U.P.B. Sci. Bull., Series C, Vol. 78, Iss. 4, 2016 ISSN 2286-3540

ALL-PAIRS SHORTEST PATH MODIFIED MATRIX-MULTIPLICATION BASED ALGORITHM FOR A ONE-CHIP MapReduce ARCHITECTURE

Voichita DRAGOMIR[1]

An implementation of a newly developed parallel all-pairs shortest path algorithm based on modified matrix-multiplication on a new one-chip many-core structure with a MapReduce architecture is presented. The generic structure's main features and performances are described together with the new general purpose features added for upgrading the existing generic structure in order to run the best performance of this algorithm. The main outcome of the presented research is that our MapReduce architecture, in spite having a simpler and smaller structure, has the same theoretical time performance as the hypercube architecture. Also, the actual energy performance of our architecture is 7 pJ for 32-bit integer operation, compared with the ~ 150 pJ per operation of the current many-cores.

Keywords: parallel computing; MapReduce; many core; all-pair shortest path; matrix multiplication; parallel algorithm

1. Introduction

Many structures - physical such as transportation or road networks, social such as friendship networks, or virtual such as computer networks - have natural graph representations. Graphs, one of the most versatile data structures, play an important role in many domains because they provide an easy and systematic way to model many problems. Because of the ever-expanding amounts of computation and captured data [1], both researchers and industry are confronted with the need to process increasingly large amounts of data, essential form by graphs and solved using standard graph algorithms. So, graph processing is becoming increasingly important nowadays. In graph theory, the shortest path problem is the problem of finding a path between two vertices (nodes) in a graph such that the sum of the weights of its constituent edges is minimum. There are two algorithms: Single-Source Shortest Path (finding the shortest path from a single vertex to every other vertex) and All-Pair Shortest Path (APSP) algorithms (finding the shortest path between all pairs of vertices). The optimal sequential algorithm for APSP is in $O(N^3)$; N being the number of vertices in the graph. This paper is about a parallel formulation of an All-Pair Shortest Path algorithm: the Modified Matrix-Multiplication based algorithm on a one-chip MapReduce architecture.

[1] Teaching assistant., Dept.of Electronic Devices, Circuits and Architectures, University POLITEHNICA of Bucharest, Romania, e-mail: voichita.dragomir@upb.ro

The modified matrix multiplication is not an optimal algorithm. Its sequential time is in $O(N^3 logN)$, but we use it because of its simplicity and the efficiency on our new structure, the one-chip many-core MapReduce engine. This new structure is described in chapter 3. This APSP algorithm is called matrix-multiplication based, because it uses the modified matrix multiplication which substitutes the multiplication operation with addition and addition operation with minimum. Our approach is different than what has been done so far. Let us take a look on the current existing solutions in the next chapter. Chapter three describes the new structure we are working on. It performs best on matrix-vector operations. Therefore, we designed an APSP algorithm based on the dense matrix representation of graph, presented in chapter four. We did not cover the sparse matrix version because we can't talk of one in this case, due to the fact that the only zeroes that appear in the matrix are on the main diagonal. Chapter five contains the new general purpose features added for upgrading the existing generic structure, in order to run the best performance of this algorithm. To determine the efficiency of the parallel algorithms we developed for the MapReduce structure, we are comparing them to the most efficient and used parallel structure today, the distributed hypercube parallel computer. Concluding remarks are presented in chapter six.

2. Current existing solutions

So far, parallel APSP algorithms have been implemented on multi-core processors, with shared external memory. They are limited in the number of cores, the memory size and they are non-scalable for big data size [2] [3]. Another existing implementation is cloud MapReduce architecture, with distributed memory, where the MapReduce approach is limited by the latency introduced by the communication network [4] [5], which means a significant increase in energy and time use. For example, if the interconnection network used is a hypercube – one of the most efficient solution for communication nowadays – then the size of the entire system belongs to $O(PlogP)$ with a latency in communication in $O(logP)$, where P represents the number of cells. What is new in our approach is that we are going to implement the parallel APSP algorithm on a one-chip many-core structure not on multi-core or distributed computing. There are other one-chip MapReduce approaches. For example, the Intel SCC family. In [6] and [7] two different MapReduce applications are presented. The use of this general-purpose array of processors has a much slower response, because it has no more than 48 cores (which are also much too complex for solving this kind of problem) and the MapReduce functionality is implemented in software, not hardware, as in our case.

3. New Generic One-Chip MapReduce Architecture

The research presented in this paper is part of a larger project set out to improve, this generic new architecture, the one-chip MapReduce architecture. For this purpose, we address the collection of algorithm families important in parallel computing that the Berkeley research report on parallel computation talks about, naming them the 13 "dwarfs" [12]. The "dwarf" considered in this paper is *Graph traversal* and the APSP with modified matrix-multiplication is one of the graph traversal algorithms.

3.1. The Structure

The structure we work on is a one-chip MapReduce architecture, presented in Fig. 1, where:

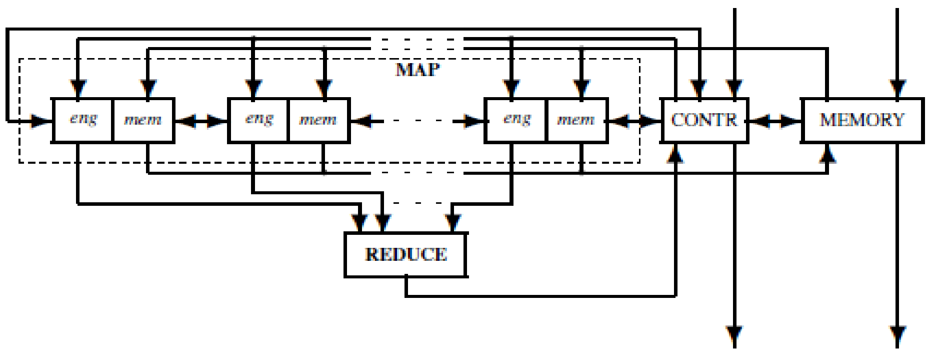

Fig. 1. MapReduce one-chip architecture

- pairs *eng-mem* in the MAP section; they correspond to each cell from a linear array of hundreds or thousands of cells containing execution units and local memory of few KB, and consist of:
  – *eng*, the engine, which is an execution unit
  – *mem*, the local memory to store data
- REDUCE unit; is a *log*-depth tree structure used to compute some reduction functions (add, min, max, ...) which provides for the controller CONTR a scalar from a vector.
- CONTR, a controller used as sequencer; a processing unit which issues in each cycle an instruction and various data distributed, if needed, in the array of cells.
- MEMORY, a memory resource for data and programs.

The cellular structure of the generic structure is accompanied by the standard scalar processing structure used as controller. In the cellular structure all the resources are of vectorial type. The instruction set architecture works on four storage resources:

- vectorial resources, distributed along the array of cells

• scalar resources, in the controller
• control resources, in the controller
• evaluation resources, used to evaluate the performance of the execution
all of which are described as follows:

```
// vector domain, V
reg [x-1:0] ixVect[0:(1<<x)-1] ;    // index read-only vector
reg [a-1:0] actVect[0:(1<<x)-1] ;   // activation vector
reg         boolVect[0:(1<<x)-1] ;  // Boolean vector
reg [n-1:0] accVect[0:(1<<x)-1] ;   // accumulator vector
reg         crVect[0:(1<<x)-1] ;    // carry vector
reg [v-1:0] addrVect[0:(1<<x)-1] ;  // address vector
reg [n-1:0] vectMem[0:(1<<x)-1][0:(1<<v)-1] ; // vector memory

// scalar domain, S
reg [n-1:0] acc ;                   // scalar accumulator
reg         cr ;                    // scalar carry
reg [s-1:0] addr ;                  // scalar address
reg [n-1:0] mem[0:(1<<s)-1] ;       // scalar memory

// control resources
reg [p-1:0] pc ;                    // program counter
reg [31:0] ir ;                     // instruction register
reg [31:0] progMem[0:(1<<p)-1] ;    // program memory

// evaluation resources
reg [31:0] cc ;                     // cycle counter
reg        ccEnable ;               // cycle counter enable
```

The generic structure starts with the simplest and smallest resources, like:
• each of the 2^x cell's engine is an execution unit (not a processing unit)
• both, the execution unit of the controller and the execution unit of each cell are accumulator based
• 32-bit interface to the external memory

The described structure has a few physical implementation versions. The last out of the three implemented versions, issued in 2008, in 65nm standard cells technology [8], provides the following performances: 100 GOPS/Watt and 5 GOPS/mm^2, while the current sequential engines (x86 architecture, for example) have, in the same technology: ~1 GOPS/Watt and ~0,25 GOPS/mm^2 (GOPS stands for Giga Operations Per Second).

The size of the structure is in $O(P)$, where P is the number of cells, while the communication latency between the array and the controller is in $O(logP)$.

3.2. The Instruction Set Architecture

Instruction Set Architecture defines the operations performed over the two data domains: scalar domain, S, and vector domain, V. Therefore, the structure of the MapReduce generic architecture consists of two parts – one associated to

Controller and another for Array –, and the resulting instruction set architecture, $ISA_{MapReduce}$, is a dual one:

$$ISA_{MapReduce} = (ISA_S \times ISA_V)$$

where:

- $ISA_S = SS_{arith\&logic} \cup SS_{control} \cup SS_{communication}$ is the ISA associated to the Controller, with its three subsets of instructions
- $ISA_V = SS_{arith\&logic} \cup SS_{spatialControl} \cup SS_{transfer}$ is the ISA associated to the cellular Array, with its three subsets of instructions

In each clock cycle from the program memory of the controller a pair of instructions is read: one from ISA_S, to be executed by Controller, and another from ISA_V to be executed by Array.

The $SS_{arith\&logic}$ are identical in the two ISAs. The $SS_{communication}$ subset controls the internal communication between array and controller and the communication of the MapReduce system with the host computer. The $SS_{transfer}$ subset controls the data transfer between the distributed local memory of the array and the external memory of the system. The $SS_{control}$ subset consists of conventional control instructions in a standard processor. We must pay more attention to the $SS_{spatialControl}$ subset used to perform the specific spatial control in an array of execution units. The main instructions in $SS_{spatialControl}$ subset are:

`activate`: all the cells of the array are activated for executing the next instructions

`where`: maintains active only the active cells where the condition `cond` is fulfilled; for example: `where(zero)` maintains active only the active cells where the accumulator is zero (it corresponds to the `if(cond)` instruction form the $SS_{control}$ subset)

`elsewhere`: activates the cells inactivated by the associated `where(cond)` instruction (it corresponds to the `else` instruction form the $SS_{control}$ subset)

`endwhere`: restores the activations existed before the previous `where(cond)` instruction (it corresponds to the `endif` instruction form the $SS_{control}$ subset)

3.2.1. The Instruction Structure

The instruction format for the MapReduce engine allows issuing two instruction at a time, as follows:

```
mrInstruction[31:0] = {controllerInstr, arrayInstr} =
                      {{instr[4:0], operand[2:0], value[7:0]},
                      {instr[4:0], operand[2:0], value[7:0]}}
```

where:

`instr[4:0]` : codes the instruction

`operand[2:0]` : codes the second operand used in instruction

`value[7:0]` : is mainly the immediate value or the address

The field `operand[2:0]` is specific for our accumulator centered architecture. It mainly specifies the second *n*-bit operand, `op`, and has the following meanings:

`val` : immediate value
```
    op = {{(n-8)value[7]}, value[7:0]}
```
`mab` : absolute from local memory
```
    op = mem[value]
```
`mrl` : relative from local memory
```
    op = mem[value + addr]
```
`mri` : relative from local memory and increment
```
    op = mem[value + addr]; addr <= value + addr;
```
`cop` : immediate with co-operand – `coop`
```
    op = coop ;
```
`mac` : absolute from local memory with co-operand
```
    op = mem[coop];
```
`mrc` : relative from local memory with co-operand
```
    op = mem[value + coop] ;
```
`ctl` : control instructions ;

where the co-operand of the array is the accumulator of the controller: `acc`, while the co-operand of the controller is provided by the four outputs of reduction section of the array:

`redSum`: the sum of the accumulators from the active cells: $\sum_0^p acc_i$

`redMin`: the minimum value of the accumulators from the active cells: $Min_0^p acc_i$

`redMax`: the maximum value of the accumulators from the active cells: $Max_0^p acc_i$

`redBool`: the sum of the active bit from the active cells: $\sum_0^p bool_i$

3.2.2. The Assembler Language

The assembly language provides a sequence of lines each containing two instructions, one for Controller (containing the prefix `c`) and another for Array. Some of the line are labeled, `LB(n)`, where *n* is a positive integer.

Example 1. *The program which provides in the controller's accumulator the sum of indexes is:*

```
cNOP;      ACTIVATE; // activate all cells
cNOP;      IXLOAD;   // load the index of each cell in accumulator
cCLOAD(0); NOP;     // load in controller's accumulator the sum of
                    indexes
```

3.3. Concluding About Our MapReduce Architecture

This one-chip MapReduce architecture is used as an accelerator in application fields like video [9], encryption, data mining and also for efficiently generating pseudo-random number sequences [10].

Matrix-vector operations are very frequent and it is important to cover this aspect. The system we work with is a *many-core* one-chip with a *MapReduce* architecture and it performs best on matrix-vector operations. The operation supposes a series of scalar (dot, inner) products whose results must be assembled in a vector stored back into the array's distributed memory. The generic MapReduce structure performs very efficiently the vector multiplication (on the Map section of the engine) and then the *n*-ary addition (in the Reduce section of the engine). This is why we choose to do the all-pair shortest path based on the modified matrix multiplication algorithm, although it's not an optimal algorithm.

4. All-Pairs Shortest Path Modified Matrix-Multiplication Based Algorithm for Dense Matrix Representation of Graphs

Important note: only the dense matrix version of the modified multiplication algorithm is considered, because the only zeroes that appear in the matrix are on the main diagonal and this means dense matrix, so there is no sparse case.

4.1. The Algorithm

Modified matrix multiplication algorithm assumes to substitute the multiplication with addition and addition with minimum, such that in computing the elements of the resulting matrix instead of:

$$c_{ij} = \sum_{k=1}^{k=N} a_{ik} \times b_{kj}$$

we use:

$$c_{ij} = \min_{k=1}^{k=N} (a_{ik} + b_{kj})$$

If, the graph is represented by the weighted adjacency matrix $\boldsymbol{A}$, then, for a graph with the number of vertexes $|V| = N$, the output of the APSP algorithm will be the $N \times N$ matrix $D = \boldsymbol{A}^{N-1}$, computed using, instead of the matrix multiplication, the modified matrix-multiplication algorithm.

4.2. The Program

The program is based on the efficiency of our MapReduce architecture in computing the inner product. The Map section computes the sums $a_{ik} + b_{kj}$, for the modified algorithm, while the Reduce section, pipeline connected, computes the minimum, for the modified algorithm. Thus, the vector-matrix product is very efficiently computed by the following program stored as the file `modifiedMatrixVectMult.v` of form:

```
       cLOAD(6);   NOP;        // acc <= last line of matrix
       cLOAD(0);   CADDRLD;    // acc <= N; addr[i] <= acc
       cVSUB(1);   RLOAD(0);   // acc <= N-1; load last matrix M1 line
       cNOP;       ADD(0);     // add line with vector
LB(6); cCPUSHL(1); RILOAD(127);// push reduction min; load next line
       cBRNZDEC(6);ADD(0);     // test end of loop; line-vector add
                               // latency = 1 + 0.5 log N
       cNOP;       NOP;        // latency
       cNOP;       NOP;        // latency
       cLOAD(9);   SRLOAD;     // acc <= mem[9]; load result in acc
       cVADD(1);   CSTORE;     // acc <= acc+1; store in vector memory
```

The loop consists of the following two lines:

```
LB(6); cCPUSHL(1);  RILOAD(127); // push reduction min; load next line
       cBRNZDEC(6); ADD(0);      // test end of loop; line-vector add
```

Actually the hole program stays mainly on these two lines. The weight of the program from the execution time point of view falls on these two lines, the loop labeled with `LB(6)`. The execution time for a $N \times N$ matrix is:

$$T_{Vector\text{-}Matrix}(N) = 2N + 6 + 0.5 logP \in O(N)$$

with $N \leq P$, where N is the number of vertices in the graph which gives the dimension of the matrix and P is the number of the execution units from the engine's array of cells. $0.5 logP$ is due to the latency introduced by the reduction network. For $N = 1024$ the execution time for vector matrix multiplication is

$$T_{Vector\text{-}Matrix}(N) = 2048 + 6 + 5 = 2048 + 11$$

which means that only 0.5% of the overall time is spent by the program outside the loop. This small and concise loop is possible because:

- the control of the loop is performed by the controller in parallel with the computation done in the Map section and Reduce section.
- the Map section and Reduce section are pipelined and, thus work in parallel contributing to the computing of two successive inner products.
- the instruction `cCPUSHL` we added in the instruction set of the MapReduce engine builds the result vector in parallel.

Based on the previous program, the modified matrix-matrix multiplication program stored as the file `modifiedMatrixMatrixMult.v` is:

```
'include "03_matrixTranspose.v"
                          // select the first N cells only
cLOAD(0);  IXLOAD;        // acc <= N; acc[i] <= index
cLOAD(0);  CSUB;          // acc[i] <= index - N
cSTORE(5); WHERECARRY;    // select only the first N cells
cLOAD(1);  NOP;
cADD(0);   NOP;
```

```
        cVSUB(1);  NOP;
        cSTORE(6); NOP;         // mem[6] <= last line in M1
        cLOAD(2);  NOP;
        cSTORE(9); NOP;
        cLOAD(3);   NOP;
        cSTORE(10); NOP;

// matrix M1 "x" matrix transpose M2T

LB(7); cLOAD(10);  NOP;       // acc = address the transp. matrix
       cVADD(1);   CALOAD;    // acc <= acc+1; acc[i]<=memVec[addr]
       cSTORE(10); STORE(0); // save the pointer; load line at 0

       `include "03_modifiedMatrixVectMult.v"

       cSTORE(9);  NOP;
       cLOAD(5);   NOP;       // acc = loopCounter
       cVSUB(1);   NOP;       // decrement loopCounter
       cSTORE(5);  NOP;       // store back loopCounter
       cBRNZ(7);   NOP;
```

The execution time for the modified matrix-matrix multiplication is:

$$T_{mMMM} = 3N^2 + 44N + 0.5NlogP + 2 \in O(N^2)$$

with $N \le P$, where P is the number of execution units and N is the number of vertices in the graph.

For $N = 1024$ we obtain the result:

$$T_{mMMM} = 3.048N^2 \ cycles$$

out of which $3N^2$ are consumed in the following lines:

- from the matrix transpose program (named `03_matrixTranspose.v`) the following two lines are executed in N cycles:

```
LB(2); cBRNZDEC(2);GLSHIFT; // global left shift cycle times
...
LB(3); cBRNZDEC(3);GRSHIFT; // global right shift N-cycles times
```

- from the modified vector–matrix multiplication program (named `03_modifiedMatrixVectMult.v)` the following two lines are executed in 2N cycles:

```
LB(2); cCPUSHL(1);  RILOAD(63);// push reduction sum; load line
       cBRNZDEC(2); ADD(0);    //test end of loop; line vector add
```

The execution time for APSP is:

$$T_{APSP} = (T_{mMMM} + 11)\ logP = (3N^2 + 44N + 0.5NlogP + 13)\ logP \quad \in \mathbf{O\ (N^2 logP)}$$

with $N \leq P$, where N is the number of vertexes and P is the number of execution units.

4.3. The Test Program and the Results

For running and evaluating the algorithm on the described architecture we used a Verilog based simulator and we obtained the following.
We considered the example represented in Fig. 2 (see in [11], Fig. 7.7).

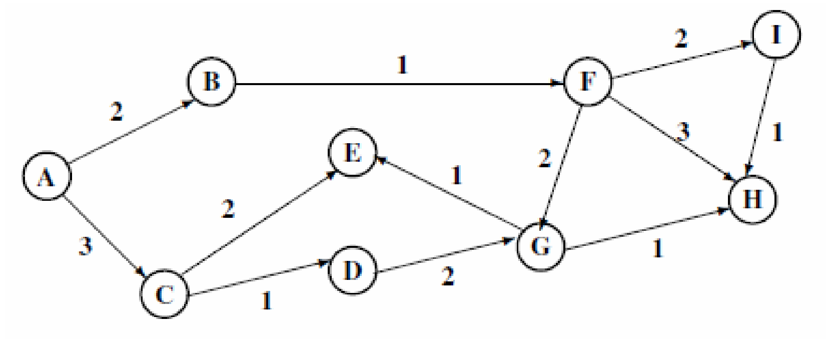

Fig. 2. The graph considered as example

There are N = 9 vertexes in the graph. The corresponding weighted adjacency matrix **A** is:

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*
A	0	2	3	99	99	99	99	99	99
B	99	0	99	99	99	1	99	99	99
C	99	99	0	1	2	99	99	99	99
D	99	99	99	0	99	99	2	99	99
E	99	99	99	99	0	99	99	99	99
F	99	99	99	99	99	0	2	3	2
G	99	99	99	99	1	99	0	1	99
H	99	99	99	99	99	99	99	0	99
I	99	99	99	99	99	99	99	1	0

The value 99 stands for ∞, meaning there is no path between the two vertexes.
For N = 9 vertexes, we are computing by turn $\mathbf{A}^2$, $\mathbf{A}^4$ and $\mathbf{A}^8$ using the modified matrix multiplication.
After running the program the results are:

```
  |           A  B  C  D  E  F  G  H  I
--+-----------------------------------------------------
A | vect[38] =  0  2  3  4  5  3  5  6  5 x x x x x x x
B | vect[39] = 99  0 99 99  4  1  3  4  3 x x x x x x x
C | vect[40] = 99 99  0  1  2 99  3  4 99 x x x x x x x
D | vect[41] = 99 99 99  0  3 99  2  3 99 x x x x x x x
E | vect[42] = 99 99 99 99  0 99 99 99 99 x x x x x x x
F | vect[43] = 99 99 99 99  3  0  2  3  2 x x x x x x x
G | vect[44] = 99 99 99 99  1 99  0  1 99 x x x x x x x
H | vect[45] = 99 99 99 99 99 99 99  0 99 x x x x x x x
I | vect[46] = 99 99 99 99 99 99 99  1  0 x x x x x x x
```

where the resulting matrix contains the minimal distances from each vertex to another. For example, the fourth component of `vect[38]` which has the value 4 represents the minimum distance from vertex A to vertex D. The value 99, like in the first component of `vect[39]`, means that there is no path from vertex A to vertex B, and so on. So the list of existing minimum distances is:

```
((A B 2)(A C 3)(A D 4)(A E 5)(A F 3)(A G 5)(A H 6)(A I 5)
 (B E 4)(B F 1)(B G 3)(B H 4)(B I 3)(C D 1)(C E 2)(C G 3)
 (C H 4)(D E 3)(D G 2)(D H 3)(F E 3)(F G 2)(F H 3)(F I 2)
 (G E 1)(G H 1)(I H 1))
```

The rest of the vertexes don't have a path between them. The vectors have 16 elements because we made the simulation with an engine having 16 processing units. Because the graph we considered has only 9 vertexes (N = 9), the last components of the vectors are unused (their value is undefined, x). The running time for the APSP algorithm, obtained in the simulation environment, is: T_{APSP} = 2041 cycles. So, for N = 9, $T_{APSP} = 8.39N^2logP$ cycles.

5. Upgraded Version of MapReduce Architecture and Organization

During the process of developing the parallel APSP algorithm based on modified matrix-multiplication on our new one-chip many-core structure with a MapReduce architecture we discovered and added some new general purpose features for the structure and so we were able to upgrade the existing generic structure in order to achieve a better, maybe the best performance. These improvement are the following:

- a serial register distributed along the array added to the generic design
- a direct loop from the Reduce module to Array (the loop does not go through the Controller, the results are sent directly to the Array). This means significantly faster response in time and less energy consumption (see Fig. 3)

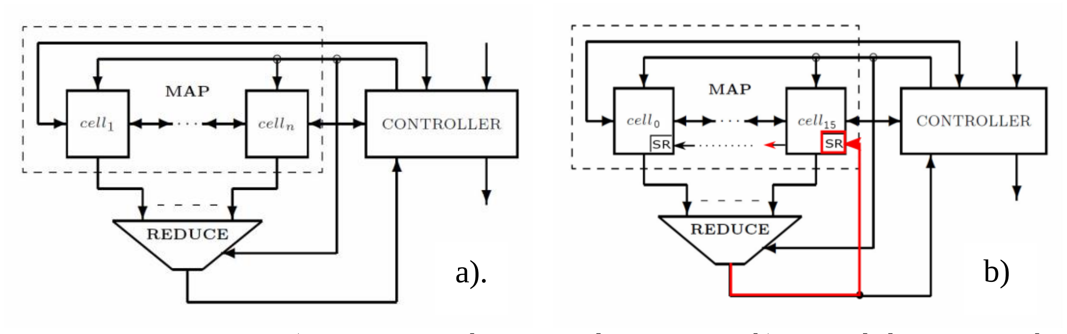

Fig. 3. a). Generic one-chip MapReduce structure; b). Upgraded structure with serial register and direct loop

- two new instruction added to the set of instructions:

```
pushl = 5'b01110, // push left to global shift register(c)
pushr = 5'b01111, // push right to global shift register(c)
```

So, the serial register will do the following:

```
reg [n-1:0] serialReg[0:(1<<x)-1]          ;
...
case(contrOpCode)
     pushl: if(i == 0) serialReg[i] <= op            ;
             else      serialReg[i] <= serialReg[i-1];
     pushr: if (i == ((1<<x) - 1))
                       serialReg[i] <= op;
             else      serialReg[i] <= serialReg[i+1];
    ...
endcase
```

As a result of this upgrading we have obtained an increase in performance by reducing the execution time for the matrix-vector multiplication from *O(NlogP)* to *O(N)*. Let us see how it worked. Our generic MapReduce structure performs very efficiently the vector-vector multiplication. But, the sum involved in the inner product and the composing of the resulting vector for matrix-vector multiplication request an embarrassingly long sequence of operations, and more than that the execution time depends logarithmically by *P*. Indeed, the loop that would be written for the generic version looks as follows:

```
00 LB(6); cSTORE(5); RILOAD(63); // load next matrix line
01        cNOP;      MULT(0);    // multiply line with vector
02        cLOAD(5);  IXLOAD;     // acc <= ixCounter; acc[i]<=ixVector[i]
03        cNOP;      CSUB;       // acc[i] <= acc[i]-acc;
04        cNOP;      WHEREZERO;  // for reduction latency
05        cNOP;      NOP;        // for reduction latency if P > 16
06        cNOP;      NOP;        // for reduction latency if P > 64
                                 // ... if needed
07        cCLOAD(0); NOP;         // acc <= reduceAdd
08        cNOP;      CLOAD;       // acc[i] <= acc;
09        cLOAD(5);  STORE(1);   // acc <= ixCounter; mem[i][1]<= acc[i]
10        cBRNZDEC(6); ENDWHERE; // test end of loop;
```

Lines 01 to 07 provide in controller's accumulator the scalar product of the vector with a line of the matrix. Because the latency of the reduction network for addition is $1+0.5logP$, the above example is for an array of 256 cells. The minimum length of the loop is of 9 cycles, for $P = 16$, because in the lines 02 to 04, in array is selected the cell which will receive the currently computed scalar product. The time for this loop is: $7 + 0.5logP$.

We were looking for an improvement to reduce this $7 + 0.5logP$ to the smallest possible constant in order to obtain:

$$T_{mMMM} \in O(N^2)$$

instead of the current:

$$T_{mMMM} \in O(N^2 logP)$$

(where *mMMM* stands for modified matrix-matrix multiplication)

This is what we obtained with the inclusion of the shift register in the design. We were able to reduce the previous loop sequence to only two lines:

```
LB(6); cCPUSHL(0);  RILOAD(63); //push redSum; load next matrix line
       cBRNZDEC(6); MULT(0);    //test end; multiply line with vector
```

And so, the execution time for matrix-matrix multiplication using the previous matrix-vector multiplication becomes:

$$T_{mMMM} = (2N + 0.5log_2P + c_1)N + N^2 + c_2 \in O(N^2)$$

Thus, both, the magnitude order and the constant is small, because, for big N, $T_{mMMM} \rightarrow 3N^2$.

6. Concluding Remarks

The APSP matrix–multiplication based algorithm on a P-processor hypercube architecture is evaluated as working in $O(N^2logP)$ cycles [11], where N is the number of vertices and $N \leq P$. Our architecture provides the same theoretical time performance, but the advantages we offer is that our engine has the size in $O(P)$ compared with a hypercube organization with a size in $O(PlogP)$.

Another advantage of our solution is that the cells in our engine are *execution units*, not *processing units* like in the hypercube engines. The program in a distributed hypercube architecture is replicated P times in each of the P processing units, while in our approach it is stored only in the Controller's program memory.

The last but not the least advantage of our solution is that the hypercube architecture supposes data multiplies many times in the processing cells' array, while in our approach data is not multiplied in the array of the execution cells. Also, the actual energy performance of our architecture is 7 pJ for 32-bit integer operation, compared with the ~ 150 pJ per operation of the current many-cores.

Acknowledgment

This work has been funded by the Sectoral Operational Program Human Resources Development 2007-2013 of the Ministry of European Funds through the Financial Agreement POSDRU/159/1.5/S/132397.

R E F E R E N C E S

[1]. *M. Hilbert, P. López*, "The world's technological capacity to store, communicate, and compute information", Science vol. 332, no. 6025, April 2011, pp. 60–65.

[2]. *G. Revesz*, "Parallel Graph-Reduction With A Shared Memory Multiprocessor System", IEEE Computer Languages, New Orleans, LA, March 1990, pp. 33-38.

[3]. *M. Yasugi, T. Hiraishi, S. Umatani and T. Yuasa*, "Dynamic Graph Traversals for Concurrent Rewriting using Work-Stealing Frameworks for Multi-core Platforms", IEEE Conference on Parallel and Distributed Systems (ICPADS), 16th edition, Dec 2010, pp. 406 – 414.

[4]. *M. Cosulschi, A. Cuzzocrea and R. De Virgilio* "Implementing BFS-based Traversals of RDF Graphs over MapReduce Efficiently", IEEE Conference on Cluster, Cloud and Grid Computing (CCGrid), Delft, May 2013, pp. 569 – 574.

[5]. *Q. Lianghong, F. Lei and L. Jianhua*, "Implementing Quasi-Parallel Breadth-First Search in MapReduce for Large-Scale Social Network Mining", IEEE Conference on Computational Aspects of Social Networks (CASoN), Fifth International Conference, 2013, pp. 7 – 14.

[6]. *A. Papagiannis, D.S. Nikolopoulos*, "MapReduce for the Single-Chip Cloud Architecture" ACACES Journal - Seventh International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems, Fiuggi, Italy, 2011.

[7]. *A. Tripathy, A. Patra, S. Mohan and R. Mahapatra*, "Distributed Collaborative Filtering on a Single Chip Cloud Computer", IEEE Conference on Cloud Engineering (IC2E), 2013, pp. 140 - 145.

[8]. *G. Stefan*, "One-Chip TeraArchitecture", Proceedings of the 8th Applications and Principles of Information Science Conference, Okinawa, Japan, 2009.

[9]. *C. Bira, R. Hobincu, L. Petrica, V. Codreanu, S.Cotofana*, "Energy - Efficient Computation of L1 and L2 Norms on a FPGA SIMD Accelerator, with Applications to Visual Search", Proceedings of the 18th International Conference on Computers (part of CSCC '14), Advances in Information Science and Applications – vol. II, Santorini, Greece, 2014, pp. 432-437.

[10]. *A.Gheolbanoiu, D.Mocanu, R.Hobincu, L.Petrica*, "Cellular Automaton pRNG with a Global Loop for Non-Uniform Rule Control", Proceedings of the 18th International Conference on Computers (part of CSCC '14), Advances in Information Science and Applications – vol. II, Santorini, Greece, 2014, pp. 415-420.

[11]. *V. Kumar, A. Grama, A. Gupta and G. Karypis*, Introduction to Parallel Computing: Design and Analysis of Algorithms, The Benjamin/Cummings Publishing Company, 1994.

[12]. *K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, K.A. Yelick*, "The landscape of parallel computing research: A view from Berkeley", Technical Report No. UCB/EECS-2006-183, December 18, 2006. http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf