MapReduce – an Integrative Paradigm in Cyber-Physical Systems

Gheorghe M. Ştefan
Politehnica University of Bucharest
`gstefan@arh.pub.ro`

Abstract

MapReduce is presented as the ubiquitous mechanism used in the coming Cyber-Physical System architectures, from elementary digital circuits to cloud computation. The composition rule, defined in Stephen Kleene's model of computation, comes across all the fundamental mechanisms involved in the informational technologies at any level. The two levels of the Kleene's composition correspond to the two aspects of the proposed integrative paradigm: the *map* level and *reduce* level.

1 Introduction

The ***cyber-physical system*** (CPS) approach goes beyond the ***embedded system*** approach by (1) increasing the *extension*, in networks of active nodes, and (2) by increasing the *intensity* of computation in each node. Both these aspects promote the same type of architectures: the many cell architecture.

In parallel, the exponential IT technology improvements impose the distinction between the ***size*** and the ***complexity*** in the emerging domain of CPS. The relation between size and complexity shapes the architectures defined at any level in a CPS.

We will prove that the Turing-based approach must be substituted by a Kleene-based one in order to support both, the extension and the intensity of functionality within a framework of a good management of a size temperate complexity.

1.1 Size vs. Complexity

Circuit complexity is a branch of computational complexity theory dealing with the size and the depth (or the execution time) of a digital system. Current approach does not make the necessary difference between size and the complexity of a (digital) system. The actual stage of development in system theory requires a clear distinction. Therefore, we propose the following definitions.

Definition 1 *The* **size** *of a system, $S_{system}(n)$, counts the number of components it contains.* $\diamond$

Definition 2 *Algorithmic complexity, or simply* **complexity**, *of a system, $C_{system}(n)$, is proportional with the size of its shortest description* [4] [5]. $\diamond$

In this context rigorous definitions for what means complex or simple are possible.

Definition 3 *A system is said complex if $S_{system}(n) \sim C_{system}(n)$.* $\diamond$

Definition 4 *A system is said simple, or recursive, if $S_{system}(n) >> C_{system}(n)$; usually a simple system is characterized by $S_{system}(n) \in O(f(n))$ while $C_{system}(n) \in O(1)$.* $\diamond$

Meaningful distinctions between *intense* computation and complex *computation* are now possible.

Definition 5 *The complex computation is driven by programs having the static size and the dynamic size of the same magnitude order.* ⋄

Definition 6 *The intense computation is driven by programs having the static size in* $O(1)$ *while the dynamic size in* $O(f(n))$. ⋄

In the previously defined context we are now able to issue the growing principle for the CPS in the nano architectural environment[1]:

> **Nano-growing thesis**: *the complexity of circuits grows slower the the size of the circuit, while the segregation between structures performing intense and complex computation is deepen.*

1.2 Parallelism

The immediate consequence of the nano-growing thesis is that the parallel computation provides the structural basis for developing CPSs. Indeed, in the *Billion-Transistors-Per-Chip Era*, the only way to provide efficiency is to have a *programmable modular* approach. *Modular*, to keep the structural complexity as low as possible. *Programmable*, to led the functional complexity at the desired level.

A modular engine does not mean unconditionally a parallel engine. What are the complementary conditions for obtaining a parallel engine based on a modular structure? An *ad hoc* gathering of computational engines, interconnected according to some "symmetrical" patterns is far from being able to provide a true parallel computing engine. What could be the reason for interconnecting, let us say, 64 sequential processing engine in a 6-dimension hyper-cube? But, if we have done this, we should not be surprised by our inability to use it efficiently! The way from a collection of modules to a parallel computer must follow an appropriate path.

1.3 From computational model to platform

The historical evolution of the sequential processing is a good example to be learned if we intend to build solid fundament for parallel processing. Let us compare the evolution of the sequential paradigm against the evolution of the parallel paradigm of computation.

1.3.1 From Turing's model to *x86*-like platform

The strongly imposed *x86*-like computing platform is the consequence of the following historical evolution:

- **1936 – computational models** : four equivalent models are published (see [20], [6], [12], [14], all reprinted in [8]), out of which the *Turing Machine* offered the most expressive and technologically appropriate suggestion for future developments
- **1944-45 – abstract machine models** : MARK 1 computer, built by IBM for Harvard University, consecrated the term *Harvard abstract model*, while von Neumann's report (see [21]) introduced what we call now the *von Neumann abstract model*; these two concepts backed the *RAM* (random access machine) abstract model used to evaluate algorithms for sequential machines
- **1953 – manufacturing in quantity** : IBM launched *IBM 701*, the first large-scale electronic computer
- **1964 – computer architecture** : in [3] the concept of *computer architecture* (low level machine model) is introduced to allow independent evolution for the two too different aspects of computer design, which have different rate of evolution: software and hardware; thus, there are now on the market few stable and successful architectures, such as x86, ARM, PowerPC.

[1]The nano architeectural environment is defined by silicon technologies under $32nm$. The $32nm$ technology is in the "middle" between *mico* size and *nano* size, at 5 binary magnitude orders from $1\mu m$ and $1nm$.

Thus, in a quarter of century, from 1936 to the early 1960s, the sequential computer domain evolved coherently from theoretical models to mature market products.

1.3.2 From Kleene's model to *MapReduce* platform

Can be retrieved similar milestones in the history of the parallel computing? The historic facts are chained as follows:

- **1962 – manufacturing in quantity** : the first symmetrical MIMD engine is introduced on the computer market by Burroughs
- **1965 – architectural issues** : Edsger W. Dijkstra formulates in [7] the first concerns about specific parallel programming issues
- **1974-76 – abstract machine models** : proposals of the first abstract models (bit vector models in [15], and PRAM models in [10], [11]) start to come in only after almost two decades of non-systematic experiments (started in the late 1950) and too early market production
- **? – computation model** : no one yet considered it, although it is there waiting for us in a "disguised" form.

It is obvious that ***the history of parallel computing is distorted by missing stages and uncorrelated evolutions***. The domain of the parallel computation is unable to provide a stable, efficient and friendly environment for a sustainable market.

The paper proves that for parallel computing, the main computational resource in Cyber-Physical Systems, there is possible to emphasize, based on Stephen Kleene's computing model of partial recursive functions [12], a ubiquitous coherent conceptual framework, from the level of elementary digital circuits to the highest level of cloud computing. It is the ***MapReduce mechanism***.

In the next section the computational model of Kleene is presented as true parallel computing model. The third section takes the MapReduce mechanism, the key concept in parallel computation, and emphasize its presence at any level from the elementary circuits to the cloud computing platforms.

2 Kleene Based Approach

Our proposal for restarting coherently the parallel computation evolution is to substitute the Turing [20] based paradigm with the equivalent Kleene [12] based paradigm.

2.1 Primitive recursion and minimalization are reducible to composition

Theorem 1 *The second rule of the partial recursive model of computation, the primitive recursive rule, is reducible to the repeated application of specific forms of composition rule.*
$\diamond$

Proof: Let be the composition: $C_i(x_1, \ldots, x_i) = g(f_1(x_1, \ldots, x_i), \ldots, f_{i+1}(x_1, \ldots, x_i))$. If g is the identity function $g(y_1, \ldots, y_{i+1}) = \{y_1, \ldots, y_{i+1}\}$ and $f_1(x_1, \ldots, x_i) = h_i(x_1)$, $f_2(x_1, \ldots, x_i) = x_1$, $\ldots$, $f_{i+1}(x_1, \ldots, x_i) = x_i$, then $C_i(x_1, \ldots, x_i) = \{h_i(x_1), x_1, x_2, \ldots, x_i\}$. The repeated application of C_i (see Figure 1a), starting from $i = 1$ with $x_1 = x$ allows us to compute the pipelined function P (see Figure 1b): $P(x) = \{h_1(x), h_2(h_1(x)), h_3(h_2(h_1(x))), \ldots h_k(h_{k-1}(\ldots(h_1(x)\ldots)), \ldots\}$, which is a total function because the functions h_i are total functions. The primitive recursion rule defines $f(x, y)$ using the expression $f(x, y) = g(x, f(x, (y-1)))$ where $f(x, 0) = h(x)$. The iterative evaluation of the function f is done using the following expression:

$$f(x, y) = \underbrace{g(x, g(x, g(x, \ldots g}_{y\,times}(x, h(x))\ldots)))$$

In Figure 2 is represented the iterative version of the structure associated to the primitive recursive rule. The functions used in the iterative evaluation are:

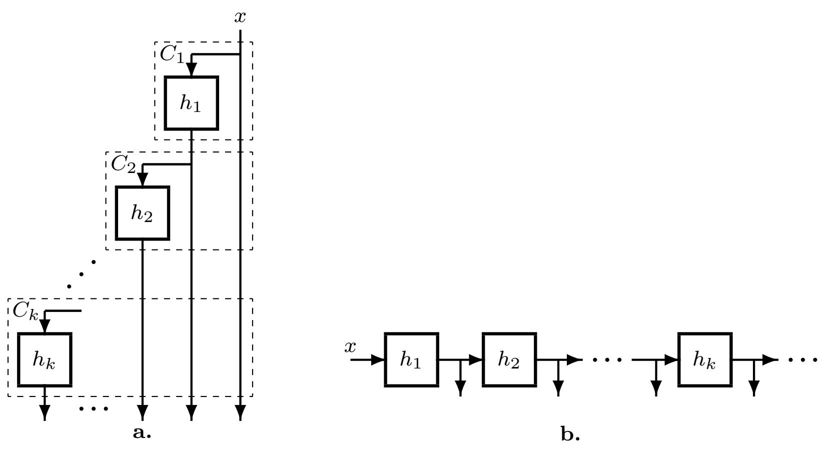

Figure 1: **The pipeline structure as a repeated application of the composition C_i. a. The explicit application of C_i. b. The resulting multi-output pipelined circuit structure P.**

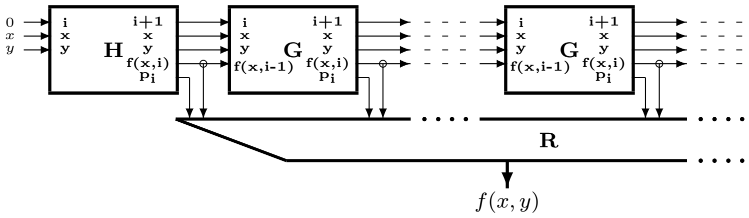

Figure 2: **The circuit which performs the partial recursive computation.**

- $H(i, x, y) = \{(i+1), x, y, f(x, 0), p_i\}$, receives the index $i = 0$ and the two input variables, x and y, and returns: the incremented index, $i + 1$, the two input variables, $f(x, i)$, which is $h(x)$, and the predicate $p_i = p_0 = (y == 0)$. The predicate and the value of the function are used by the reduction function R, while the next function in pipe, G_1, receives $\{(i+1), x, y, f(x, 0)\}$.

- $G(i, x, y, f(x, (i-1))) = \{(i+1), x, y, f(x, i), p_i\}$ receives the index i, the two input variables, x and y, $f(x, (i-1))$, and returns: the incremented index, $i + 1$, the two input variables, $f(x, i)$, and the predicate $p_i = (y == i)$.

- $R(\{\{p_0, f(x, 0)\}, \{p_1, f(x, 1)\}, \ldots, \{p_i, f(x, i)\}, \ldots\}) =$
  $IP(trans(\{\{p_0, f(x, 0)\}, \{p_1, f(x, 1)\}, \ldots \{p_i, f(x, i)\}, \ldots\})) =$
  $IP(\{p_0, p_1, \ldots, p_i, \ldots\}, \{f(x, 0), f(x, 1), \ldots f(x, i), \ldots\}) = f(x, y)$
  is a reduction function; it receives a vector of pairs predicate-value, of form $\{(y == i), f(x, i)\}$, and returns the value whose predicate is `true`. Function R is a composition of two functions: *trans* (transpose), and IP (inner product). Both are simple functions computed by composition.

The two stage computation just described, as an indefinitely extensible to the right structure, is a theoretical model, because the index i takes values no matter how large, similar with the indefinitely extensible ("infinite") tape of Turing's machine. But, it is very important that the algorithmic complexity of the description is in $O(1)$, because the functions H, G and R have constant size descriptions.

$\diamond$

Theorem 2 *The minimalization rule, is also reducible to the repeated application of specific forms of composition rule.*

$\diamond$

Proof: The minimalization rule computes the value of $f(x)$ as the smallest y for which $g(x, y) = 0$. The algorithmic steps used in the evaluation of function $f(x)$ consists of 4 reduction-less compositions and a final reduction composition, as follows:

1. $f_1(x) = \{h_0^1(x), \ldots h_i^1(x), \ldots\} = X_1$, with $h_i^1(x) = \{x, i\}$
2. $f_2(X_1) = \{h_0^2(X_1), \ldots h_i^2(X_1), \ldots\} = X_2$, with $h_i^2(X_1) = \{i, p_i\}$, where $p_i = (g(sel(i, X_1)) == 0)$ is the predicate indicating if $g(x, i) = 0$, and sel is the basic function *selection* in Kleene's definition; it provides pairs index-predicate having the predicate equal with 1 where the function g takes the value 0
3. $f_3(X_2) = \{h_0^3(X_2), \ldots h_i^3(X_2), \ldots\} = X_3$, with $h_i^3(X_2) = \{i, pref_i\}$, where $\{pref_0, \ldots pref_i, \ldots\} = prefixOR(p_0, \ldots p_i, \ldots)$; in [13] is provided a $O(log\ n)$ steps optimal recursive algorithm for computing the prefix function for n inputs
4. $f_4(X_3) = \{h_0^4(X_3), \ldots h_i^4(X_3), \ldots\} = X_4$, with $h_i^4(X_3) = \{i, ADN(pref_i, NOT(pref_{i-1}))\} = \{i, first_i\}$; provides pairs index-predicate where only the first occurrence, if any, of $\{i, 1\}$ is maintained, all the others takes the form $\{i, 0\}$
5. $f_5(X_4) = R(\{\{first_0, 0\}, \ldots, \{first_i, i\}, \ldots\}) =$
$\{OR(\{first_0, \ldots, first_i, \ldots\}), IP(\{first_0, \ldots\ldots, first_i, \ldots\}, \{0, \ldots, i, \ldots\})\} =$
$\{p, f(x)\} = p\ ?\ f(x)\ :\ -$
is a reduction function; it receives a vector of pairs predicate-value, of form $\{(y == i), f(x, i)\}$, and returns the value whose predicate is `true`, **if any**. If $p = 0$, then the function has no value.

The computation just described is also a theoretical model, because the index i has an indefinitely large value. But, the size of algorithmic description remains $O(1)$, because the functions f_j are completely defined by the associated generic functions h_i^j, for $j = 1, 2, 3, 4$.

$\diamond$

2.2 The preeminence of the composition rule

Theorem 3 *The composition rule is the only rule to be considered as defined the Kleene's recursive function computational model.*

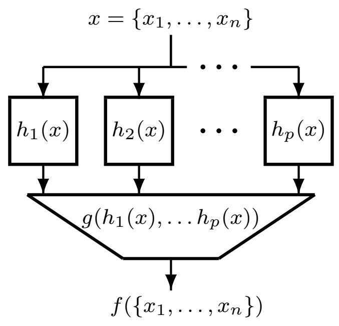
$x = \{x_1, \ldots, x_n\}$
$h_1(x)$
$h_2(x)$
$h_p(x)$
$g(h_1(x), \ldots h_p(x))$
$f(\{x_1, \ldots, x_n\})$

Figure 3: **The circuit structure associated to composition.**

$\diamond$

Proof: According to *Theorem 1* and *Theorem 2* any partial recursive function can be computed using the composition rule and specific compositions.

$\diamond$

The circuit form associated to the composition rule is presented in Figure 3, where the functions $h_i(x)$ are implemented using any type of digital system from combinational circuits to complex computing engines.

2.3 The minimized form of the partial recursive function model

Theorem 4 *Any partial recursive function, according to the Kleene's definition, $f : \mathbf{N}^n \rightarrow \mathbf{N}$, where $\mathbf{N}$ is the set of positive integers, can be computed using the basic functions:*

- ***zero**: $z(x) = 0$*
- ***elementary projection**: $p(i, x_1, x_0) = x_i$, where $x_1, x_0 \in \{0, 1\}$*

*and the **composition** rule:*

$$f(x_1, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), \ldots, h_p(x_1, \ldots, x_n))$$

$\diamond$

Proof: The composition rule allow the expanding of the elementary projection to the Kleene's projection. The elementary increment function is $p(x, 0, 1)$. It can be used for defining the increment function. According to Theorem 1 and Theorem 2 the primitive recursion and the minimalization result as specific compositions.

$\diamond$

2.4 Diachronic & synchronic parallelism in Kleene's model

The Kleene's computational model is an ***intrinsic parallel model of computation***. The only rule defining it – the composition rule – provides two kinds of parallelism: the *synchronic parallelism* on the first stage of $h_i(x)$ functions, and a *diachronic parallelism* between the first stage and the reduction stage.

The theoretical ***degree of parallelism***, δ, emphasized for the two-level function

$$f(x_1, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), \ldots, h_p(x_1, \ldots, x_n))$$

is p for the first level of computation, if $h_i(x_1, \ldots, x_n)$, for $i = 1, \ldots p$, are considered atomic functions, while for the second level δ is given by the actual description of the p-variable function g. The theoretical degree of parallelism depends on the possibility to provide the most detailed description as a composition using atomic functions.

Informally, we ***conjecture*** that *the degree of parallelism for a given function f, δ_f, is the sum of the degree o parallelism found on each level divided by the number of levels.* Therefore, theoretically the function f can be computed in parallel only if $\delta_f > 1$.

It seems that a degree of parallelism $\delta \in O(n/log\, n)$ is the lower limit for what we can call a *reasonable efficient parallel computation.*

3 MapReduce From Circuits to Cloud Supports CPS

In this section is shown how, based on the composition rule, the structural approach at any information technology level is shaped as a MapReduce mechanism.

3.1 Starting with MapReduce in Elementary Circuits

The projection function in the minimized form of Kleene's model is expressed as a composition as follow:

$$p(i, x_1, x_0) = OR(AND(i, x_1), AND((p(i, 0, 1)), x_0))$$

The resulting circuit, the elementary multiplexer (EMUX), is presented in Figure 4a, where the *map* level consists of the two ANDs and the *reduce* level is an OR. The elementary decoder, represented by the NOT circuit, is a composition with the reduction function implemented as the identity function.

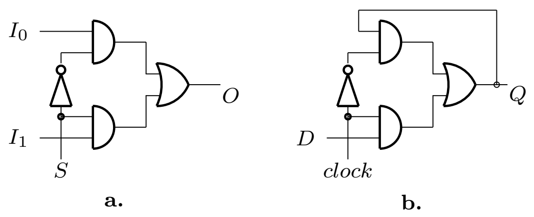

Figure 4: **Elementary multiplexor used as (a) selector and as (b) clocked latch**.

Any combinational circuit results composing EMUXs [19]. More, the elementary storing circuit is obtained closing a *first-order* [19] loop from the output of an EMUX to one of its selected inputs (see Figure 4b.

3.2 Abstract parallel machine model

Once we identified a model of parallel computation, waiting for us from 1936, the next step is to provide an abstract model (similar with the model proposed by von Neumann and the Harvard team).

3.2.1 The 5 abstract forms of parallelism

Definition 7 *The composition rule provide the following 5 abstract forms of parallel computation:*

data-parallel : if $h_i(x_1, \ldots, x_p) = h(x_i)$ and $g(y_1, \ldots, y_p) = \{y_1, \ldots, y_p\}$, then, $f(x_1, \ldots, x_p) = \{h(x_1), \ldots, h(x_p)\}$

reduction-parallel : if $h_i(x_i) = x_i$, then $f(x_1, \ldots, x_p) = g(x_1, \ldots, x_p)$

speculative-parallel : if $g(y_1, \ldots, y_p) = \{y_1, \ldots, y_p\}$, then $f(x) = \{h_1(x), \ldots, h_p(x)\}$

time-parallel : for many applications of $f(x) = g(h(x))$, when $p = 1$, results the pipeline execution $f(x) = f_m(f_{m-1}(\ldots f_1(x)\ldots))$

thread-parallel : if $h_i(x_1, \ldots, x_n) = h_i(x_i)$ and $g(h_1, \ldots, h_p) = \{h_1, \ldots, h_p\}$, then $f(x_1, \ldots, x_p) = \{h_1(x_1), \ldots, h_p(x_p)\}$

$\diamond$

Conjecture: The 5 abstract forms of parallel computation cover efficiently all the aspects of parallel computation.
$\diamond$

The above conjecture is supported (not proved) by actual implementation (for example, ***ConnexArray***$^{\mathbf{TM}}$ [17] [18]) and researches in progress to provide solutions for the 13 *dwarfs* from Berkeley [1].

3.2.2 MapReduce recursive parallel structure

The five forms of parallel computation can be combined in the recursive abstract model for parallel computation presented in Figure 5, where the *elementary engine*, in $cell_{0\_j}$, is a sequential execution or processing unit and the *elementary memory*, in $cell_{0\_j}$, is usually a static RAM of $1 - 16KB$.

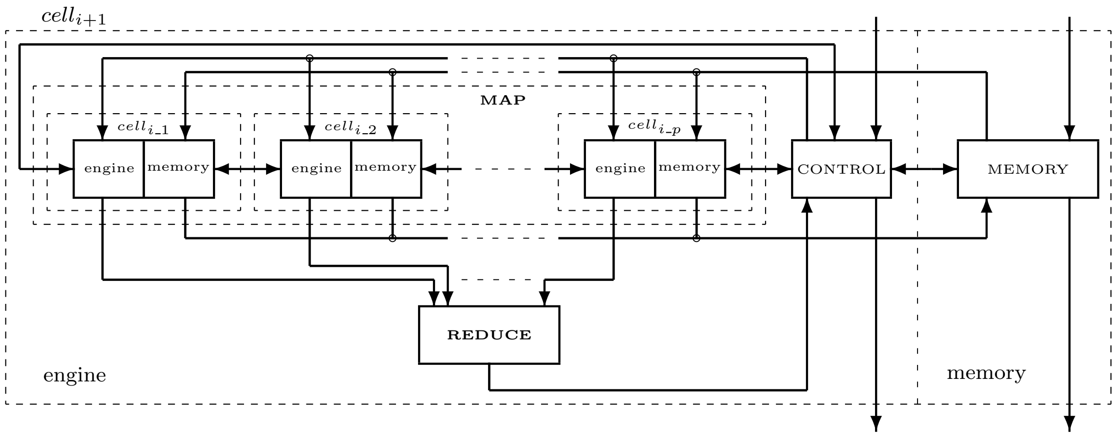

Figure 5: **MapReduce recursive parallel abstract model for parallel computation**

The CONTROL unit is used to issue sequences of parallel functions to be executed in the *MAP* array and/or the *REDUCTION* log-depth network.

The first level of the MapReduce structure the *engine* in $cell_1$, has the optimal implementation as a one-chip solution [17] [18]. Two kinds of operations are mainly defined at this level:

```
x = (x1 x2 ... xp)
y = (y1 y2 ... yp)
z = (z1 z2 ... zp)
(mapOp2 x y) => ((Op2 x1 y1) (Op2 x2 y2) ... (Op2 xp yp))
(mapOp1 x)   => ((Op1 x1) (Op1 x2) ... (Op1 xp))
(reduceOp x) => (x1 Op (x2 Op (... Op xp ...)))
(where (leq x y))
   (reduceOp z)
(endwhere)
```

where: `xi` and `yi`, for `i = 1, 2, ... p`, are scalars; `Op2` and `Op` represent binary operations (`Add, Sub, Mult, And, Or, ...`), while `Op1` represents unary operations (`Inc, Dec, ...`). The `where` operation is used to *select* the results of an `map` operation in order to be applied to a `reduce` operation. Thus, the actual process of ***mapping*** – ***selecting*** – ***reducing*** is supported by the MapReduce recursive parallel abstract model.

3.3 Backus Based Integral Parallel Architecture

Backus's concept of *Functional Programming Systems* (FPS) **can be seen as a *low level description* for the parallel computing paradigm** [2]. In the following we use a FPS-like form to provide a low level functional description for the abstract model defined in the previous subsection. Thus, we obtain the description defining the transparent interface between the hardware system and the software system in a real parallel computer.

3.3.1 Backus's functional forms

A functional form is made of functions that are applied to objects. The following functional forms

- **Apply to all** : $\alpha f : x \equiv (x =< x_1, \ldots, x_p >) \rightarrow < f : x_1, \ldots, f : x_p >$
- **Insert** : $/f : x \equiv ((x =< x_1, \ldots, x_p >) \,\&\, (p \geq 2)) \rightarrow f :< x_1, /f :< x_2, \ldots, x_p >>$
- **Construction** : $[f_1, \ldots, f_n] : x \equiv < f_1 : x, \ldots, f_n : x >$
- **Composition** : $(f_q \circ f_{q-1} \circ \ldots \circ f_1) : x \equiv f_q : (f_{q-1} : (f_{q-2} : (\ldots : (f_1 : x) \ldots)))$
- **Threaded construction** : for $f_i = g_i \circ i$,
  $\theta[f_1, \ldots, f_p] : x \equiv (x =< x_1, \ldots, x_p >) \rightarrow < g_1 : x_1, \ldots, g_p : x_p >$

describe the use of a parallel architecture. `Apply to all` and `Insert` define explicitly a MapReduce mechanism, while the other three are complementary forms used to improve the efficiency of computation.

3.3.2 Kleene-Backus synergy

Between the 5 abstract forms of parallelism and the functional forms, proposed according to the Backs's approach, there is a synergy which provide a good insight related to the above *Conjecture.*

Kleene's parallelism	$\leftrightarrow$	**Backus's functional forms**
data-parallel	$\leftrightarrow$	`apply to all`
reduction-parallel	$\leftrightarrow$	`insert`
speculative-parallel	$\leftrightarrow$	`construction`
time-parallel	$\leftrightarrow$	`composition`
thread-parallel	$\leftrightarrow$	`threaded construction`

This synergy between an abstract model, non-rigorously derived from a mathematical model, and a rigorous programming model support the thesis that a MapReduce based architecture could be used as a successful paradigm for parallel computing in CPS.

3.4 MapReduce Based Cloud Computation

Functional languages, like Lisp/Scheme, promote functional forms. Here are two of the most used functional forms:

```
(define (myMap func list)
 (cond ((null? list) ())
       (#t (cons (func (car list))
                 (myMap func (cdr list))))
))

(define (myReduce binaryOp list)
 (cond ((atom? list) list)
       (#t (binaryOp (car list)
                     (myReduce binaryOp
                               (cdr list))))
))
```

Both, `myMap` funciton and `myReduce` function are hardware supported by our Integral Parallel Architecture, but, in the same time, they are currently used in the intens computational programs. Therefore, it is natural to develop programming models based on *map & reduce.*

The MapReduce programming model, created by Google [9] *"is inspired by functional languages and targets data-intensive computations"* [16]. It helps the development of functional-style code that is automatically parallelized.

MapReduce libraries are written in various programming languages. One of the most popular open-source implementation is Apache Hadoop [22]. It is a parallel software framework for easy writing applications on big amounts of data in *cloud.* A MapReduce program splits the input data-set into independent tasks which are processed by the map tasks in parallel on distinct hardware resources. The the outputs of the maps are *selected* and applied as inputs to the reduce tasks. We retrieve, at this highest level, the low level operations mechanisms described for the MapReduce recursive parallel structure where the `(mapOp2 x y)` or `(mapOp1 x)` operations execute in parallel p binary or unary operations, then the operation `(where cond)` selects from the p results the subset to be submitted to the `(reduceOp z)` operation.

4 Concluding Remarks

The parallel approach in CPS, triggered by the acute need to manage the complexity, is supported at any level by the MapReduce mechanism, as follows:

- circuit level: the elementary multiplexor (which ***maps*** the AND logic function and ***reduces*** by the OR function) is the elementary brick for both combinational and memory circuits
- parallel computational model: the composition rule provide the theoretical environment for both, synchronic and diachronic parallelism
- parallel abstract model: the integral parallel system integrate the 5 forms of parallelism into a recursive parallel structure
- recursive architecture: Backus-based parallel architecture supports the parallel abstract model, and, by turn, it is supported by the recursive parallel structure
- cloud computing: Hadoop MapReduce

From circuits, which are naturally parallel, to cloud, which are inherently parallel, the MapReduce paradigm supports CPS development as an integrative mechanism.

References

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

[2] John Backus: "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs". *Communications of the ACM*, August 1978, 613-641.

[3] Blaauw, G., Brooks, F.P., J.: "The structure of *System 360*, part I - Outline of the logical structure", in *IBM Systems Journal* 3, 2, 1964, pp 119 - 135.

[4] Gregory Chaitin: "On the Difficulty of Computation", in *IEEE Transactions of Information Theory*, ian. 1970.

[5] Gregory Chaitin: "Algorithmic Information Theory", in *IBM J. Res. Develop.*, Iulie, 1977.

[6] Alonzo Church: "An Unsolvable Problem of Elementary Number Theory", in *American Journal of Mathematics*, vol. 58, p. 345-363, 1936.

[7] E. W. Dijkstra: “Co-operating sequential processes” in F. Genuys, editor, *Programming Languages*, pages 43112. Academic Press, New York, 1968. Reprinted from: *Technical Report EWD-123*, Technological University, Eindhoven, the Netherlands, 1965.

[8] Martin Davis: *The Undecidable. Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions*, Dover Publications, Inc., Mineola, New-York, 2004

[9] J. Dean, J. Ghemawat: “MapReduce: Simplified Data Processing on Large Clusters”, *Proceedings of the 6th Symp. on Operating Systems Design and Implementation*, Dec. 2004.

[10] S. Fortune, J. C. Wyllie: “Parallelism in random access machines”, in *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing*, pages 114118, San Diego, CA, May 1978.

[11] L. M. Goldschlage: “A universal interconnection pattern for parallel computers”, *Journal of the ACM*, 29(4):10731086, October 1982.

[12] Stephen C. Kleene: “General Recursive Functions of Natural Numbers”, in *Math. Ann.*, 112, 1936.

[13] R. E. Ladner, M. J. Fischer: “Parallel prefix computation”, *J. ACM*, Oct. 1980.

[14] Emil Post: “Finite Combinatory Processes. Formulation I”, in*The Journal of Symbolic Logic*, vol. 1, p. 103 -105, 1936.

[15] Vaughan R. Pratt, Michael O. Rabin, Larry J. Stockmeyer: “A Characterization of the Power of Vector Machines”, in *Proceedings of STOC’1974*. pp.122 134

[16] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, Christos Kozyrakis: “Evaluating MapReduce for Multi-core and Multiprocessor Systems”, *HPCA 07: Proceedings of the 13th International Symposium on High-Performance Computer Architecture* posted at: `http://csl.stanford.edu/ christos/publications/2007.cmp_mapreduce.hpca.pdf`

[17] Gheorghe M. Ştefan: “One-Chip TeraArchitecture”, in *Proceedings of the 8th Applications and Principles of Information Science Conference*, Okinawa, Japan on 11-12 January 2009.

[18] Gheorghe M. Ştefan: “Integral Parallel Architecture in System-on-Chip Designs”, in *The 6th International Workshop on Unique Chips and Systems*, Atlanta, GA, USA, December 4, 2010, pag. 23-26.

[19] Gheorghe M. Ştefan: *Loops & Complexity in Digital Systems. Lecture Notes on Digital Design in the Giga-Gate per Chip Era*, posted at: `http://arh.pub.ro/gstefan/0-BOOK.pdf`

[20] Alan M. Turing: “On computable Numbers with an Application to the Eintscheidungsproblem”, in *Proc. London Mathematical Society,* 42 (1936), 43 (1937).

[21] John von Neumann: “First Draft of a Report on the EDVAC”, reprinted in *IEEE Annals of the History of Computing,* Vol. 5, No. 4, 1993.

[22] `http://hadoop.apache.org/docs/r1.0.4/mapred_tutorial.html#Purpose`