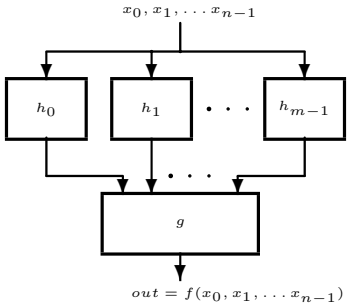

Fig. 1. The physical structure associated to the composition rule. The composition of the function g with the functions $h_0, \ldots, h_{m-1}$ implies a two-level system. The first level, performing in **parallel** m computations, is **serially** connected with the second level which performs a reduction function.

The following four particular, but meaningful forms (see Figure 2) can be emphasized:

1. **data parallel composition**: with $n = m$, each function $h_i = h$ depends on a single input variable x_i, for $i = 0, 1, \ldots n-1$, and g performs the identity function (see Figure 2a). Being given an input **vector** containing n scalars:

$$\mathbf{X} = \{x_0, x_1, \ldots, x_{n-1}\}$$

the result is another vector:

$$\{h(x_0), h(x_1), \ldots, h(x_{n-1})\}$$

2. **speculative composition**: with $n = 1$, i.e. $x_0 = x$, (see Figure 2b), and g performs the identity function. It computes a vector of functions:

$$\mathbf{H} = [h_0(x), \ldots h_{n-1}(x)]$$

on the same **scalar** input x, generating a vector of results:

$$\mathbf{H}(x) = \{h_0(x), h_1(x), \ldots, h_{n-1}(x)\}$$

3. **serial composition**: with $n = m = 1$ (see Figure 2c). A *"pipe"* of two different machines receives a **stream** of n scalars as input:

$$< \mathbf{X} >=< x_0, x_1, \ldots, x_{n-1} >$$

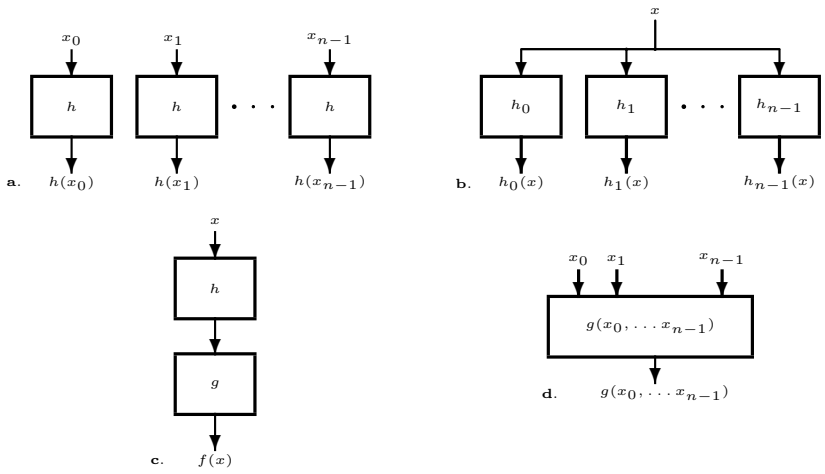

Fig. 2. The four simple forms of composition. a. Data parallel composition. **b.** Speculative composition. **c.** Serial composition. **d.** Reduction composition.

and provides another stream of scalars

$$< f(x_0), f(x_1), \ldots, f(x_{n-1}) > .$$

In the general case the function $f(x)$ is a composition of more than two functions h and g. Thus, the function f can be expressed as a vector of functions $\mathbf{F}$ receiving as input a data stream $< \mathbf{X} >$:

$$\mathbf{F} = [f_0, \ldots f_{p-1}]$$

(in Figure 2c $\mathbf{F} = [h, g]$)

4. **reduction composition**: for each h_i performing the identity function (see Figure 2d), receives a vector $\{x_0, \ldots, x_{n-1}\}$ as input and provides the scalar, $g(x_0, \ldots x_{n-1})$ (it transforms a stream of vectors into a stream of scalars).

Concluding, the composition rule provides the context of defining computation using the following basic concepts:

scalar : x
vector : $\mathbf{X} = \{x_0, x_1, \ldots, x_{n-1}\}$
stream : $< \mathbf{X} >=< x_0, x_1, \ldots, x_{n-1} >$
function : $f(x)$
vector of functions :

- $\mathbf{F} = [f_0, \ldots f_{p-1}]$ applied on streams
- $\mathbf{F}(\mathbf{x}) = [f_0(x), \ldots f_{p-1}(x)]$ applied on scalars.

Using the previous features all the requirement for the next two rules (primitive recursion, minimalization) are fulfilled.

Primitive recursive rule

There are two ways to implement in parallel the primitive recursive rule. In both cases a lot of data is supposed available to be computed, i.e. there are vector or streams of data as inputs for a primitive recursive function.

The primitive recursive rule computes the function $f(x, y)$ using the rule:

$$f(x, y) = h(x, f(x, y - 1))$$

where: $f(x, 0) = g(x)$. This rule can be translated in the following serial composition:

$$f(x, y) = h(x, h(x, h(x, \ldots h(x, g(x)) \ldots)))$$

If the function $f(x, y)$ must be computed for the vector of scalars $\mathbf{X} = \{y_0, y_1, \ldots, y_{n-1}\}$, then a data parallel structure is used. Each machine will compute, using a local *data loop*, the function $f(x, y_i)$ in $max(y_0, y_1, \ldots, y_{n-1})$ "cycles".

If the function $f(x, y)$ must be computed for a stream of scalars, a time parallel structure is used. A "pipe" of n machines will receive in each "cycle" a new scalar from the stream of scalars. If $y > n$, then a *data loop* can be closed from the output of the pipe to its input.

Minimalization

Minimalization has also two kinds of parallel solutions: one using data parallel structures and another using time parallel structures.

The minimalization rule assumes

$$f(x) = min(y)[m(x, y) = 0]$$

i.e., the value of $f(x)$ is the minimum y for which $m(x, y) = 0$.

The first, "brute force" implementation uses the speculative structure represented in Figure 2b, where each block computes a function which returns a pair containing a predicate and a scalar:

$$h_i = \{(m(x, i) = 0), i\}$$

after which reduction step (using a structure belonging to the class represented in Figure 2d) selects the smallest i from all pairs having the form $\{1, i\}$, if any, that were generated on the previous speculative composition level (all pairs of the form $\{0, i\}$ are ignored).

The second implementation occurs in time-parallel environments where speculation can be used to speed-up the pipe processing. **Reconfigurable pipes** can be conceived and implemented using special reduction features distributed along a pipe. Let be a pipe of functions described by the function vector:

$$\mathbf{P} = [f_0(x), \ldots f_{p-1}(x)]$$

where $y_i = f_i(x)$, for $i = 0, \ldots p-1$. The associated reconfigurable pipe means to transform the original pipe characterized by:

$$\mathbf{P} = [\ldots f_i(y_{i-1}), \ldots]$$

into a pipe characterized by:

$$\mathbf{P} = [\ldots f_i(y_{i-1}, \ldots y_{i-s}), \ldots]$$

where: $f_i(y_{i-1}, \ldots y_{i-s})$ is a function or a program which decides in each step the variable to be involved in the current computation, selecting (which is one of the simplest reduction functions) one variable of $\{y_{i-1}, \ldots y_{i-s}\}$. The maximum *degree of speculation* is s.

2.2 Functional Taxonomy of Parallel Computing

According to the previously identified simple form of compositions (see Figure 2) we propose a functional taxonomy of parallel computation. We will consider the following types of parallel processing:

data-parallel computing : uses operators that take vectors as arguments and returns vectors, scalars (by reduction operations) or streams (input values for time-parallel computations); it is very similar to a SIMD machine

time-parallel computing : uses operators that take streams as arguments and returns streams, scalars, or vectors (input values for data-parallel computations): it is a kind of MIMD machine which works to compute only one function (while a true MIMD performs multi-threading)

speculative-parallel computing : with operators that take scalars as arguments and return vectors reduced to scalars using selection (used mainly to speed up time-parallel computations); this contains a true MISD-like structure (completely ignored in the current multi-processing environments).

An IPA is a parallel architecture featured with all kinds of parallelism.

2.3 IPA & Market Tendencies

The market tendencies emphasized in the Intel approach and based on Berkeley's dwarfs demand for an IPA. IPA is a many-core (not multi-core) architecture designed to support data intensive computations. It is supposed to work as an accelerator in a mono- or multi-core environment. For all the computational patterns emphasized in the Berkeley's view an IPA provides efficient solutions. Even for the 13th dwarf – Finite State Machine – the speculative- & time-parallel aspects of an IPA provides a solution. (Berkeley's view claims that "nothing helps".)

The need for solving real hard applications promotes IPA as an efficient actual solution.

3 The Connex System

3.1 Structural description

The first embodiment of an IPA is the Connex System. It is part of CA1024 chip produced by Connex Technology Inc[1]. The Connex System contains mainly an array of 1024 PEs working as a **data parallel sub-system**, DPS, a stream accelerator machine containing 8 PEs (the **time parallel sub-system**, TPS). DPS is driven by an instruction *sequencer*, S, used to broadcast in each clock cycle the same instruction toward each PE from DPS. An *input output controller*, IOC, feeds DPS with data and sends out the results from it. An interconnection fabric allows DPS and TPS to communicate with each other and with the other components of the chip. S and IOC interact using interrupts. They are both simple stack machines with their own data and program memory.

The Connex System uses also other components on the chip to be interfaced with the external world. They are: a MIPS processor acting as a local host, PCI interface to the external host, and a DDR interface to the external memory.

TPS receives streams of data under the control of the local host, and sends the results into the external memory. DPS receives the data vectors from the external memory and sends back the results in the same place. Thus, the two parallel machines communicate usually through the content of the external memory. A data stream is converted into a vector of data, and vice versa, by the programs, run by *Host* and IOC, used to control the buffers organized in the external memory.

3.2 General performances

The first embodiment of the Connex Architecture is designed for $130nm$ standard process technology, and has the following general performances:

- clock frequency: $f_{CK} = 200\ MHz$
- area for the Connex System: $\sim 70\ mm^2$
- 200 $GOPS$ (OP is a 16-bit simple operation; no multiplication, division or floating point)
- $> 60\ GOPS/Watt$
- $> 2\ GOPS/mm^2$
- internal bandwidth: $400\ GB/sec$
- external bandwidth: $3.2\ GB/sec$, involving an additional $2\ Watt$

3.3 Specific performances

The first application domain investigated in the Connex environment is of **High Definition TV** (HDTV). We estimated 80% of the computational power of the Connex System is necessary to decode in real time two H.264 HDTV streams. Some figures referring to specific functions in HDTV domains follow:

[1] From the moment the title of this paper was announced the name of the company changed in **BrightScale Inc.**.

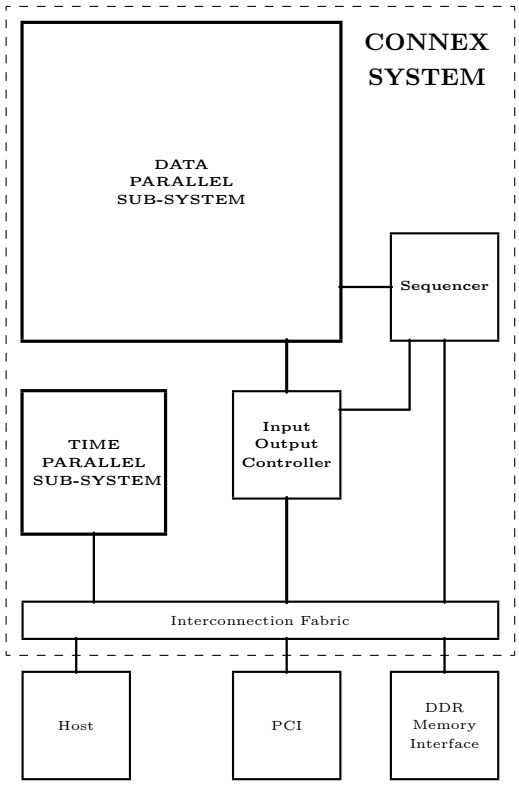

Fig. 3. The Connex System.

- 8×8 DCT: 4.2 *clock_cycle* (0.066 *clock_cycle/pixel*)
- 8×8 IDCT: 4.9 *clock_cycle* (0.077 *clock_cycle/pixel*)
- 4×4 SAD: 0.04 *clock_cycle* (0.0025 *clock_cycle/pixel*)

Graphics is another application domain. A preliminary investigation for an image having the complexity characterized by:

- dynamic images having 10,000 triangles, each covering an average of 100 pixels, one-half being obscured by other triangles
- ambient and diffuze illumination model
- 1920 x 1080 display screen, at 30 frames per second

provides the following figures:

- uses 6.6 $GOPS = 3.3\%$ of the total computational power of the Connex System
- and 390 $MB/sec = 12.2\%$ of the total external bandwidth of the CA1024 chip.

For **linear algebra** domain we present here only the computation of the dot product for vectors of up to 1024 components. Two cases are estimated:

- for vectors having as components 32-bit floats:
  150 $clock\_cycle$ ($> 1.3\ MDot\_Product/sec$)
- for vectors having as components 16-bit signed integer:
  28 $clock\_cycle$ ($\sim 7\ MDot\_Product/sec$)

The **neural network** domain is also targeted as an application domain. A preliminary estimation is done in [Andonie '07]: 5 Giga Connection Updates per Second (about 17 times faster than the fastest *specialized* chip on the market: *Hitachi WSI*).

All these estimations are very encouraging for those who are looking for using the Connex environment as an accelerator for membrane computation.

4 An IPA: The Connex Architecture

The IPA of the Connex System is described in the following two subsections. The vector section describes the architecture of the data parallel sub-system, and the stream section is devoted to describe the time parallel sub-system.

4.1 Vector section

The main physical resources of the Connex System are represented in Figure 4 and are described also in the following pseudo-Verilog form:

```
// Scalar vectors & the index veector
   reg [15:0]  svec_000[0:1023],
               svec_001[0:1023],
               ...
               svec_255[0:1023],
               ixVect[0:1023]  ;
   initial
       ixVect = {0, 1, 2, ... 1023}; // 16-bit scalars
// Boolean vectors
   reg         bvec_0[0:1023]  ,
               bvec_1[0:1023]  ,
               ...
               bvec_7[0:1023]  ,
               selVect[0:1023] ; // it is used only as variable
// Scalars
   reg [31:0]  scalar[0:1023]  ;
// Flag vectors
   wire        cryFlag[0:1023] ,
               zeroFlag[0:1023],
```

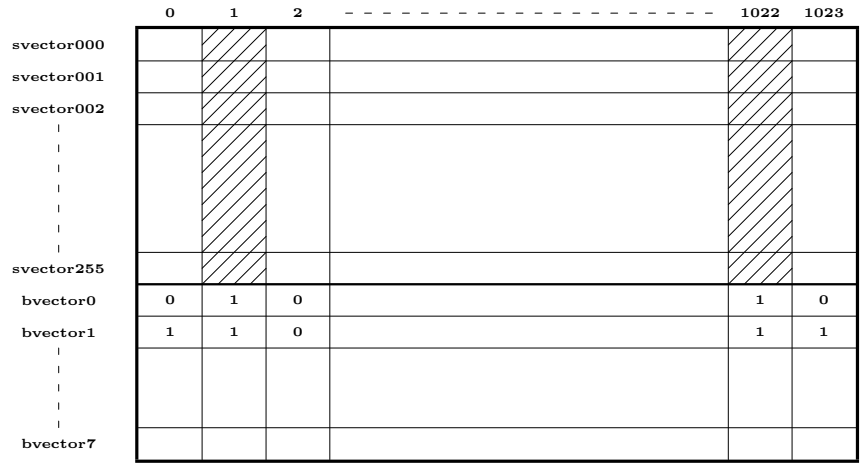

Fig. 4. The vector variables of the data parallel subsystem. If the execution is conditioned by `AND(bvector0, bvector1)`, then only `column1, ... column1022` of scalars can be involved in computation.

```
eqFlag[0:1023]  ,
gtFlag[0:1023]  ,
...             ;
```

The Boolean vectors are used to select the active components of the scalar vectors. The `where` construct is a sort of "spatial if".

```
// 'where' construct
   where BooleanOP(booleanVect_i, booleanVect_j, ...) {
       svect_k = ScalarOP(svect_p, svect_q, ...),
       bvect_r = xxxFlag;
   elsew {
       ...
       }
       }
```

Here is an example of how this construct can be used:

```
   where AND(bvect_2, OR(bvect_0, bvect_5)) {
       svect_034 = ADD(svect_012, svect_078, svect_002),
       bvect_3 = cryVect;
       }
   elsew {
       svect_034 = ADD(svect_022, svect_222),
```

```
    bvect_3 = cryVect;
    }
```

It is executed by the Connex System as follows:

```
selVect = OR(bvect_0, bvect_5);
selVect = AND(bvect_2, selVect);
for(i=0; i<1024; i=i+1)
 if (selVect[i]) {
    svect_034[i] = ADD(svect_012[i], svect_078[i]),
        bvect_3[i] = cryVect[i];
    svect_034[i] = ADD(svect_034[i], svect_002[i]),
        bvect_3[i] = OR(cryVect[i], bvect_3[i]);
    }
 else {svect_034[i] = ADD(svect_022[i], svect_222[i]),
        bvect_3[i] = cryVect[i];
      }
```

There are two distinct ways to generate selections. One is pattern based. It starts from the index vector. Here is an example:

```
/* Pattern based selection example (each other of 4 bit in selVect
will be set on 1)*/
   svect_000 = ixVect;
   svect_000 = AND(svect_000, 16'b11);
   svect_000 = XOR(svect_000, 16'b11), selVect = zeroFlag;
```

The second way to make a selection is to start from the data contained in the scalar vector.

```
// Patternless (data dependent) selection example:
   svect_070 = SUB(svect_070, 16'b10011001), selVect = gtFlag;
```

Usually, any operation specified by one line, having the form:

```
svect_xyz = ScalarOp(...), bvect_q = BooleanOP(...);
```

is executeable in one clock cycle. (Exceptions are specified. For example `MULT(...)` is executed in 9 clock cycles for 16-bit signed integers, and in 10 clock cycles for unsigned integers.)

4.2 Stream section

The stream section of the Connex System receives the input stream $< \mathbf{X} >$ and sends back the output stream $< \mathbf{Z} >$, where:

```
<X> = <x_0, ... x_(p-1)>
<Z> = <z_0, ... z_(q-1)>
```

with $p = q$ or $p \neq q$.

The function of the two-dimension pipe ($n \times w$) is specified by the function vector F, as follows:

```
F = [func_0, ... func_7];
func_i(y_(i-1), y_(i-2), ...  y_(i-w)) = y_i;
```

where: `func_i` is the program executed by PE_i. It could be a one instruction looping program, if the pipe "advances" in each clock cycle, or a s-instruction loop for pipe propagation executed at each s clock cycles. Each PE can have the associated program using variables generated by the previous w PEs. The degree of speculation is w.

Let be, as an example, the following partially defined computation:

```
...
x = ...
y = y[15] ? y + (x + c1) : y + (x + c2);
...
```

Where `c1` and `c2` are constants. The associated function vector is:

```
F = [... func_i(...),
         func_(i+1)(y_i),
         func_(i+2)(y_i),
         func_(i+3)(y_(i-1), y_(i-2)), ...];
```

where:

```
...
y_i = ...;
y_(i+1) = y_i + c1;
y_(i+2) = y_i + c2;
y_(i+3) = y_(i+3)[15] ? y_(i+3) + y_(i+1) : y_(i+3) + y_(i+2);
...
```

The output of the processing element PE_i works as input for both, PE_{i+1} and PE_{i+2}. The processing element PE_{i+3} receives the input variables from the previous two machines PE_{i+1} and PE_{i+2}. The second constant dimension of the pipe allows these "shortcuts" which accelerate the computation.

4.3 Putting together the vector section and the stream section

The two sections of the IPA interact through the content of the external memory. In the external memory a vector or a stream have the same representation. Thus, depending on the source or on the destination, an array of data can be interpreted as a vector or as a stream.

Data exchange between the vector section (DPS) and the stream section (TPS) is done by executing one of the two operation:

```
X <= <Y>; // stream to vector transfer
<X> <= Y; // vector to stream transfer
```

where:

```
X = {x_0, ... x_(n-1)};
<X> = <x_0, ... x_(n-1)>;
Y = {y_0, ... y_(n-1)};
<Y> = <y_0, ... y_(n-1)>;
```

because the destination and the source must have the same dimension n.

5 How to Use Connex to Accelerate Membrane Computing

The key is the representation. The big amount of parallel resources of the Connex architecture can be activated only if an appropriate representation of membrane is adopted. Follow some simple suggestions. The functionality used in these proposals are described in Appendix A.

The first suggestion:

Using the formal definition from [Păun '0x] (see pag. 11), the content of a membrane system can have associated an n-component vector which contains an m-component list (with $n \geq m$). For example (see Fig. 3 in [Păun '0x]):

```
[[[<w_3>]<w_2>]<w_1>]... =
[[[a f c] ] ]...
```

where each symbol is represented by a 2-byte word, `(index, ASCII_code)`, as follows:

```
(1,[) (2,[) (3,[) (0,a) (0,f) (0,c) (3,]) (2,]) (1,]) ...
```

For this first suggestion, only the square parenthesis are indexed, and all the objects are represented with the index having the value 0.

The sets of rules $(R_1, R_2, \ldots)$ are represented inside the program run by the sequencer S. Thus, the list representing the membrane system will evolve as follows:

```
(00) [[[a f c] ] ]... =>
(01) [[[a b f f c] ] ]... =>             // in 11 clock cycles
(02) [[[a b b f f f f c] ] ]... =>       // in 15 clock cycles
(03) [[b b b f f f f f f f c ] ]... => // in 27 clock cycles
(04) [[d d d f f f f c ] ]... =>         // in 10 clock cycles
(05) [[d e d e d e f f c ] ]... =>       // in 10 clock cycles
(06) [d e d e d e d f c ]... =>          // in 10 clock cycles
(07) [d d d d f c ] e e e... =>          // in 15 clock cycles
```

The degree of parallelism is not big enough in each cycle during the previously described computation. Only in step (04) all the three bs were substituted by ds in parallel (in 2 clock cycles). The parallelism is also involved in searching different symbols such as `[, ], a, f`. On the other hand, all the insertions asked by the evolution rules are performed sequentially.

The performance of the implementation can be increased only by changing the representation of the membrane system.

The second suggestion:

Another way to represent the membrane system is to use indexes also for objects. The most significant byte of each vector component is used to tell us how many objects of the kind indicated by the other byte are represented. The same membrane system have now the following content:

```
(1,[) (2,[) (3,[) (1,a) (1,f) (1,c) (3,]) (2,]) (1,]) ...
```

For the same rules applied results the following evolution of the system:

```
[[[1a 1f 1c] ] ]... =>
[[[1a 1b 2f 1c] ] ]... =>   // in 5 clock cycles
[[[1a 2b 4f 1c] ] ]... =>   // in 5 clock cycles
[[3b 8f 1c ] ]... =>        // in 10 clock cycles
[[3d 4f 1c ] ]... =>        // in 7 clock cycles
[[3d 3e 2f 1c ] ]... =>     // in 8 clock cycles
[4d 3e 1f 1c ]... =>        // in 8 clock cycles
[4d 1f 1c ] 3e... =>        // in 5 clock cycles
```

Now applying the rule $f \rightarrow ff$ is executed by simply doubling the index associated to `f`. The same for the rule $d \rightarrow de$. For the rule $ff \rightarrow f$ the index is divided. The main effects are: the representation is kept smaller and the execution time is reduced more than two times.

The degree of parallelism remains small because the application supposes to work only in one membrane at a time. It will be improved if many membranes having the same rules are processed in the same time.

The third suggestion:

The degree of parallelism increases if on the lowest level more similar membranes are defined. Let us make a little more complex the example presented in [Păun 0x] (see Fig. 3). Suppose on the lowest level there are two membranes (`[1a 1f 1c]` and `[2a 1f 1c]`) with the initial content a little different, but working governed by the same rules. Results the following evolution:

```
[[[1a 1f 1c] [2a 1f 1c] ] ]... =>
[[[1a 1b 2f 1c] [2a 2b 2f 1c] ] ]... => // in 5 clock cycles
[[[1a 2b 4f 1c] [2a 4b 4f 1c] ] ]... => // in 5 clock cycles
```

```
[[3b 8f 1c 6b 8f 1c ] ]... =>          // in 10 clock cycles
[[3d 4f 1c 6d 4f 1c ] ]... =>          // in 7 clock cycles
[[3d 3e 2f 1c 6d 6e 2f 1c ] ]... =>    // in 8 clock cycles
[4d 3e 1f 1c 7d 6e 1f 1c]... =>        // in 8 clock cycles
[4d 1f 1c 7d 1f 1c ] 9e... =>          // in 10 clock cycles
```

The execution time has very little increased (only in the last step). It is obvious that having 3 or more low level membranes the degree of parallelism will increase correspondingly.

The performance can be increased more if the rules are integrated in or as a vector representation. In the previous examples the rules were applied sequentially because they were "known" only by the program issued by the sequencer S. The sequencer must know only to apply rules defined inside the Connex Array in an appropriate manner.

6 Concluding Remarks

Functional vs. Flynn's taxonomy

Our functional taxonomy works better in many-processor environment, while Flynn's taxonomy fits better the multi-processor environment. The functional taxonomy supposes three different types equally involved in defining a high performance architecture, while Flynn's taxonomy proposes also three kinds of parallel machines, only one of them being (MIMD) considered as an effective efficient solution for real machines (see [Hennessy '07]).

Limited non-determinism

The physical resources added for the speculative mechanism are used to support a sort of limited non-deterministic computation inside an IPA.

Can we accelerate molecular computing in vector environment?

The vector section of an IPA can be used to accelerate molecular computation if appropriate representations are imagined. Molecular computing has a huge potential for data parallelism and vector processing is a special kind of data parallel computation. The main problem is to reformulate the molecular approach to fit with the restrictions and promises imposed/offered by the vector computation. The Connex System has also some additional features helping the implementation of specific search functions, very helpful for rewriting rule based processing. Various insert and delete capabilities can be used for the same purpose.

An efficient P-Architecture is slightly different from the current Connex Architecture

Although the Connex environment is helpful for investigating molecular computing based applications, there are needed few specific features in order to obtain a market efficient environment.

Why not a P-language?

A very useful intermediary step toward the definition of a marketable environment for this new computation model is providing a P-language. Working with the basic definition of P-systems is not enough flexible for solving real and complex problems. Using a high level type language and developing for it a specific environment will speed-up the work for a specific membrane platform.

Acknowledgments

I would like to thank Emanuele Altieri, Frank Ho, Mihaela Maliţa, Bogdan Mîţu, Tom Thomson, Dominique Thiébaut and Dan Tomescu for their technical contributions in developing the Connex System.

References

[Andonie '07] R. Andonie, M. Maliţa, The Connex Array as a Neural Network Accelerator, accepted at *Third IASTED International Conference on Computational Intelligence, 2007*, Bannf, Alberta, Canada, July2-4, 2007.

[Asanovic '06] K. Asanovic, et al.: *The Landscape of Parallel Computing Research: A View from Berkeley*, *Technical Report No. UCB/EECS-2006-183*, December 18, 2006.

[Borkar '05] S.Y. Borkar, et al.: *Platform 2015: Intel Processor and Platform Evolution for the Next decade*, Intel Corporation, 2005.

[Dubey '05] P. Dubey: *A Platform 2015 Workload Model: Recognition, Mining and Synthesis Moves Computers to the Era of Tera*, Intel Corporation, 2005.

[Flynn '72] M.J. Flynn: Some computer organization and their affectiveness, *IEEE Trans. Comp.* C21:9 (Sept. 1972), pp. 948-960.

[Hennessy '07] J.L. Hennessy, David A Patterson: *Computer Architecture. A Quantitative Approach*, Fourth Edition, Morgan Kaufmann, 2007.

[Kleene '36] S.C. Kleene: General Recursive Functions of Natural Numbers, in *Math. Ann.*, 112, 1936.

[Maliţa '06] M. Maliţa, Gh. Ştefan, M. Stoian: Complex vs. Intensive in Parallel Computation, in *International Multi-Conference on Computing in the Global Information Technology - Challenges for the Next Generation of IT&C - ICCGI 2006*, Bucharest, Romania, August 1-3, 2006.

[Mîţu '05] B. Mîţu: private communication.

[Păun '02] Gh. Păun: *Membrane Computing. An Introduction*, Springer, Berlin, 2002.

[Păun '0x] Gh. Păun: *Introduction to Membrane Computing*, chapter 1 in *Applications of Membrane Computing* (G. Ciobanu, Gh. Păun, M.J.Pérez-Jiménez, eds.), Springer 2006.

[Ştefan '06a] Gh. Ştefan: The CA1024: A Massively Parallel Processor for Cost-Effective HDTV, in *SPRING PROCESSOR FORUM: Power-Efficient Design*, May 15-17, 2006, Doubletree Hotel, San Jose, CA. & in *SPRING PROCESSOR FORUM JAPAN*, June 8-9, 2006, Tokyo.

[Ştefan '06b] Gh. Ştefan, A. Sheel, B. Mîţu, T. Thomson, D. Tomescu: The CA1024: A Fully Programable System-On-Chip for Cost-Effective HDTV Media Processing, in *Hot Chips: A Symposium on High Performance Chips*, Memorial Auditorium, Stanford University, August 20 to 22, 2006.

[Ştefan '06c] Gh. Ştefan: "The CA1024: SoC with Integral Parallel Architecture for HDTV Processin, in *4th International System-on-Chip (SoC) Conference & Exhibit*, November 1 & 2, 2006 - Radisson Hotel Newport Beach, CA.

[Ştefan '06d] Gh. Ştefan: Integral Parallel Computation, in *Proceedings of the Romanian Academy, Series A: Mathematics, Physics, Technical Sciences, Information Science*, vol. 7, no. 3 Sept-Dec 2006.

[Thiébaut '06] D. Thiébaut, Gh. Ştefan, M. Maliţa: DNA search and the Connex technology, in *International Multi-Conference on Computing in the Global Information Technology - Challenges for the Next Generation of IT&C - ICCGI 2006*, Bucharest, Romania, August 1-3, 2006.

[Thiébaut '07] D. Thiébaut, M. Maliţa: Pipelining the Connex array, *BARC07*, Boston, Jan. 2007.

[Xavier & Iyengar '98] C. Xavier, S.S. Iyengar: *Introduction to Parallel Algorithms*, John Wiley & Sons, Inc, 1998.