# Functional Language for Map-Reduce Architecture

Malița M.
Computer Science Department
Saint Anselm College
Manchester, NH
mmalita@anselm.edu

Ștefan G. M.
Electronic Devices, Circuit and Architectures Dept.
Politehnica University of Bucharest, Fac. ETTI
Bucharest, Romania
gstefan@arh.pub.ro

*Abstract* — **Functional forms proposed by John Backus are used to specify a Map-Reduce Architecture. The associated functional programming language is defined as a Scheme-like language.**

*Key words* — **parallel computing, parallel architecture, parallel programming, functional languages, Scheme-like programming language**

## I. Introduction

The parallel abstract machine defined in [4], as an integral parallel machine able to perform data-, speculative-, reduction-, time-, and thread-parallelism, has a recursive Map-Reduce organization. It describes parallelism, from the level of one-chip solution to the cloud implementation, based on Stephen Kleene's mathematical model for computation [3]. The architectural description of this abstract machine is provided in the next section using the concept of Functional Programming Systems (FPS) introduced by John Backus [2]. In the third section is defined a high level programming language for the Map-Reduce organization and architecture. The "View from Berkeley" [1] will offer the theoretical environment to validate the whole approach.

## II. Map-Reduce Recursive Architecture

Backus' system contains functions which map objects into objects. An *object* is an *atom x* or a *sequence* $<x_1,\ldots,x_p>$, where $x_i$ are atoms or objects. There are three types of functions:

- *primitive functions*: performed atomically
- *functional forms*: expand functions on sequences
- *definitions*: develop programs

Because an object could be an atom or a sequence of atoms, FPS provide an adequate description for the computation able to exploit at the maximum the features of a multi- or many-core engine.

### A. Primitive Functions

Primitive functions are applied to atom or sequences. For example:

1) *Arithmetic & Logic:* consist of binary operations

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

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

or unary operations

$$\mathbf{op1}:x \equiv ((x=y) \& (y \text{ atom})) \rightarrow \mathbf{op1}\,y$$

where: $\mathbf{op1} \in \{\texttt{inc,dec,zero,not,…}\}$

2) *Operations on Sequences:* such as

*a) Selector:* for $x = <x_1,\ldots,x_p>$ the *i*-th element is selected, as follows $i:x \equiv x_i$

*b) Distribute:* an atom is distributed along a sequence $\mathbf{distr}:<y,<x_1,...,x_p>>) \equiv <<y,x_1>,...,<y,x_p>>$

*c) Transpose:* is applied to a two-dimension array $\mathbf{trans}:<<x_{11},...,x_{1m}>,...,<x_{n1},...,x_{nm}>>) \equiv <<x_{11},...,x_{n1}>,...,<x_{1m},...,x_{nm}>>$

### B. Functional Forms

The primitive functions are expanded to parallel execution using functional forms as follows:

1) *Apply to all:* for $x = <x_1,\ldots,x_p>$ the function $f$ is mapped as $\boldsymbol{\alpha}\mathbf{f}:x \equiv <f:x_1,\ldots,f:x_p>$

2) *Construction:* the object *x* is mapped to a sequence of functions as $[\mathbf{f_1},\ldots,\mathbf{f_p}]:x \equiv <f_1:x,\ldots,f_p:x>$

3) *Insert:* the sequence $x = <x_1,\ldots,x_p>$ is reduced to an atom by $\mathbf{/f}:x \equiv f:<x_1, /f<x_2,\ldots,x_p>>$

4) *Composition:*
$$(\mathbf{f_p}\circ\mathbf{f_{p-1}}\circ\ldots\circ\mathbf{f_1}):x \equiv f_p:(f_{p-1}:(\ldots(f_1:x)\ldots))$$

5) *Threaded Construction:* the sequence $x=<x_1,\ldots,x_p>$ is mapped to the sequence of functions, as follows:
$$\boldsymbol{\theta}[\mathbf{f_1},\ldots,\mathbf{f_p}]:x \equiv <f_1:x,\ldots,f_p:x>$$

The five functional forms correspond to five types of parallelism supported by Kleene's pure theoretical approach.

### C. Definitions

In order to write programs are used definitions:

```
Def new_function_symbol≡functional_form
```

For example, this program is computing the inner product of two sequences seen as vectors:

```
Def IP ≡ (/ add)∘(α mult)∘ trans
```

### B. Recursive Hierarchy

The structure associated with the previously defined architecture is a Map-Reduced recursive hierarchy represented in Fig. 1, where:

I. *eng*: execution or processing units associated with binary or unary primitive functions
II. *mem*: data or data & program memory
III. REDUCE: *log*-depth structure associated with the insert functional form
IV. CONTR: used to compose the functions defined using **Def**
V. MEMORY: stores data and programs for the entire system.

The linear array of cells, each containing a pair (*eng*, *mem*), represents the MAP level. In a recursive view, the sub-system MAP+REDUCE+CONTR could be assimilated with an *eng*, while MEMORY with a *mem* unit. Thus, the representation from Fig. 1 stands also for a recursive definition of the Map-Reduce organization/architecture, from the chip level up to cloud level.

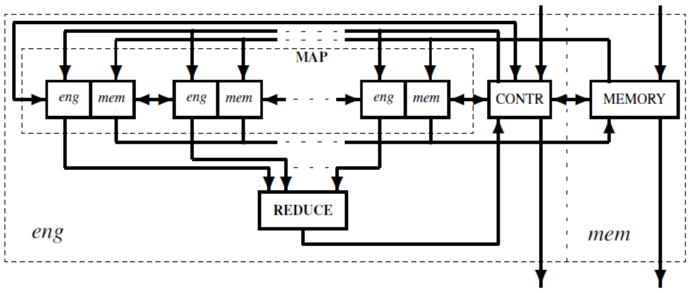

Fig. 1. Map-Reduce recursive organization.

Backus' description is able to cover any level in the hierarchical development of a Map-Reduce system.

## III. Map-Reduce Language

The language developed using a Scheme-type language catches both, the ***map*** aspect and ***reduce*** aspect of our architecture. We shall call it Map-Reduce Programming Language (MRPL).

The map functions are of three types: ***mapFunc***, ***mapArg***, ***mapBoth***.

```
(define(mapFunc func argList)
  (cond((null? argList)())
       (#t(cons(func(car argList))
               (mapFunc func (cdr argList))
 ))))

(define(mapArg funcList arg)
  (cond((null? funcList)())
       (#t(cons((car funcList)arg)
               (mapArg (cdr funcList)arg)))
 ))

(define(mapBoth funcList argList)
  (cond((or(null? funcList)
           (null? argList))())
       (#t(cons((car funcList)
                (car argList))
                (mapBoth (cdr funcList)
                         (cdr argList))))
  ))
```

Here are a few examples of using the map function:

```
  (mapFunc Inc '(1 2 3 4))->(2 3 4 5)
(mapArg '(Add Sub Mult Div) '(2 5))->
        (7 -3 10 0.4)
```

```
(mapBoth '(Add Mult Sub Div)
        '((2 5)(3 4)(4 3)(5 2)))->(7 12 1 2.5)
```

The reduce functions have the form:

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

Here are examples of reduction function use:

```
(myReduce Add   '(1 2 3 4 5))  -> 15
(myReduce Max   '(3 2 5 8 1 6))-> 8
(myReduce BwOr '(3 2 7 6 3))  -> 7
```

A program at any level in the recursive hierarchy can be described using the above introduced structures. For example, the inner product of two vectors is computed by:

```
(define(inProd firstVect secondVect)
 (myReduce Add (mapFunc Mult
                  (firstVect secondVect)
               )
 ))
```

## IV. Concluding Remarks

The Map-Reduce recursive architecture is supported and validated by the functional forms introduced by John Backus.

The MRPL is an appropriate functional language for writing programs for parallel accelerators. The functional aspects of MRPL allow the same programming style for any level in the recursive hierarchy of parallel computing machines (see Fig.1)

## Bibliography

[1] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams and Katherine A. Yelick, "The landscape of parallel computing research: A view from Berkeley", Technical Report No. UCB/EECS-2006-183, December 18, 2006.

[2] John Backus, "Can Be Programming Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", Communications of the ACM, 21, 8, 1978, pp. 613-641.

[3] Stephen Kleene, "General recursive functions of natural numbers", Mathematische Annalen 112, 5, 1936.

[4] Gheorghe M. Ștefan, Mihaela Malița, "Can One-Chip Parallel Computing Be Liberated from Ad Hoc Solutions? A Computational Model Based Approach and Its Implementation", 18th International Conference on Circuits, Systems, Communications and Computers (CSCC 2014), Santorini Island, Greece, July 17-19, 2014, pp. 582-597.