Can One-Chip Parallel Computing *Be Liberated From* Ad Hoc Solutions? *A* Computation Model Based Approach *and Its* Implementation

Gheorghe M. Ştefan and Mihaela Maliţa

In July 2010 David Patterson said in *IEEE Spectrum* that *"the semiconductor industry threw the equivalent of a Hail Mary pass when it switched from making microprocessors run faster to putting more of them on a chip – doing so without any clear notion of how such devices would in general be programmed"* warning us that one-chip parallel computing seems to be in trouble. Faced with the problems generated by all those *ad hoc* solutions, we propose a fresh restart of parallel computation based on the synergetic interaction between: (1) a parallel computing model (Kleene's model of partial recursive functions), (2) an abstract machine model, (3) an adequate architecture and a friendly programming environment (based on Backus's FP Systems) and (4) a simple and efficient generic structure. This structure is featured with an *Integral Parallel Architecture*, able to perform all the five forms of parallelism (data-, reduction-, speculative-, time- and thread-parallelism) which result from Stephen Kleene's model and is programmed as John Backus dreamed. Our first embodiment of a one-chip parallel generic structure is centered on the cellular engine *ConnexArray*^{TM} which is part of the SoC BA1024 designed for HDTV applications. On real chips we measured $6\,GOPS/mm^2$ and $120\,GOPS/Watt$ peak performance.

***Index Terms*—Parallel computing, recursive functions, parallel architecture, functional programming, integral parallel computation.**

I. Introduction

IT seems that the emergence of parallelism brings difficult times for computer users who lack a friendly environment to develop their applications. But the situation is not new. In 1978 John Backus complained in a similar manner, in [4], telling us that *"programming languages appear to be in trouble"*. In the following fragment he coined the term *"von Neumann bottleneck"*, trying to explain the main limitation of the sequential programming model, dominant in his time:

> *"Von Neumann programming languages use variables to imitate the computer's storage cells; control statements elaborate its jump and test instructions; and assignment statements imitate its fetching, storing, and arithmetic. The assignment statement is the von Neumann bottleneck of programming languages and keeps us thinking in word-at-a-time terms in much the same way the computer's bottleneck does."*

In his seminal paper John Backus proposes two important things: (1) the main limitation of sequential computing, proposing its *PF Systems*, a new programming style, and (2) a formal definition of a generic parallel architecture. Removing the "von Neumann bottleneck" means not only freeing the programming style from parasitic control actions, but also opens the way for triggering parallel actions on large amount of data avoiding explicit cumbersome data and code manipulations.

The history of parallel computing ignored the second suggestion offered by Backus. The parallel computation already begun wrong, with *ad hoc*, speculative constructs, considering that more than one machine, more or less sophisticatedly interconnected, will have the force to solve the continuously increasing hunger for computing power. The scientific community was from the beginning too much focused on parallel hardware and parallel software, instead of solving first the computational model and architectural issues. In fact our computing community started from building too early parallel hardware and then learned quickly that we are not able to program it efficiently. There are few errors in this approach.

First of all, putting together 4, 8 or 128 processors does not mean necessarily that we built a parallel machine. A parallel machine must be thought as a n-cell system, where n is a however large number. Scalability must be the main feature of a parallel engine.

Second, a number of n Turing-based sequential machines, interconnected in a certain network can not offer a starting point in designing a parallel computer, because the role of the interconnections could be more important and complex than the effects of the cells they interconnect.

Third, while the mono-processor computer is theoretically grounded in a *computing model*[1], is based on an *abstract machine model*[2] and is supported by an appropriate *architectural approach*[3], the multi- or many-processor approach is not yet based on an appropriate computational model, there is no a validated abstract machine model or a stabilized architectural environment which refers to a n-sized computational mechanism. Indeed, the sequential, mono-processor computer was backed, by turn, by Turing's (or equivalent) *computing*

Gheorghe M. Ştefan is with the Department of Electronic Devices, Circuits and Architectures, Politehnica University of Bucharest, Bucharest, Romania, e-mail: gstefan@arh.pub.ro

Mihaela Maliţa is with Department of Computer Science, Saint Anselm College, Manchester, NH, e-mail: mmalita@anselm.edu

[1] A ***computing model*** is a *mathematical definition* for automatic computing provided by the theory of computation originated during the 1930s from the seminal ideas triggered by Kurt Gödel in the works of Alonzo Church, Stephen Kleene, Emil Post and Alan Turing.

[2] An ***abstract machine model*** is a *structural definition* which provides the computer organization able to implement the computation defined by a computing model.

[3] An ***architecture*** provides a clear segregation between the hardware and software, using the interface offered by the functionality of an appropriate instruction set.

ISBN: 978-1-61804-237-8

model, by the von Neumann or Harvard *abstract machine model*, and later by an *appropriate architectural approach* when the complexity of hardware-software interaction became embarrassing.

We believe that considering a parallel machine as an *ad hoc* collection of already known sequential machines "appropriately" interconnected is the worst way to start thinking about parallel computation. Maybe, parallel computing is the natural way to make computation and the sequential computation is an early stage of computation we were obliged to accept because of obvious technological limitations, and now is the time to restart the process in the current, improved technological conditions.

A. What is Wrong with Parallel Computing?

There is a big difference between the history of how the sequential computing domain emerged and what happened with the parallel computing domain in the last half century. In the first case there is a coherent sequence of events leading to the current stage of sequential computation, while for parallel computation the history looks like a chaotic flow of events. Let us schematize the emergence of the two sub-domains of computing. First, for sequential computation we have:

- **1936 – computational models** : four equivalent models are published [33] [7] [14] [23] (all reprinted in [9]), 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 [34] 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 [6] the concept of *computer architecture* (low level machine model) is introduced to allow independent evolution for the two 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.

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.

Let's see now what happened in the parallel computing domain:

- **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 [10] the first concerns about specific parallel programming issues
- **1974-76 – abstract machine models** : proposals of the first abstract models (bit vector models in [24], [25], and PRAM models in [11], [12]) start to come in 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 (it is about Kleene's model [14]).

Now, in the second decade of the 3rd millennium, after more than half century of chaotic development, it is obvious that ***the history of parallel computing is distorted by missing stages and uncorrelated evolutions***[4]. The domain of what we call parallel computation is unable to provide a stable, efficient and friendly environment for a sustainable market.

In the history of parallel computation the stage of defining the parallel computational model is skipped, the stage of defining the abstract machine model is too much delayed and confused with the definition of the computation model, while we do not have yet a stable solution for a parallel architecture. Because of this incoherent evolution even the parallel abstract models, by far the most elaborated topics in parallel computation, are characterized by a high degree of artificiality, due to their too speculative character.

B. Parallel Abstract Machine Models

What we call today parallel computation models are in fact a sort of abstract machine models, because true computational models are about how computable functions are defined, not (unrealistic) hardware constructs which interconnect sequential computing engines. Let us take a look in the world of parallel abstract machine models.

1) Parallel Random Access Machine – PRAM

The PRAM abstract model is considered, in [13], a "natural generalization" of the Random Access Machine (RAM) abstract model. It is proposed in [11] and in [12], and consists of n processors and a m-module shared memory, with both, n and m of unbounded size. Each processor has its own local memory. A memory access is executed in one unit time, and all the operations are executed in one unit time. The access type a processor has to the shared memory differentiates four types of PRAMs: EREW (exclusive read, exclusive write), CREW (concurrent read, exclusive write), ERCW (exclusive read, concurrent write), CRCW (concurrent read, concurrent write). The flavor of this taxonomy is too structural, somehow speculative and artificial, unrelated directly with the idea of computation, besides it refers to unrealistic mechanisms. The model is a collection of machines, memories and switches, instead of a functionally oriented mechanism as we have in the Turing Machine, lambda-calculus, recursive functions models. More, as an abstract machine model it is unable to provide:

- accurate predictions about the effective performances related to time, space, energy, because of the unrealistic structural suppositions it takes into account
- a lead to programming languages, because of the fragmented cellular approach which associates a programming language only at the cell level, without any projection to the system level

[4]In this paper the economical, social, psychological aspects are completely ignored, not because they are irrelevant, but because we concentrate only on the pure technical aspects of the problem.

ISBN: 978-1-61804-237-8

- any realistic embodiment suggestion, because it ignores any attempt to provide details about memory hierarchy, interconnection network, communication, ...

Ignoring or treating superficially details about communication and memory hierarchy is deceptive. For example: pure theoretic model for RAM says $n \times n$ matrix multiplication time is in $O(n^3)$, but real experiments (see [26]) provide $O(n^{4.7})$. If for such a simple model and algorithm the effect is so big, then we can imagine the disaster for the PRAM model on really complex problems!

2) Parallel Memory Hierarchy – PHM

The PHM model is also a "generalization", but this time of the Memory Hierarchy model applied to the RAM model. This version of the PRAM model is published in [2]. The computing system is hierarchically organized on few levels and in each node the computation is broken in many independent tasks distributed to the children nodes.

3) Bulk Synchronous Parallel – BSP

The BSP model divides the program in *super-steps* [35]. Each processor executes a *super-step*, which consists of a number of computational steps using data stored in their own local memories. At the end of the super-step processors synchronize data by message passing mechanisms.

4) Latency-overhead-gap-Processors – LogP

The *LogP* model is designed to model the communication cost in a parallel engine [8]. The parameters used to name and to define the model are: *latency* – L – time for a message to move from a processor to another; *overhead* – o – time any processor spends for sending or receiving a message; *gap* – g – is the minimum time between messages; the number of *processors* – P –, each having a big local memory. The first three parameters are measured in clock cycles. The model is able to provide an evaluation which takes into account the communication costs in the system.

The last three models are improved forms of the PRAM model; they provide a more accurate image about parallel computation, but all of them inherit the main limitation of the mother model, the too speculative and artificial PRAM model. The general opinions about PRAM are not very favorable:

> *"Although the PRAM model is a natural parallel extension of the RAM model, it is not obvious that the model is actually reasonable. That is, does the PRAM model correspond, in capability and cost, to a physically implementable device? Is it fair to allow unbounded numbers of processors and memory cells? How reasonable is it to have unbounded size integers in memory cells? Is it sufficient to simply have a unit charge for the basic operations? Is it possible to have unbounded numbers of processors accessing any portion of shared memory for only unit cost? Is synchronous execution of one instruction on each processor in unit time realistic?"* ([13], p. 26)

Maybe even the authors of the previous quotation are somehow wrong, because parallel computation modelled by PRAM is not a natural extension of the sequential computation, on the contrary, we believe that the sequential computation is a special case of parallel computation.

We are obliged to assert that the PRAM model, and its various versions[5], have little, if any, practical significance. In addition, the delayed occurrence of these models, after real improvised parallel machines were already on the market, have a negative impact on the development of the parallel computation domain.

C. What Must be Done?

The question *What must be done?* is answered by *We must restart as we successfully started for sequential computing*. And we have also good news: a lot of stuff we need is there waiting to be used. There is a ***computational model*** which is a perfect match for the first step in the emergence of the parallel computation: the *partial recursive functions* model proposed by Stephen Kleene in 1936, the same year in which Turing, Church and Post made their proposals. It can be used, in a second step, to derive from it an ***abstract machine model*** for parallel computation. For the third step, the *FP Systems*, proposed in 1978 by John Backus, are waiting to be used in order to provide the ***architecture*** or the ***low level model*** for parallel computation.

However, we learned a lot from the work already done for developing abstract models for parallelism. The evaluation made in [17] provides the main characteristics to be considered in the definition of any abstract machine model:

- ***Computational Parallelism*** must be performed using the *simplest* and *smallest* cells; in order to increase the area and energy efficiency, the structural granularity must decrease with the number of cells (see [27])
- ***Execution Synchronization*** can be maintained simple only if the computational granularity is small, preferably minimal
- ***Network Topology*** is suggested by the parallel computational model and must be kept as simple as possible; communication depends on many other aspects and the optimization is a long and complex process which will provide only in time insights about how the network topology must be structured and tuned
- ***Memory Hierarchy*** is a must, but the initial version of the abstract model is preferably to have only a minimal hierarchy; it can be detailed only as a consequence of an intense use in various application domains; the question *caches or buffers?* has not yet an unanimously accepted answer
- ***Communication Bandwidth*** being very costly, in size and energy, must be carefully optimized taking into account all the other six characteristics
- ***Communication Latency*** can be hidden by carefully designed algorithms and by disconnecting, as much as possible, the computation processes by the communication processes

[5]Besides the discussed ones there are many others, like: Asynchronous PRAM, XPRAM (performs periodic synchronization), LPRAM (includes memory latency costs), BPRAM (block transfer oriented), DRAM (adds the level of distributed local memory), PRAM(m) (limits the size of the globally shared memory to m).

- ***Communication Overhead*** has a small impact only for simple communication mechanisms involving simple network, simple synchronization, simple cells, small hierarchy.

Obviously, the complexity can not be avoided eventually, but the way complexity manifests can not be predicted, it must be gradually discovered after a long time use of a simple generic model.

Unfortunately, the emergence of parallel computing occurred in a too dynamic and hurried world, with no time to follow the right path. A new restart is required in order to define a simple generic parallel machine, *subject to organic improvements in a long term evaluation process*. Our proposal is a five-stage approach:

1) use Kleene's partial recursive functions as the ***parallel computational model*** to provide the theoretical framework
2) define the ***abstract machine model*** using meaningful forms derived from Kleene's model
3) put on top of the abstract machine model a ***low level (architectural) model*** description based on Backus's FP Systems
4) provide the simplest ***generic parallel structure*** able to run the functions requested by the low level model
5) ***evaluate*** the options made in the previous three steps in the context of *the computational motifs* highlighted by *Berkeley's View* in [3], looking mainly for systematic or local weaknesses of the architectural model or generic structure in implementing typical algorithms.

The first two steps will be completed in the next two sections, followed by a section used to sketch only the third stage. For the fourth stage an already implemented engine is presented. The fifth step will be only shortly reviewed; it is left for future work. Only after the completion of this 5-stage project the discussion on parallel programming models can be started based on a solid foundation.

II. Kleene's Model is a Parallel Computational Model

Kleene's model of partial recursive functions contains the parallel aspects of computation in its first rule – the ***composition rule*** –, while the next two rules – *primitive recursion* and *minimalization* – are elaborated forms of composition (see [18]). Thus, composition captures directly the process of parallel computing in a number of $p+1$ functions by:

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

where, there are involved, at the first level, p functions – $h_1,\ldots,h_p$ – and a p-variable reduction function g (sometimes implementable as a $(log\, p)$-depth binary tree of $p-1$ functions).

In Figure 1 the ***circuit embodiment*** of the composition rule is represented. It consists of a layer of p cells (each performs the function h_i, for $i = 1,2,\ldots,p$) and a module which performs the reduction function g. In the general case the values of the input variables are sent to all the p cells performing functions h_i and the resulting p-component vector $\{h_1(x_1,\ldots,x_n),\ldots,h_p(x_1,\ldots,x_n)\}$ is reduced to a scalar by the module g.

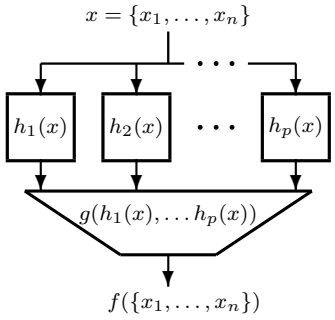

Fig. 1. **The circuit structure associated to composition.**

Two kinds of parallelism are foreseen at this level of our approach: a *synchronic* parallelism – on the first layer of cells – and a *diachronic* (pipeline) parallelism – between the two levels of the structure.

The partial recursive functions model uses two other rules: the *primitive recursive rule* and the *minimalization rule*. We will prove that both can be defined composing special forms of the composition rule. Thus, we will conclude that the composition rule could be the only mechanism to be considered in describing the parallel computation.

A. Reducing Primitive Recursion to Composition

In this subsection is proved that 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. 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 $f_1(x_1,\ldots,x_i) = h_i(x_1)$, $f_2(x_1,\ldots,x_i) = x_1$, ..., $f_{i+1}(x_1,\ldots,x_i) = x_i$, 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 2a), starting from $i = 1$ with $x_1 = x$ allows us to compute the pipelined function P (see Figure 2b):
$P(x) = \{h_1(x), h_2(h_1(x)), h_3(h_2(h_1(x))),\ldots$
$\ldots, h_k(h_{k-1}(\ldots(h_1(x)\ldots)),\ldots\}$

The function $P(x)$ is a total function if the functions h_i are total functions and it is computed using only the repeated application of the composition rule.

The primitive recursion rule defines the function $f(x,y)$ 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:

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

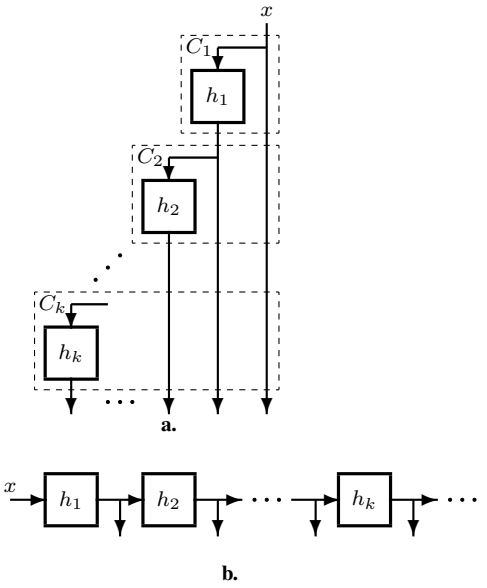

Fig. 2. **The pipeline structure as a repeated application of the composition C_i. a. The explicit application of C_i. b. The resulting multi-output pipelined circuit structure P.**

In Figure 3 is represented the iterative version of the structure associated to the primitive recursive rule. The functions used in the iterative evaluation are:

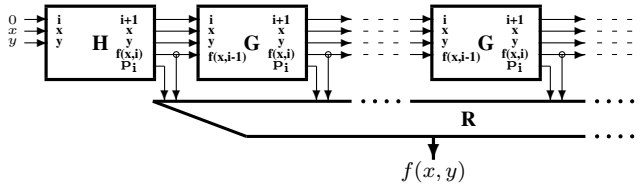

Fig. 3. **The circuit which performs the partial recursive computation.**

- $H(i, x, y) = \{(i+1), x, y, f(x, 0), p_i\}$, 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 $p_i = (y == i)$.
- $R(\{\{p_0, f(x, 0)\}, \{p_1, f(x, 1)\}, \ldots, \{p_i, f(x, i)\}, \ldots\}) = IP(trans(\{\{p_0, f(x, 0)\}, \{p_1, f(x, 1)\}, \ldots \ldots, \{p_i, f(x, i)\}, \ldots\})) = IP(\{p_0, p_1, \ldots, p_i, \ldots\}, \{f(x, 0), f(x, 1), \ldots \ldots, f(x, i), \ldots\}) = f(x, y)$
  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 a structure indefinitely extensible to the right, 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.

B. Reducing Minimalization to Composition

In this subsection is proved that the third rule of the partial recursive model of computation, the minimalization rule, is also reducible to the repeated application of specific forms of the composition rule.

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 $f(x)$ consist 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; 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 [15] 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 take 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 $j = 1, 2, 3, 4$.

C. Concluding about Kleene's Model

In this section we proved that the model of partial recursive functions can be expressed using only the composition rule,

because the other two rules – primitive recursion and minimalization – are reducible to multiple applications of specific compositions. The resulting 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 reduction stage can be expressed in turn using log-stage applications of the composition rule.)

Thus, Kleene's model of parallel computation is described by the circuit construct represented in Figure 1, where p has value no matter how large. For a theoretical model it does not hurt. The *abstract model of parallel computation*, introduced in the next section, aims to remove the theoretical freedom allowed by the "infinite" physical resources.

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 $h_i(x_1, \ldots, x_n)$ are considered atomic functions for $i = 1, \ldots p$, 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 $\delta_f > 1$.

For example, if $f(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i^2$ is the inner product of a vector with itself, then the first level of computation is *data-parallel* with $h_i = x_i^2$ and the second level of computation, the function g, is a *reduction-parallel* function computed by a log-depth binary tree of two-input adders. If *multiply* and *add* are considered atomic operations and n a power of 2, then the value of δ for f is:

$$\delta_f = (n + n/2 + n/4 + \ldots + 1)/(1 + log_2 n) =$$

$$(2n - 1)/(1 + log_2 n) \in O(n/log\, n)$$

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*.

The composition rule will be considered as the starting point, in the next section, for defining an abstract parallel machine model.

III. An Abstract Parallel Machine Model

The distance from Turing's model to Harvard or von Neumann models is the distance between a ***mathematical*** *computational model* and an *abstract* ***machine*** *model*. The first model is mainly about ***What*** *is computation?* and the second is more about ***How*** *computation is done?*. Our abstract machine model takes into consideration some meaningful simplified forms of the composition rule. We claim that the following five forms of composition provide the structural requirements for a *reasonable* efficient parallel engine.

A. Meaningful Simplified Forms of Compositions

1) Data-parallel

The first simplified composition distributes along the functionally identical cells the input sequence of data $\{x_1, \ldots, x_p\}$, and considers that the second level executes the identity function, i.e., $h_i(x_1, \ldots, x_p) = h(x_i)$ and $g(y_1, \ldots, y_p) = \{y_1, \ldots, y_p\}$. Then,

$$f(x_1, \ldots, x_p) = \{h(x_1), \ldots, h(x_p)\}$$

where $x_i = \{x_{i1}, \ldots, x_{im}\}$ are sequences of data, is a ***data-parallel*** computation.

A more complex data-parallel operation is the conditioned (predicated) execution:
$f(\{x_1, \ldots, x_p\}, \{b_1, \ldots, b_p\}) =$
$\{(b_1\ ?\ h_T(x_1) : h_F(x_1)), \ldots, (b_p\ ?\ h_T(x_p) : h_F(x_p))\}$
where: b_i are Boolean variables.

The circuit associated with the data-parallel computation is a cellular structure (see Figure 4a), where each cell receives its own component x_i from the input sequence. The execution is *unconditioned* – each cell executes: $h(x_i)$ –, or it is *conditioned* by the state of the cell, expressed by locally computed Booleans, and each cell executes: $b_i\ ?\ h_T(x_i)\ :\ h_F(x_i)$. Each cell must have local memory for storing the sequence x_i, for the working space and data buffers. The sequence of operations performed by the array of cells is stored in the program memory of the associated control circuit.

2) Reduction-parallel

While the first type of parallelism assumes that the reduction function is the identity function, the second form makes the opposite assumption: the first layer, of the synchronous parallelism, contains the identity functions: $h_i(x_i) = x_i$. Thus the general form becomes:

$$f(x_1, \ldots, x_p) = g(x_1, \ldots, x_p)$$

which *reduces* the input sequence of variables to a single variable (see Figure 4b). The circuit organization of the reduction is tree-like. It consists of a repeated application of various compositions. The size of the associated structure is in the same range as for the data-parallel, while the depth is in $O(log\ p)$.

Because, in the current applications there are only few meaningful reduction functions, the reduction-parallel operations are usually performed using circuits instead of programmable structures.

3) Speculative-parallel

The third simplified composition is somehow complementary to the first: the functionally different cells – h_i – receive the same input variable – x – while the reduction section is the same. Then,

$$f(x) = \{h_1(x), \ldots, h_p(x)\}$$

where: x is a sequence of data. The function returns a sequence of sequences.

There are two ways to differentiate the functions $h_i(x)$:

1) $h_i(x)$: represents a specific sequence of operation for each i. Then, the local memory in each cell contains, besides data, the sequence of operations

ISBN: 978-1-61804-237-8

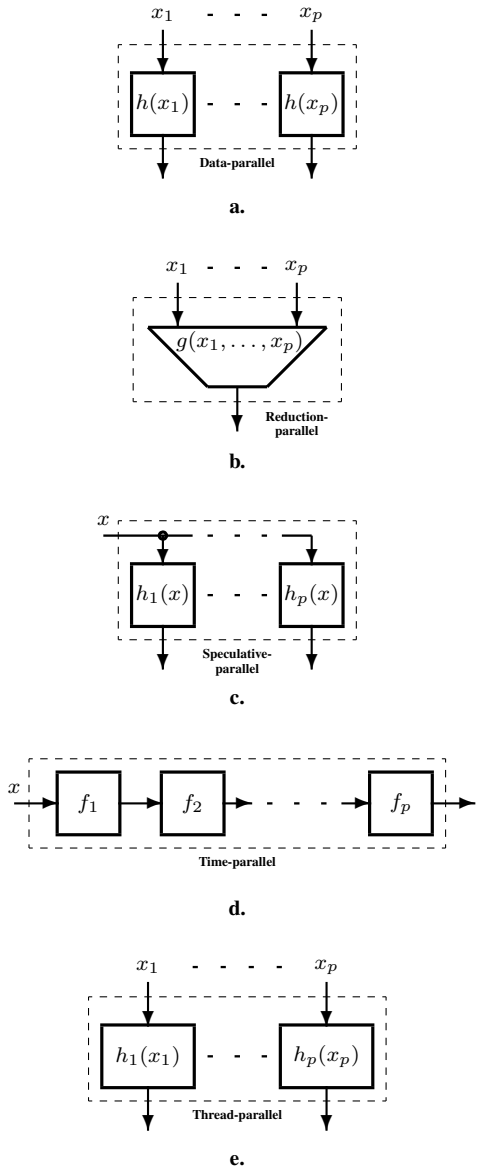

Fig. 4. **Five types of parallelism as particular forms of composition (see Figure 1)**

2) $h_i(x) = g(i, x)$: the sequence of operations are identical for each i, but the function has the parameter i. Then, the local memory contains only data, and the execution takes into account the index of the cell to differentiate the local execution of the sequence of operations stored in the memory of the associated control device.

The circuit associated to the speculative-parallel computation is a cellular structure (see Figure 4c), each cell receiving the same input variable – x – which is used to compute different functions. The general case of speculative-parallel computation requests local data and program memory. While the data-parallel cell is an *execution unit*, the speculative-parallel cell is sometimes a *processing unit*.

4) Time-parallel

There is the special case when the functions are defined for $p = 1$, i.e., $f(x) = g(h(x))$. Then, here is no synchronous parallelism. Only time (diachronic), pipelined parallelism is possible if in each "cycle" a new value is applied to the input. Thus, in each "cycle" the function h is applied to $x(t)$ (which is x at the "moment" t) and g is applied to $h(x(t-1))$ (where $x(t-1)$ is the value applied to the input at the "moment" $t-1$). The system delivers in each "cycle" the result of a computation supposed to be performed in 2 "cycles", or we say that the system works in parallel for computing the function f for 2 successive values.

Many applications of $f(x) = g(h(x))$ result in the m-level *"pipe"* of functions:

$$f(x) = f_m(f_{m-1}(\ldots f_1(x)\ldots))$$

where: x is an element in a stream of data. The resulting structure (see Figure 4d) is a parallel one if in each "cycle" a new value for x is inserted in the pipe, i.e., it is applied to f_1.

This type of parallelism comes with a price: the *latency* time, expressed in number of cycles, between the insertion of the first value and the occurrence of the corresponding result.

5) Thread-parallel

The last simplified form of composition, we consider for our abstract machine model, is the most commonly used in current real products. It is in fact the simplest parallelism, applied when the solution of a function is a sequence of objects computed completely independent. 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 the general form of composition is reduced to:

$$f(x_1, \ldots, x_p) = \{h_1(x_1), \ldots, h_p(x_p)\}$$

where: x_i is an sequence of data. Each $h_i(x_i)$ represents a distinct and independent *thread* of computation performed in distinct and independent cells (see Figure 4e). Each cell has its own data and program memory.

B. Integral Parallelism

We make the assumption that the previously described particular forms of composition – the only rule that we showed is needed for the calculation of any partial recursive function – cover the features requested for a ***parallel abstract machine model***. This assumption remains to be (at least partially) validated in the fifth stage of our proposal, which evaluates the model against all the known computational motifs.

1) Complex vs. Intense in Parallel Computation

The five forms of parallelism previously emphasized perform two distinct types of computation:

- **intense computation** : the algorithmic complexity of the function $f(x_1, \ldots, x_n)$ is constant, while the size of data is in $O(F(n))$
- **complex computation** : the algorithmic complexity of the function is in $O(F(n))$.

Revisiting the types of parallel computation we find that:

- ***data-parallel*** computation is ***intense***, because it is defined by one function, h, on many data, $\{x_1, \ldots, x_p\}$

ISBN: 978-1-61804-237-8

- ***reduction-parallel*** computation is ***intense*** because the function is simple (constant size definition) and the size of data is in $O(p)$
- ***speculative-parallel*** computation ***most of time is intense*** while ***sometimes is complex***, because:
  - when $h_i(x) = g(i,x)$ the resulting computation is intense, because the functional description has constant size, while the data is $\{x, 0, 1, \ldots, p\}$; we call it *intense speculative-parallel* computation
  - when $h_i \neq h_j$ for $i, j \in \{0, 1, \ldots, p\}$ the functional description has the size in $O(p)$ and the size of data is in $O(1)$; we call it *complex speculative-parallel* computation.
- ***time-parallel*** computation is ***most of the time complex***, because the definition of the m functions f_i has the size in $O(m)$, while ***sometimes is intense***, when the pipe of functions is defined using only a constant number of different functions
- ***thread-parallel*** computation is ***complex*** because the size of the description for $h_i \neq h_j$ for $i, j \in \{1, \ldots, p\}$ is in $O(p)$.

2) Many-Core vs. Multi-Core

For a general purpose computational engine all the five forms must be supported by an integrated abstract model. We call the resulting model: ***integral parallel*** *abstract machine model* (see Figure 5).

In Figure 5 there are emphasized two sections, called MANY-CORE and MULTI-CORE. The first section contains the cells $c_1, \ldots c_p$ and the log-depth network **redLoopNet** (which performs reduction functions and closes a global loop responsible for performing *scan* functions). They are used for data-, speculative-, reduction- and time-parallel computation (sometimes for thread-parallel computation). The second section, contains the cells $C_1, \ldots C_q$, mainly used for thread-parallel computation. Each cell c_i is a minimalist implementation of a processing element or of an execution unit, while each C_j can be a strong and complex processing element. One C_i core is used as ***controller*** (C_1 in our representation) for the MANY-CORE array. In real applications the system is optimal for $p >> q$.

The MANY-CORE section is involved mainly in ***intense computations*** (characterized by: many-core, sequence computing, high-latency functional pipe, buffer-based memory hierarchy), while the MULTI-CORE section is mainly responsible for ***complex computations*** (characterized by: mono/multi-core, multi-threaded programming model, cache-based memory hierarchy) (details in [31]).

The main differences between complex and intense computation at this level of description are: (1) $p >> q$ and (2) the access to the system memory through the cache memory for complex computation and through an explicitly controlled (multi-level) buffer for intense computation.

3) Vertical vs. Horizontal Processing

The buffer-based memory hierarchy allows two kinds of intense computation in the MANY-CORE section. Because the first level **Buffer** module stores m p-element sequences which can be seen as a two-dimension array, the computation can

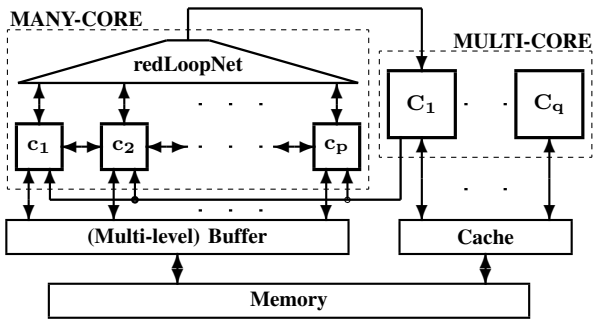

Fig. 5. **The integral parallel abstract machine model.**

be organized ***horizontally*** or ***vertically***. Let us consider the following m sequences stored in the first level **Buffer**.

$$s_1 =< x_{11}, \ldots, x_{1p} >$$
$$s_2 =< x_{21}, \ldots, x_{2p} >$$
$$\ldots$$
$$s_m =< x_{m1}, \ldots, x_{mp} >$$

Horizontal computing means to consider a function defined on the sequence s_i. For example, FFT on s_i which returns as result the sequences s_{i+1} and s_{i+2}. *Vertical computing* means to compute p times the same function on the sequences

$$< x_{1i}, x_{2i}, \ldots x_{ji} >$$

for $i = 1, 2, \ldots p$. For the same example, p FFT computations can be performed on

$$< x_{1i}, x_{2i}, \ldots x_{ji} >$$

for $i = 1, 2, \ldots p$, with results in

$$< x_{(j+1)i}, x_{(j+2)i}, \ldots x_{(2j)i} >$$

$$< x_{(2j+1)i}, x_{(2j+2)i}, \ldots x_{(3j)i} >$$

for $i = 1, 2, \ldots p$. If $p = j$, the same computation is performed on the same amount of data, but organizing the data for vertical computing has, in this case of the FFT computation, some obvious advantages. Indeed, the "interconnection" between x_{ij} and x_{ik} depends on the value of $|j - k|$ in s_i, while the "interconnection" between x_{ji} and x_{ki} does not depend on the value of $|j - k|$ because the two atoms are processed in the same cell c_i. For other kinds of computations maybe the horizontal computing must be chosen.

The capability of organizing data on two dimensions, vertically or horizontally, allows the use of a p-cell organization to perform computation on data sequences of size different from p. The job to adapt the computation on n-component sequences into a p-cell system organization is left to the compiler. If $n > p$, then the actual sequence will be loaded as few successive p-sized sequences in the two-dimension array $< s_1, \ldots, s_m >$. If $n < p$, then few n-component sequences are accommodated in one p-sized system sequence.

Thus, the MANY-CORE section emulates the computation on a two-dimension network of atoms with a very restrictive but inexpensive horizontal interconnection (due to the linear

ISBN: 978-1-61804-237-8

interconnection between the cells c_i) and a very flexible vertical interconnection (because of the random access in the first level **Buffer**).

C. Recapitulation

A synthetic representation of our abstract model for parallel computation is in Figure 6, where processing is done in the *fine grain* array MANY-CORE and the *coarse grain* array MULTI-CORE.

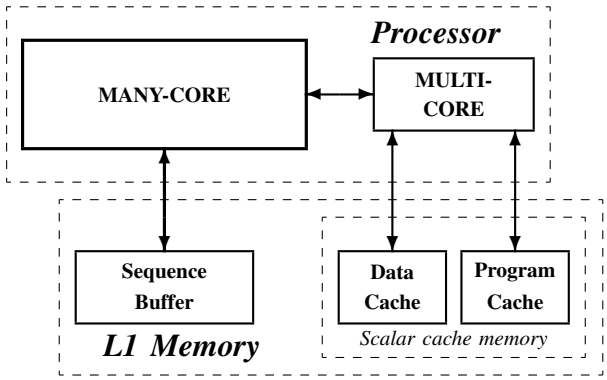

Fig. 6. **Parallel abstract machine model.** Between ***Processor*** and the first level memory hierarchy there are three channels: for program, atoms and sequences.

The first level of memory hierarchy, *L1 Memory*, consists of **Data Cache** for the scalar part of the computation performed mainly by MULTI-CORE, while **Sequence Buffer** is for the sequence computation performed in MANY-CORE. For the code, executed by both, MANY- and MULTI-CORE, there is the **Program Cache**. Due to its high "predictability" the data exchange for the intense computation is supported by a buffer-based memory hierarchy, unlike the complex computation which requests a cache-based memory hierarchy.

The "bottleneck" incriminated by John Backus, for slowing down and making more complex the computation, is not completely avoided. It is substituted only by an enlarged "jarneck", which allows a higher bandwidth between *Processor* and *L1 Memory*. But, while p, the number of cells in MANY-CORE, increases easily from hundreds to thousands, the bandwidth between *L1 Memory* and the system memory is more drastically limited by technological reasons (number of pins, power, technological incompatibilities between logic and DRAMs, ...). The only way to mitigate the effect of this limitation is to design the on-chip **Sequence Buffer** as big as possible in order to avoid frequent data exchange with the off-chip system memory.

There are many possible forms of implementing the abstract model, depending on the targeted application domain. For most of the applications, the use of data-parallel, intense speculative-parallel and reduction-parallel computations covers all the intense computational aspects needed to be accelerated by parallel computation. A good name for this case could be **MapReduce abstract machine model**.

IV. Backus FP Systems as Low Level, Architectural Description

Although Backus's concept of *Functional Programming Systems* (FPS) was introduced as an alternative to the *von Neumann style of programming* in [4], we claim that **they can be seen also as a *low level description* for the parallel computing paradigm**. In the following we use a FPS-like form to provide a low level functional description for the abstract model defined in the previous section. Thus, we obtain the ***virtual machine*** description of a parallel computer, i.e., the description defining the transparent interface between the hardware system and the software system in a real parallel computer. Starting from this virtual machine, the actual ***instruction set architecture*** could be designed for the physical embodiment of various parallel engines.

This section provides, following [4], the low level description for what we call Integral Parallel Machine (IPM). It contains functions which map objects into objects, where an object could be:

- atom, x; special atoms are: T (true), F (false), ϕ (empty sequence)
- sequence of objects, $< x_1, \ldots, x_p >$, where x_i are atoms or sequences
- $\perp$: undefined object

The set of functions contains:

- **primitive functions**: the functions performed atomically, which manage:
  - atoms, using functions defined on constant length sequences of atoms, returning constant length sequence of atoms
  - p-length sequences, where p is the number of cells of the MANY-CORE section
- **functional forms** for:
  - expanding to sequences the functions defined on atoms
  - defining new functions
- **definitions**: the programming tool used for developing applications.

A. Primitive Functions

An informal and partial description of a set of primitive functions follows.

- **Atom** : if the argument is an atom, then T is returned, else F is returned.

$$atom : x \equiv (x \; is \; an \; atom) \rightarrow T; F$$

  The function is performed by the controller or at the level of each c_i cell if the function is applied to each element of a sequence (see *apply to all* in the next subsection).
- **Null** : if the argument is the empty sequence, it returns T, else F.

$$null : x \equiv (x = \phi) \rightarrow T; F$$

  It is a reduction-parallel function performed by the reduction/loop network, *redLoopNet* (see Figure 5), which returns a predicate to the controller.

ISBN: 978-1-61804-237-8

- **Equals** : if the argument is a pair of identical objects, then returns T, else F.

$$eq : x \equiv ((x =< y, z >) \,\&\, (y = z)) \rightarrow T; F$$

If the argument contains two atoms, then the function is performed by the controller, else, if the argument contains two sequences, the function is performed in the cells c_i, and the final results is delivered to the controller through *redLoopNet*.

- **Identity** : is a sort of *no operation* function which returns the argument.

$$id : x \equiv x$$

- **Length** : returns an atom representing the length of the sequence.

$$length : x \equiv (x =< x_1, \ldots, x_i >) \rightarrow i; (x = \phi) \rightarrow 0; \perp$$

If the sequence is distributed in the MANY-CELL array, then a Boolean sequence, $< b_1, \ldots, b_p >$, with 1 on each position containing a component x_j is generated and *redLoopNet* provides $\sum_1^p b_j$ for the controller.

- **Selector** : if the argument is a sequence with no less than i objects, then the i-th object is returned.

$$i : x \equiv ((x =< x_1, \ldots, x_p >) \,\&\, (i \leq p)) \rightarrow x_i$$

The function is performed composing an intense speculative-parallel search operation with a data-parallel mask operation and the reduction-parallel OR operation which sends to the controller the selected object.

- **Delete** : if the first argument, k, is a number no bigger than the length of the second argument, then the k-th element in the second argument is deleted.
$del : x \equiv (x =< k, < x_1, \ldots, x_p >>) \,\&\, (k \leq p) \rightarrow$
$< x_1, \ldots, x_{k-1}, x_{k+1}, \ldots >$
The *ORprefix* circuit included in the *redLoopNet* subsystem selects the sequence $< x_k, x_{k+1}, \ldots >$, then the left-right connection in the MANY-CELL array is used to perform a one position left shift in the selected subsequence.

- **Insert data** : if the second argument, k, is a number no bigger than the length of the third argument, then the first argument is inserted in the k-th position in the last argument.
$ins : x \equiv (x =< y, k, < x_1, \ldots, x_p >>) \,\&\, (k \leq p) \rightarrow$
$< x_1, \ldots, x_{k-1}, y, x_k, \ldots >$
The *ORprefix* function performed in the *redLoopNet* subsystem selects the sequence $< x_k, x_{k+1}, \ldots >$, then the left-right connection in the MANY-CELL array is used to perform one position right shift in the selected sub-sequence and write y in the freed position.

- **Rotate** : if the argument is a sequence, then it is returned rotated one position left.

$$rot : x \equiv (x =< x_1, \ldots, x_p >) \rightarrow < x_2, \ldots, x_p, x_1 >$$

The *redLoopNet* subsystem and the left-right connection in the MANY-CELL array allows this operation.

- **Transpose** : the argument is a sequence of sequences which can be seen as a two-dimension array. It returns a sequence of sequences which represents the transposition of the argument matrix.
$trans : x \equiv$
$(x =<< x_{11}, \ldots, x_{1m} >, \ldots, < x_{n1}, \ldots, x_{nm} >>) \rightarrow$
$<< x_{11}, \ldots, x_{n1} >, \ldots, < x_{1m}, \ldots, x_{nm} >>$
There are two possible implementations. First, it is naturally solved in the MANY-CELL section because, loading each component of x "horizontally", as a sequence in *Buffer*, we obtain, associated to each cell c_i, the n-component final sequences on the "vertical" dimension (see paragraph 3.2.3):

$$\begin{gathered} < x_{11}, \ldots, x_{n1} > \text{ accessed by } c_1 \\ < x_{12}, \ldots, x_{n2} > \text{ accessed by } c_2 \\ \ldots \\ < x_{1m}, \ldots, x_{nm} > \text{ accessed by } c_m \end{gathered}$$

where each initial sequence is a m-variable "line" and each final sequence is n-variable "column" in **Buffer**. Second, using rotate and inter sequence operations.

- **Distribute** : returns a sequence of pairs; the i-th element of the returned sequence contains the first argument and the i-th element of the second argument.
$distr : x \equiv (x =< y, < x_1, \ldots, x_p >>) \rightarrow$
$<< y, x_1 >, \ldots, < y, x_p >>$
The function is performed in two steps: (1) generates the p-length sequence $< y, \ldots, y >$, then (2) performs $trans << y, \ldots, y >, < x_1, \ldots, x_p >>$.

- **Permute** : the argument is a sequence of two equally length sequences; the first defines the permutation, while the second is submitted to the permutation.
$perm : x \equiv$
$(x =<< y_1, \ldots, y_p >, < x_1, \ldots, x_p >>) \rightarrow$
$< x_{y_1}, \ldots, x_{y_p} >$
With no special hardware support it is performed in time $O(p)$. An optimal implementation, in time belonging to $O(log\, p)$, involves a *redLoopNet* containing a Waksman permutation network, with $< y_1, \ldots, y_p >$ used to program it.

- **Search** : the first argument is the searched object, while the second argument is the target sequence; returns a Boolean sequence with T on each match position.
$src : x \equiv (x =< y, < x_1, \ldots, x_p >>) \rightarrow$
$< (y = x_i), \ldots, (y = x_p) >$
It is an intense speculative-parallel operation. The scalar y is issued by the controller and it is searched in each cell generating a Boolean sequence, distributed along the cells c_i in MANY-CELL, with T on each match position and F on the rest.

- **Conditioned search** : the first argument is the searched object, the second argument is the target sequence, while the third argument is a Boolean sequence (usually generated in a previous search or conditioned search); the search is performed only in the positions preceded by T in the Boolean sequence; returns a Boolean sequencer with T on each conditioned match position.
$csrc : x \equiv$
$(x =< y, < x_1, \ldots, x_p >, < b_1, \ldots, b_p >>) \rightarrow$
$< c_1, \ldots, c_p >$

ISBN: 978-1-61804-237-8

where: $c_i = ((y = x_i)\ \&\ b_{i-1})\ ?\ T : F$.
The combination of `src` or `csrc` allows us to define a `sequence_search` operation (an application is described in [32]).

- **Arithmetic & logic operations** :

$$op2 : x \equiv ((x =< y, z >)\ \&\ (y, z\ atoms)) \rightarrow y\ op2\ z$$

where: $op2 \in \{add, sub, mult, eq, lt, gt, leq, and, or, ...\}$ or

$$op1 : x \equiv ((x = y)\ \&\ (y\ atom)) \rightarrow op1\ y$$

where: $op1 \in \{inc, dec, zero, not\}$. These operations will be applied on sequences of any length using the functional forms defined in the next sub-section.

- **Constant** : generates a constant value.

$$\bar{x} : y \equiv x$$

B. Functional Forms

A functional form is made of functions that are applied to objects. They are used to define complex functions, for an IPM, starting from the set of primitive functions.

- **Apply to all** : represents the ***data-parallel*** computation. The same function is applied to all elements of the sequence.

$$\alpha f : x \equiv (x =< x_1, \ldots, x_p >) \rightarrow < f : x_1, \ldots, f : x_p >$$

Example:
$\alpha\, add :<< x_1, y_1 >, \ldots, < x_p, y_p >> \rightarrow$
$< add :< x_1, y_1 >, \ldots, add :< x_p, y_p >>$
expands the function add, defined on atoms, to be applied on sequences, $<< x_1, \ldots, x_p >< y_1, \ldots, y_p >>$, transposed in a sequence of pairs $< x_i, y_i >$.

- **Insert** : represents the ***reduction-parallel*** computation. The function f has as argument a sequence of objects and returns an object. Its recursive form is:
$/f : x \equiv ((x =< x_1, \ldots, x_p >)\,\&\,(p \geq 2)) \rightarrow$
$f :< x_1, /f :< x_2, \ldots, x_p >>$
The resulting action looks like a sequential process executed in $O(p)$ cycles, but on the Integral Parallel Abstract Model (see Figure 5) it is executed as a reduction function in $O(log\ p)$ steps in the *redLoopNet* circuit.
- **Construction** : represents the ***speculative-parallel*** computation. The same argument is used by a sequence of functions.

$$[f_1, \ldots, f_n] : x \equiv < f_1 : x, \ldots, f_n : x >$$

- **Composition** : represents ***time-parallel*** computation if the computation is applied to a stream of objects. By definition:
$(f_q \circ f_{q-1} \circ \ldots \circ f_1) : x \equiv$
$f_q : (f_{q-1} : (f_{q-2} : (\ldots : (f_1 : x) \ldots)))$
The previous form is:
  - sequential computation, if only one object x is considered as input variable
  - pipelined *time-parallel* computation, if a ***stream*** of objects, $|x_n, \ldots, x_1|$, are considered to be inserted, starting with x_1, in c_1 in the MANY-CORE section (see Figure 5) so as in each successive two cells, c_i and c_{i+1}, are performed

$$f_i(f_{i-1} : (f_{i-2} : (\ldots : (f_1 : x_j) \ldots)))$$
$$f_{i+1}(f_i : (f_{i-1} : (\ldots : (f_1 : x_{j-1}) \ldots)))$$

Thus, the array of cells $c_1, \ldots, c_p$ can be involved to compute in parallel the function

$$f(x) = (f_q \circ f_{q-1} \circ \ldots \circ f_1) : x$$

for maximum q values of x.

- **Threaded construction** : is a special case of construction for: $f_i = g_i \circ i$ which represents the ***thread-parallel*** computation:
$\theta[f_1, \ldots, f_p] : x \equiv$
$(x =< x_1, \ldots, x_p >) \rightarrow < g_1 : x_1, \ldots, g_p : x_p >$
where: $g_1 : x_1$ represents an independent thread.
- **Condition** : represents a conditioned execution.
$(p \rightarrow f; g) : x \equiv$
$((p : x) = T) \rightarrow f : x; ((p : x) = F) \rightarrow g : x$
- **Binary to unary** : is used to express any function as an unary function.

$$(bu\ f\ x) : y \equiv f :< x, y >$$

This function allows the algebraic manipulation of programs.

C. Definitions

Definitions are used to write programs conceived as functional forms.

$$\textbf{Def}\ new\_function\_symbol \equiv functional\_form$$

Example : Let be the following definitions used to compute the *sum of absolute difference* (SAD) of two sequence of numbers:
$\textbf{Def}\ SAD \equiv (/+) \circ (\alpha ABS) \circ trans$
$\textbf{Def}\ ABS \equiv lt \rightarrow (sub \circ REV); sub$
$\textbf{Def}\ REV \equiv (bu\ perm < \bar{2}, \bar{1} >)$

D. Recapitulation

The beauty of the relation between the abstract machine components resulting from Kleene's model and the FPS proposed by Backus is that all the five meaningful forms of composition correspond to the main functional forms, as follows:

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

Let us agree that Kleene's model, and the FPS proposed by Backus represent a solid foundation for parallel computing, avoiding risky *ad hoc* constructs. The generic parallel structure proposed in the next section is a promising start in saving us from saying "Hail Mary" (see [22]) when we decide what to do in order to improve our computing machines with parallel features.

V. A Generic Parallel Engine

The fourth step in trying to restart the parallel computation domain (see subsection I.C) is to propose a simple and, as much as possible, efficient ***generic*** embodiment, able to provide both, a good use of energy and area, and an easy to program computing engine. This section describes the simplest one-chip generic parallel engine, already implemented in silicon: the ***Connex System*** chip. The next four subsections present the organization, the programming style, rough estimates for the 13 Berkeley's motifs, and the physical performances of this first embodiment, which will be, for sure, the subject of many successive improvements.

A. The Organization

The ***Connex System***, centered on the many-cell engine ***ConnexArray***^{TM}, is the first, partial embodiment (see [30]) of the *integral parallel abstract machine model* (see Figure 5). It is proposed as a possible initial version for a generic parallel engine. While the **Cache** module in Figure 5 is by default a hidden physical resource, the *Buffer* module is an explicit part of the architecture that differentiates strongly ***Connex System*** from a standard architecture. Let's start by representing the content of the *Buffer* by $A =< v_1, v_2, \ldots, v_m >$, a two-dimension array containing m p-scalar vectors:

$$v_1 =< x_{11}, \ldots, x_{1p} >$$
$$v_2 =< x_{21}, \ldots, x_{2p} >$$
$$\ldots$$
$$v_m =< x_{m1}, \ldots, x_{mp} >$$

where: each "column", $< x_{1i}, \ldots, x_{mi} >$, is a "vertical" vector of scalars associated to the computational cell c_i in Figure 5. In the first embodiment, represented in Figure 7, **Linear ARRAY of CELLS** is a linear array of p *Connex cells*, $cc_1, \ldots, cc_p$. Each cc_i contains c_i (see Figure 5) and the **local memory** which stores the associated "vertical" vector. The set of local memories represents the first level of **(Multi-level) Buffer** which allows the engine to work as a **stream processor**, not as a simple *vector processor*. Each cell is connected only to the left and to the right cell.

In the ***data-parallel mode*** each Connex cell, cc_i, receives, trough the **Broadcast** net, the same instructions issued by one of the threads, called the *control thread*, running on the **Multi-Threaded Processor**.

The ***reduction-parallel mode*** sends, to the control thread running on the **Multi-Threaded Processor**, scalars returned by functions performed, in the *log*-depth **Reduction** circuit, on sequences of atoms distributed over the array of cells cc_i. The reduction net is a pipelined tree *circuit* because there are only a small number of meaningful reduction functions in the current applications.

In ***speculative-parallel mode*** the difference between h_i and h_j can be done in two ways: (1) by some local parameters specific for each cell (example: its index), or (2) by a specific program loaded in the local memory of each cell. The process is triggered by the *control thread*, while the variable x is issued by the same thread.

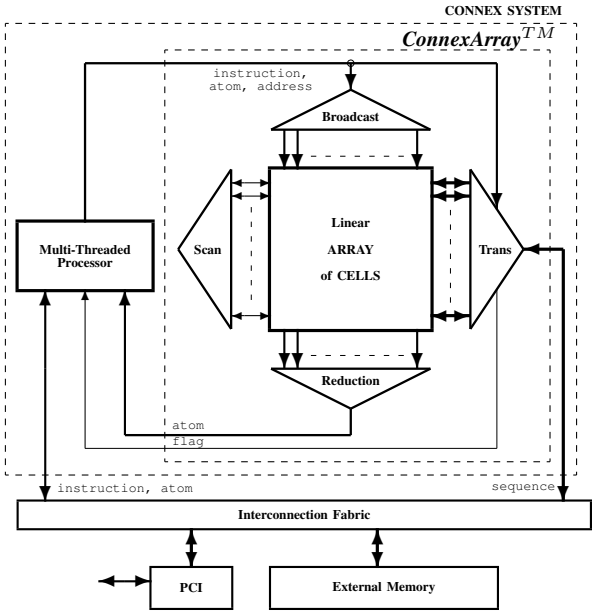

Fig. 7. **The Connex System.** It is centered on ***ConnexArray***^{TM}, a linear array of p execution units, each with its own local memory, connected to the external memory through the **Trans** network. The array has two loops, an external one through **Reduction** net, **Multi-Threaded Processor** and **Broadcast** net, and an internal one through the **Scan** net.

The ***time-parallel mode*** uses the linear interconnection network to configure a pipe of p machines, each controlled by the program stored in the **local memory**. The resulting pipeline machine receives a stream of data and processes it with a latency in $O(p)$. The i-th cell computes function f_i (see subsection 3.1.4). Cell 1 receives rhythmic a new component of the input stream x.

The ***thread-parallel mode*** can be implemented in two ways: (1) each cell works like an independent processor running the locally stored program on local data, or (2) **Multi-Threaded Processor** is used to run, usually, 4 to 16 threads, including the threads used to control the array of cells and the IO process. The second way is more frequently used in the current application domains.

The **Trans** module connects the array to the external memory. It is controlled by one thread running on the **Multi-Threaded Processor**, and works transparent to the computation done in the array.

The global loop closed over the array through **Scan** takes from each cell an atom and computes global functions sending back in the array a sequence of atoms. One example is the function `first` defined on a sequence of Booleans (see subsection 2.2). Another example is the permutation function for which the **Scan** network is programmed with $(-1+2\times log_2 p)$ p-length Boolean sequences.

ISBN: 978-1-61804-237-8

B. Machine Level Programming for the Generic Parallel Engine

A low level programming environment, called Backus-Connex Parallel FP system – BC for short –, was defined in Scheme for this generic parallel engine (see [19]). Some of the most used functions working on the previously defined array A are listed below:

```
(SetVector a v); a: address, v: vector content
(UnaryOp x)    ; x: scalar|vector
(BinaryOp x y) ; (x,y): scalar | vector
(Cond x y)     ; (x,y): scalar | vector
(RedOp v)      ; RedOp = {RedAdd, RedMax,...}
(ResetActive)  ; activate all cells
(Where b)      ; active where vector b is 1
(ElseWhere)    ; active where vector b was 0
(EndWhere)     ; return to previous active
```

Let us take as example the function *conditioned reduction add*, CRA, which returns the sum of all the components of the sequence $s_1 =< x_{11}, \ldots, x_{1p} >$ corresponding to the positions where the element in the sequence $s_2 =< x_{21}, \ldots, x_{2p} >$ is *less or equal than* the element of the sequence $s_3 =< x_{31}, \ldots, x_{3p} >$:

$$CRA(s_1, s_2, s_3) = \sum_{i=1}^{p} (x_{2i} \leq x_{3i}) \,?\, x_{1i} : 0$$

The computation of this function is expressed as follows:

$$\textbf{Def}\ CRA \equiv (/+) \circ (\alpha((leq \circ (bu\ del 1)) \rightarrow (id \circ 1); \bar{0})) \circ trans$$

where the argument must be a sequence of three sequences:

$$x =< s_1, s_2, s_3 >$$

and the result is returned as an atom. For

$$x =<< 1,2,3,4 >, < 5,6,7,8 >, < 8,7,6,5 >>$$

the evaluation is the following:
$CRA : x \Rightarrow$
$(/+) \circ (\alpha((leq \circ (bu\ del\ 1)) \rightarrow (id \circ 1); \bar{0})) \circ trans :$
$<< 1,2,3,4 >, < 5,6,7,8 >, < 8,7,6,5 >> \Rightarrow$
$(/+) \circ (\alpha((leq \circ (bu\ del\ 1)) \rightarrow (id \circ 1); \bar{0})) :<< 1,5,8 >, < 2,6,7 >, < 3,7,6 >< 4,8,5 >> \Rightarrow$
$(/+) :<$
$((leq \circ (bu\ del\ 1)) \rightarrow (id \circ 1); \bar{0}) :< 1,5,8 >,$
$((leq \circ (bu\ del\ 1)) \rightarrow (id \circ 1); \bar{0}) :< 2,6,7 >,$
$((leq \circ (bu\ del\ 1)) \rightarrow (id \circ 1); \bar{0}) :< 2,6,7 >,$
$((leq \circ (bu\ del\ 1)) \rightarrow (id \circ 1); \bar{0}) :< 4,8,5 >> \Rightarrow$
$(/+) :< ((leq :< 5,8 >) \rightarrow (id : 1); \bar{0}), \ldots, ((leq :< 8,5 >) \rightarrow (id : 4); \bar{0}) > \Rightarrow$
$(/+) :< ((leq :< 5,8 >) \rightarrow 1; \bar{0}), \ldots, ((leq :< 8,5 >) \rightarrow 4; \bar{0}) > \Rightarrow$
$(/+) :< (T \rightarrow 1; 0), (T \rightarrow 2; 0), (F \rightarrow 3; 0), (F \rightarrow 4; 0) > \Rightarrow$
$(/+) :< 1,2,0,0 > \Rightarrow 3$

At the level of machine language the previous program is translated into the following BC code:

```
(define (CRA v0 v1 v2 v3)
    (Where (Leq (Vec v2) (Vec v3)))
        (SetVector v0 (Vec v1))
      (ElseWhere)
        (SetVector v0 (MakeAll 0))
    (EndWhere)
    (RedAdd (Vec v0))
)
```

The function `CRA` returns a scalar and has as side effect the updated content of the vector `v0`.

C. Short Comments about Application Domains

The efficiency of ***Connex System*** in performing all the aspects of intense computation remains to be proved. In this subsection we sketch only the complex process of evaluation using the report "A View from Berkeley" [3]. Many decades just an academic topic, "parallelism" becomes an important actor on the market after 2001 when the clock rate race stopped. This research report presents 13 computational motifs which cover the main aspects of parallel computing. Short comments follows about how the proposed architecture and generic parallel engine work for all of the 13 motifs.

For **dense linear algebra** the most used operation is the inner product (IP) of two vectors. It is expressed in FP System as follows:

$$\textbf{Def}\ IP \equiv (/+) \circ (\alpha\times) \circ trans$$

while the BC code is:

```
(define (IP v0 v1)
    (RedAdd (Mult v0 v1))
)
```

allowing a linear acceleration of the computation.

For **sparse linear algebra** the band arrays are first transposed using the function `Trans` in a number of vectors equal with the width w of the band. Then the main operations are naturally performed using the appropriate `RotLeft` and `RotRight` operations. Thus, the multiplication of two band matrices is done on Connex System in $O(w)$.

For **spectral methods** the typical example is FFT. The vertical and horizontal vectors defined in the array A help the programmer to adapt the data representation to obtain an almost linear acceleration [5], because the **Scan** module is designed to hide the performance of the matrix transpose operation. In order to eliminate the slowdown caused by the rotate operations, the stream of samples are operated as vertical vectors (see also [16], where for example: FFT for 1024 floating point samples is done in less than 1 clock cycle per sample).

***N*-Body method** fits perfect on the proposed architecture, because for $j = 0$ to $j = n - 1$ the following equation is computed:

$$U(x_j) = \sum_i F(x_j, X_i)$$

using one cell for each function $F(x_j, X_i)$, followed b the sum (a *reduction* operation).

Structured grids are distributed on the two dimensions of the array A. Each processor is assigned a column of nodes. Each node has to communicate only with a small, constant number of neighbor nodes on the grid, exchanging data at the end of each step. The system works like a cellular automaton.

Unstructured grids problems are updates on an irregular grid, where each grid element is updated from its neighbor grid elements. Parallel computation is disturbed by the non-uniformity of the data distribution. In order to solve the

ISBN: 978-1-61804-237-8

non-uniformity problem a preprocessing step is required to generate an easy manageable representation of the grid. Slow-downs are expected compared with the structured grid.

The typical example of **mapReduce** computation is the Monte Carlo method. This method is highly parallel because it consists in many completely independent computations working on randomly generated data. It requires the add reduction function. The computation is linearly accelerated.

For **combinational logic** a good example is AES encryption which works in 4×4 arrays of bytes. If each array is loaded in one cell, then the processing is pure data-parallel with linear acceleration.

For **graph traversal** in [21] are reported parallel algorithms achieving asymptotically optimal $O(|V| + |E|)$ work complexity. Using sparse linear algebra methods, the breadth-first search for graph traversal is expected to be done on a Connex System in time belonging to $O(|V|)$.

For **dynamic programming** the Viterbi decoding is a typical example. The parallel strategy is to distribute the states among the cells. Each state has its own distinct cell. The inter-cell communication is done in a small neighborhood. Each cell receives the stream of data which is thus submitted to a speculative computation. The work done on each processor is similar. The last stage is performed using the reduction functions. The degree of parallelism is limited to the number of states considered by the algorithm.

Parallel **back-track** is exemplified by the SAT algorithm which runs on a p-cell engine by choosing $log_2\ p$ literals, instead of one on a sequential machine, and assigning for them all the values from $00\ldots0$ to $11\ldots1 = p - 1$. Each cell evaluates the formula for one value. For parallel **branch & bound** we use the case of the Quadratic Assignment Problem. The problem deals with two $N \times N$ matrices: $A = (a_{ij})$, $B = (b_{kl})$. The global cost function:

$$C(p) = \sum_{i}^{n} \sum_{j}^{n} a_{ij} \times b_{p(i)p(j)}$$

must be minimized finding the permutation p of the set $N = \{1, 2, \ldots, n\}$. Dense linear algebra methods, efficiently running on our architecture, are involved here.

Graphical models are well represented by parallel hidden Markov models. The architectural features reported in research papers refers to fine-grained data-parallel processor arrays connected to each node of a coarse-grained PC-cluster. Thus, our engine can be used efficiently as an accelerator for general purpose sequential engines.

For **finite state machine** (FSM) the authors of [3] claim that "nothing helps". But, we consider that the array of cells with their local memory loaded with non-deterministic FSM descriptions work very efficient as a speculative engine for applications such as deep packet inspection, for example.

At the end of this superficial introductory analysis, *which must be deepened by future investigations*, we claim that for almost all the computational motifs the ***Connex System***, in its simple generic form, perform at least encouraging if not pretty well.

D. About the First Implementation

Actual versions of the Connex System have already been implemented as part of a SoC designed for HDTV market: *BA1024* (see [28], [29] and [31]). The $90\,nm$ version of *BA1024*, with 1024 16-bit EUs, is in Figure 8. The last version, implemented in $65\,nm$, provides a peak performance of $400\,GOPS$[6], which translates in:

- area efficiency: $> 6\,GOPS/mm^2$
- power efficiency: $> \mathbf{120\,GOPS/Watt}$

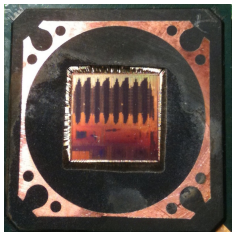

Fig. 8. **The Connex Chip.** The $90\,nm$ version of *BA1024* chip. The Connex System uses 60% of the total area.

Compared with a standard sequential processor implemented in $65\,nm$ results $20\times$ in area use and $100\times$ in power use (the evaluation was made for applications involving 32-bit integer and float operations). For integer (no-float) applications the previous improvements are around 4 times higher (the actual measurements were made for programs running HDTV *frame rate conversion* on the $65\,nm$ version of the *BA1024* chip). This first implementation of a generic parallel system suggests that *genuine parallelism is naturally green*.

The performances of this first embodiment of a *generic parallel structure* looks so good because the architecture is focused on intense computation, starting from the idea, largely exposed in [31], that only the strong ***segregation between intense computation and complex computation*** allows a good use of *area and power*. The standard sequential processors perform on the same hardware both, complex and intense computation, but they are designed and optimized only for complex computation. More, the multi-core systems are *ad-hoc* constructs gathering together few standard sequential processors able to perform efficiently no more than complex multi-threaded computations. Many-core GPU systems, like ATI or NVIDIA, do not obey to the golden rule "small is beautiful" stated in [3] (see pag. 20), to which we must add that "simple is efficient". Thus, the main GPUs do not obey (our version of) the "*kiss* principle": *keep it small & simple*. Unlike the Connex approach, they have complex hardware (hardware multipliers, float units), cache memories (why caches for a very predictable flow of data and programs!?), hierarchical organization (unrequested by any computational model), most of them imposed unfortunately by oppressive legacies.

VI. Conclusion

The intrinsic parallel computational model of Kleene fits perfect as theoretical foundation for parallel computation. Because, as we proved, primitive recursion and minimalization

[6]GOPS: **G**iga 16-bit **O**perations **P**er **S**econd

ISBN: 978-1-61804-237-8

are particular forms of compositions, only the composition rule is used to highlight the five forms of parallelism: data-, speculative-, reduction-, time-, thread-parallelism.

Integral Parallel Abstract Machine Model is defined as the model of the simplest generic parallel engine. Real embedded computation frequently involves all forms of parallelism for running efficiently complex applications.

Both, Kleene's model and Backus's programming style promote one-dimension arrays, thus supporting the simplest hardware configuration for the initial architectural proposal. If needed, a more complex solution will be adopted. But, till then we must struggle in this initial context inventing new algorithms by keeping the hardware as *small & simple* as possible. The linear array of cells is complemented by four *log*-depth networks – Broadcast, Reduction, Trans and Scan – in order to compensate its simplicity.

Segregation between intense computation and complex computation is the golden rule for increasing the computational power lowering in the same time both, cost (silicon area) and energy.

Parallelism is meaningful only if it is "green". The one-chip solution for parallelism provides a low power computational engine if it is developed starting from genuine computational and abstract models. The Connex Chip proves that our proposal for a generic one-chip solution provides two magnitude orders improvement in reducing energy per computational task compared with standard sequential computation, while GPGPU-like chips, an *ad hoc* solution for parallelism, are unable to provide the expected reduction of energy per computational task (they consume hundreds of Watts per chip and are unable to achieve, on average, more than 25% of their own peak performance running real applications [1]).

Programming the generic parallel structure is simple because of the simplicity of its organization, easy to hide behind a well defined architecture. Backus's FP Systems capture naturally the main features of a parallel engine and provide a flexible environment for designing an appropriate generic parallel architecture. In this paper we didn't touch the problem of a programming model, because it must be based, in our opinion, on the insights provided in the fifth stage of our approach – the algorithmic evaluation of the generic parallel structure against the 13 Berkeley's computational motifs (see 1.3). All the current programming models (such as Intel TBB, Java Concurrency, .Net Task Parallel Library, OpenMP, Clik++, CnC, X10) are basically focused on the multi-threading computation because the current market provides the multi-core hardware support for this kind of parallelism, while the scalable programming models presented in [20] are focused on the current many-core market products (such as AMD/ATI, NVIDIA).

Future work refers to the last stage of our approach (see subsection I.C). Preliminary evaluations tell us that almost all the 13 computational motifs, highlighted by *Berkeley's view* in [3], are reasonable well supported by the proposed generic parallel structure initially programmed in a sort of FP System programming language (for example BC). During this evaluation a lot of new features will be added to the simple generic engine described in the present paper. The resulting improvement process will allow a gradual maturation of the concept of parallelism, because *Nihil simul inventum est et perfectum* (Marcus Tullius Cicero).

ACKNOWLEDGMENT

The authors got a lot of support from the main technical contributors to the development of the *ConnexArray*^{TM} technology, the *BA1024* chip, the associated language, and its first application: Emanuele Altieri, Frank Ho, Bogdan Mîţu, Marius Stoian, Dominique Thiebaut, Tom Thomson, Dan Tomescu. The comments received from Bogdan Niţulescu helped a lot the improvement of the paper.

REFERENCES

[1] ***, http://www.siliconmechanics.com/files/c2050benchmarks.pdf. Silicon Mechanics, 2012.

[2] B. Alpern, L. Carter, and J. Ferrante, Modeling parallel computers as memory hierarchies. In Giloi, W. K. *et al.* eds. *Programming Models for Massively Parallel Computers*, IEEE Press, 1993.

[3] K. Asanovic, *et al.*, The landscape of parallel computing research: A view from Berkeley, 2006. At: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf.

[4] J. Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM* 21, 8 (August) 1978. 613641.

[5] C. Bîra, L. Gugu, M. Maliţa, G. M. Ştefan: Maximizing the SIMD Behavior in SPMD Engines, in *Proceedings of the World Congress on Engineering and Computer Science 2013 Vol I WCECS 2013*, 23-25 October, 2013, San Francisco, USA. 156-161.

[6] G. Blaauw, and F.P. Brooks, The structure of System/360, part I - Outline of the logical structure. IBM Systems Journal 3, 2, 1964. 119135.

[7] A. Church, An unsolvable problem of elementary number theory. *The American Journal of Mathematics* 58, 1936. 345363.

[8] D. Culler, *et al.*, LogP: Toward a realistic model of parallel computation. *Proc. of the ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, 1991. 112.

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

[10] E. W. Dijkstra, Co-operating sequential processes. *Programming Languages* Academic Press, New York, 43112. Reprinted from: Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, 1965.

[11] S. Fortune, and J. C. Wyllie, Parallelism in random access machines. *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing*, 1978. 114118.

[12] L. M. Goldschlage, A universal interconnection pattern for parallel computers. *Journal of the ACM* 29, 4, 1982. 10731086.

[13] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo, *Limits to Parallel Computation*. Oxford University Press, 1995.

[14] S. Kleene, General recursive functions of natural numbers. *Mathematische Annalen* 112, 5, 1936. 727742.

[15] R. E. Ladner, and M. J. Fischer, Parallel prefix computation. *Journal of the ACM* 27, 4, 1980. 831838.

[16] I. Lorentz, M. Maliţa, R. Andonie Fitting fft onto an energy efficient massively parallel architecture. *The Second International Forum on Next Generation Multicore / Manycore Technologies*, 2010

[17] B. M. Maggs, L. R. Matheson, and R. E. Tarjan, Models of parallel computation: a survey and synthesis. *Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences*, 2, 1995. 6170.

[18] M. Maliţa, and G. Ştefan, On the many-processor paradigm. *Proceedings of the 2008 World Congress in Computer Science, Computer Engineering and Applied Computing*, Las Vegas, vol. PDPTA08, 2008. 548554.

[19] M. Maliţa, and G. Ştefan, Backus language for functional nano-devices. *CAS 2011, vol. 2*, 331334.

[20] M. D. McCool, Scalable programming models for massively multicore processors. *Proceedings of the IEEE 96*, 5 (May), 2008. pp. 816831.

[21] D. Merrill, M. Garland, and A. Grimshaw, High Performance and Scalable GPU Graph Traversal, *Technical Report CS-2011-05*, Department of Computer Science, University of Virginia, Aug, 2011.

[22] D. Patterson, The trouble with multicore. *IEEE Spectrum*, July 1, 2010.
[23] E. Post, Finite combinatory processes. formulation I. The Journal of Symbolic Logic 1, 1936. 103105.
[24] V. R. Pratt, M. O. Rabin, and L. J. Stockmeyer, A characterization of the power of vector machines. *Proceedings of STOC1974*, 1974. 122134.
[25] V. R. Pratt, and L. J. Stockmeyer, A characterization of the power of vector machines. *Journal of Computer and System Sciences* 12, 2 (April), 1976. 198221.
[26] V. Sakar, *Parallel computation model*, 2008. At: http://www.cs.rice.edu/vs3/comp422/lecture-notes/comp422-lec20-s08-v1.pdf.
[27] G. Ştefan, and M. Maliţa, Granularity and complexity in parallel systems. *Proceedings of the 15 IASTED International Conf*, 2004. 442447.
[28] G. Ştefan, The CA1024: A massively parallel processor for cost-effective HDTV. *Spring Processor Forum: Power-Efficient Design*, May 15-17, Doubletree Hotel, San Jose, CA.
[29] G. Ştefan, *et al.*, The CA1024: A fully programmable system-on-chip for cost-effective HDTV media processing. *Hot Chips: A Symposium on High Performance Chips*. Memorial Auditorium, Stanford University.
[30] G. Ştefan, Integral parallel architecture in system-on-chip designs. *The 6th International Workshop on Unique Chips and Systems*, Atlanta, GA, USA, December 4, 2010, pp. 2326.
[31] G. Ştefan, One-chip TeraArchitecture. *Proceedings of the 8th Applications and Principles of Information Science Conference*. Okinawa, Japan, 2009.
[32] D. Thiebaut and M. Maliţa, "Real-time Packet Filtering with the Connex Array" *The 33rd Annual International Symposium on Computer Architecture*, Boston, MA, USA June 17-21, 2006, pp. 17-21.
[33] A. M. Turing, On computable numbers with an application to the Eintscheidungsproblem. *Proceedings of the London Mathematical Society* 42, 1936.
[34] J. von Neumann, First draft of a report on the EDVAC. *IEEE Annals of the History of Computing* 5, 4, 1993.
[35] L. G. Valiant, A bridging model for parallel computation. *Communications of the ACM* 33, 8 (Aug.), 1990. 103111.

Gheorghe M. Ştefan teaches digital design in *Politehnica* University of Bucharest. Its scientific interests are focused on digital circuits, computer architecture and parallel computation. In the 1980s, he led a team which designed and implemented the Lisp machine DIALISP. In 2003-2009 he worked as Chief Scientist and co-founder in *Brightscale*, a Silicon Valley start-up which developed the BA1024, a many-core chip for the HDTV market. More at `http://arh.pub.ro/gstefan/`.

Mihaela Maliţa teaches computer science at Saint Anselm College, US. Her interests are programming languages, computer graphics, and parallel algorithms. She wrote and tested different simulators for the Connex parallel chip. More at `http://www.anselm.edu/mmalita`.