International Journal on Recent and Innovation Trends in Computing and Communication ISSN: 2321-8169
Volume: 4 Issue: 4 76 - 81

Breadth-First Search on a MapReduce One-Chip System

Voichita Dragomir
Electronic Devices, Circuits and Architectures
Politehnica University of Bucharest
Bucharest, Romania
voichita.dragomir@upb.ro

***Abstract*—** An implementation of a newly developed parallel graph traversal algorithm on a new *one-chip many-core structure with a MapReduce architecture* is presented. The generic structure's main features and performances are described. The developed algorithm uses the representation of the graph as a matrix and the new MapReduce structure performs best on matrix-vector operations so, the algorithm considers both, dense and sparse matrix cases. A *Verilog* based simulator is used for evaluation. The main outcome of the presented research is that our *MapReduce* architecture (with P execution units and the size in $O(P)$) has the same theoretical time performance: $O(NlogN)$ for $P = N = |V| =$ *number of vertices in the graph*, as the hypercube architecture (having P processors and the size in $O(PlogP)$). Also, the actual energy performance of our architecture is 7 pJ for 32-bit integer operation, compared with the ~150pJ per operation of the current many-cores.

Keywords- *parallel computing; mapReduce; many core; graph traversal; breadth first search; parallel algorithm*

*****

I. INTRODUCTION *(HEADING 1)*

Nowadays, there is a growing need for more divers and complex functions, resulting in the need for faster and bigger computer and communication networks. Parallelism and how to do thing faster, cheaper and more efficient are on every engineer's mind.

One of the needed function is the ability to operate with complex representations. The most complex representation is the graph representation and that is why we chose to focus on graphs, meaning graph algorithms and parallel accelerators.

Why are graphs important? Because many real life problems, like computer networks, navigation systems, social networks and functions, etc., can be expressed in terms of graphs and can be solved using standard graph algorithms. There is a constant growth of applications like this nowadays, resulting in the growing need for processing larger and larger graphs with more complex functions, hence the need for developing new parallel algorithms and structures that can handle all this.

So graph traversal/ searching algorithms can be applied to solve a multitude of real life problems: Computer Networks: peer to peer applications need to locate files requested by users. This is achieved by applying breadth-first search algorithm - BFS- on one's computer network. GPS Navigation systems: navigation systems use shortest path algorithms. They take your location as the source node and your destination as the destination node on the graph. (A city can be represented as a graph by taking landmarks as nodes and the roads as edges.) Using these algorithms, the shortest route is generated, to give directions for real time navigation. Social networks: treat each user as a node on the graph and two nodes are connected if they are each other's friends.

The Breadth-First-Search (BFS) and Depth-First-Search (DFS) algorithm are fundamentally the same and consist of visiting all the vertices and edges in a graph in a particular manner, updating and checking their values along the way (BFS works with queue and DFS with stack), each with it's own optimal application.

For example, BFS is very good for finding the shortest path in a graph, while DFS algorithm is usually better from memory perspective (no need to store all pointers) it has a lower space complexity and it is good for detecting cycle in a graph, topological sorting or finding strongly connected components of a graph. But if the search domain is infinite, depth-first-search may not find a solution, even if a finite solution does exist. BFS algorithm can do that.

Taking all this into consideration, and having explained why graph traversal is important, we set out to present a different approach from what has been done so far. We are developing a parallel breadth-first search graph traversal algorithm and we will have it working on a newly developed structure, a one-chip many-core MapReduce architecture.

So far, BFS algorithms have been implemented on multi-core processors, which are still based on the shared-memory model. They are limited in the number of cores, the memory size and they are non-scalable for big data size [9], [13]. Another existing implementation of BFS algorithm is done in cloud, where the MapReduce approach is limited by the latency introduced by the communication network [3], [7].

Both of these solutions have limitations:

a) Multi-core architecture with shared external memory:

- the cores compete for the shared external memory (known as the bottleneck effect).
- limited in the number of cores.
- limited in memory capacity.
- the solution does not scale well

b) Cloud MapReduce architecture with distributed memory which implies multithreading (dividing the problem into threads and processing them simultaneous on multiple cores) has a latency in communicating between the machines, meaning a significant increase in energy and time use.

What is new in our approach is that we are going to achieve the parallel BFS algorithm on a one-chip many-core structure not on multi-core or distributed computing. These techniques have limitations due to the communication and power issues. For example, if the interconnection network used is a hypercube, then the size of the entire system is in O(PlogP) with a latency in communication in O(log P).

There are other one-chip MapReduce approaches. For example, the Intel SCC family. In [8] and [12] two different MapReduce applications are presented. The use of this general purpose array of processors has a much slower response than ours, because it has no more than 48 cores (which are much too complex for solving this kind of problem) and the MapReduce functionality is implemented in software, not hardware, as in our case.

The architecture we work on is different than these existing ones, it is a one-chip many-core MapReduce architecture. It is presented in the next section. In sections three and four we preset the parallel version for breadth-first search graph traversal algorithm, developed for this new, one-chip many-core MapReduce architecture. This system performs best on matrix-vector operations. Therefore, we designed an algorithm based on both dense and sparse matrix cases. In order 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.

II. The Generic One-Chip MapReduce Architecture

The research presented in this paper is part of a larger project having as the main goal to improve, a generic new architecture, using the 13 "dwarfs" (a collection of algorithm families that are important to parallel computing) emphasized in the Berkeley research report on parallel computation [1]. The "dwarf" considered in this paper is Graph traversal.

A. The Structure (Heading 2)

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

Linear Array of Cells : an array of hundreds of thousands cells containing execution units and local memory of few KB (kilobytes).

Controller : a processing unit which issues in each cycle an instruction and various data, if needed, to be distributed in the array of cells.

Distr : is a log-depth tree structure used to distribute instructions and data in the array.

Reduce : is a log-depth tree structure used to compute few reduction functions (add, min, max, ...) which provides for the controller a scalar starting from a vector.

Trans : is a log-depth two-direction tree structure used to exchange data between the internal distributed memory and the external memory, Memory.

Scan : is a log-depth two-direction tree structure used to close a combinatorial loop over the linear array of cells.

Host : is a general purpose computer whose complex program is accelerated by the MapReduce structure formed by Linear Array of Cells & Reduce.

Memory : is the external memory in the range of 1 GB.

Int : id the system interface (usually PCIe).

Interconnection Fabric : is a multi-point network used to transfer data between the chip and the external resources: Memory & Int.

This structure has a very short response time because of the controller and the log-depth Reduction Module that works for many (thousands) cores. The main features of this structure are: high degree of parallelism, the cores are small and simple, the local memory is big enough for data mining applications [11].

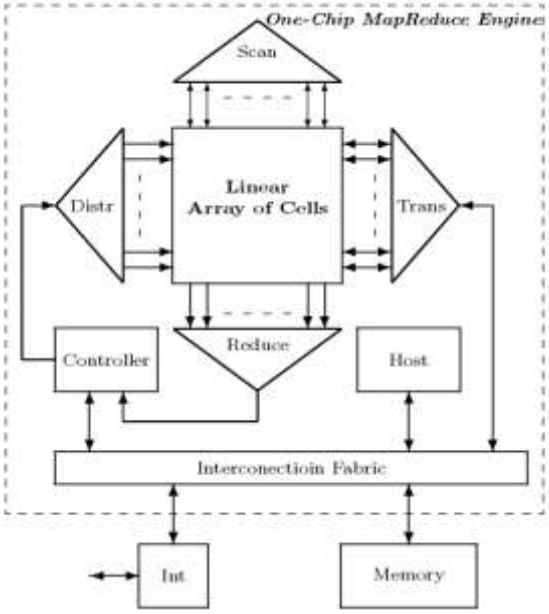

Figure 1. The one-chip many-core MapReduce structure

The structure supports two data domains:

- the *S* domain: a linear array of scalars from the Memory module (see Fig. 1)
$$S = < s_0, s_1, \ldots , s_{n-1} >$$
- the *V* domain: a linear array of vectors distributed in the local memory of the cells in Linear Array of Cells (see Fig. 1)
$$V = < v_0, v_1, \ldots , v_{m-1}, ix >$$
where:
$$v_i = < x_{i\,0}, \ldots , x_{i\,p-1} >$$
for $i = 0, \ldots m$ -1 which forms a two-dimension array containing m p-scalar vectors distributed along the linear array of cells.

There are also tree special vectors:

$$indexVector =< 0, 1, \ldots , p - 1 >$$
$$activeVector =< a_0, a_1, \ldots , a_{p-1} >$$
$$accVector =< acc_0, acc_1, \ldots . acc_{p-1} >$$

used, by turn, to *index* the position of each cell in the linear array of cells, to *activate* the cells (if the *i*-th component of the *activeVector* is 1, than the *i*-th cell is active), and to form a distributed *accumulator* along the one-dimension array of cells.

The previously described structure was implemented in a few versions. The last of the three implemented version, issued in 2008 in 65nm standard cells technology [10], provides the following performances: 100 GOPS/Watt, 5 GOPS/mm^2, while the current sequential engines (x86 architecture, for example) have, in the same technology: ~1GOPS/Watt, ~0,25 GOPS/mm^2 (GOPS stands for Giga Operations Per Second).

The size of the previously described 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(log p).

B. *Instruction Set Architecture*

Instruction Set Architecture defines the operations performed over the two data domains: *S* domain and *V* domain. A short description follows. Because the structure of the MapReduce generic engine consists of two programmable parts, the Controller and the Linear Array of Cells, the instruction set architecture, $ISA_{MapReduce}$, is a dual one:

$$ISA_{MapReduce} = (ISA_{controller} \times ISA_{array})$$

In each clock cycle, a pair of instructions is read from the program memory of the Controller: one from $ISA_{controller}$, to be executed by Controller, and another from ISA_{array}, to be executed by Linear Array of Cells.

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

`activate` : all the cells of Linear Array of Cells are activated.
`where` : maintains active only the cells where the condition `cond` is fulfilled; example: `where(zero)` maintains active only the active cells where the accumulator is zero (it corresponds to the `if(cond)` instruction form a standard sequential subset of control flow instructions).
`elsewhere` : activates the cells inactivated by the associated `where(cond)` instruction (it corresponds to the else instruction form a standard sequential subset of control flow instructions).
`endwhere` : restores the activations existed before the previous `where(cond)` instruction (it corresponds to the `endif` instruction).

1) *The Instruction Structure (Heading 3)*

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

```
mapRedInstr[31:0] =
{controlInstr[15:0], arrayInstr[15:0]} =
      {{cInstr[4:0], cOperand[2:0], cValue[7:0]},
       {aInstr[4:0], aOperand[2:0], aValue[7:0]}}
```

where:

`cInstr[4:0]/aInstr[4:0]`: codes the instruction for Controller / Linear Array of Cells;
`cOperand[2:0]/aOperand[2:0]` : codes the second operand used in instruction for Controller/Linear Array of Cells;
`cValue[7:0]/aValue[7:0]` : is mainly the immediate value or the address for Controller/Linear Array of Cells;

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

```
val : op = {{(n-8){value[7]}}, value[7:0]}
mab : op = mem[value]
mrl : op = mem[value + addr]
mri : op = mem[value + addr]; addr <= value + addr;
cop : op = coop ;
mac : op = mem[coop];
mrc : 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.
`redMin` : the minimum value of the accumulators from the active cells.
`redMax` : the maximum value of the accumulators from the active cells.
`redBool` : the sum of the active bit from the active cells.

2) *The Assembler Language (Heading 3)*

The assembly language provides a sequence of lines each containing an instruction for Controller (with the prefix c) and another for Array. Some of the line are labeled – LB(n) – where *n* is a positive integer used to specify the jumps in the flow of instructions.

Example 1. The program which provides in the controllers accumulator the sum of indexes is:

```
cNOP; ACTIVATE;  // activate all cells
cNOP; IXLOAD;    // acc[i] <= i
cCLOAD(0); NOP;  // acc <= sum of indexes
```

Example 2. Let us show how the spatial selection works. Initially, we have:

```
indexVector = <0 1 2 ... p-2 p-1>
activeVector = <x x x ... x x >
```

The following sequence of instructions will provide:

```
cNOP; ACTIVATE;   // activeVector <= <1 1 1 1...>
cNOP; IXLOAD;     // acc[i] <= i
cNOP; VSUB(3);    // acc[i] <= acc[i]-3;
                     // cryVector <= <1 1 1 0...>
cNOP; WHERECARRY; // actVector <= <1 1 1 0...>
cNOP; VADD(2);    // acc[i] <= acc[i]+ 2;
cNOP; WHEREZERO;  // activeVector <= <0 1 0 0...>
cNOP; ENDWHERE;   // activeVector <= <1 1 1 0...>
cNOP; ENDWHERE;   // activeVector <= <1 1 1 1...>
```

The first `where` instruction lets active only the first three cell. The second `where` adds a new restriction: only the cell with index 1 remains active. The first `endwhere` restore the activity of the first three cells, while the second `endwhere` reactivates all the cells.

This one-chip MapReduce architecture is used as an accelerator in various application fields: video [2], encryption, data mining. Also, nonstandard versions of this architecture are used for generating efficiently pseudo-random number sequences [5].

The system we work with is a many-core chip with a MapReduce architecture that performs best on matrix- vector operations. Therefore, we designed an algorithm based on the representation of the graph as a matrix and so we have to cover both dense and sparse matrix cases.

Furthermore, the steps we took were: choosing the BFS function, understanding how it works and what the result should be, developing the parallel algorithm having the same result on our MapReduce structure, testing and comparing it with the algorithm implemented on a Hypercube (the most efficient and used parallel structure nowadays).

III. BREADTH-FIRST SEARCH ALGORITHM FOR DENSE MATRIX REPRESENTATION OF GRAPHS

Given a graph (G) and a starting vertex (s), a parallel breadth first search algorithm explores all edges in the graph to find all the vertices in (G) for which there is a path from (s). It finds *all* the vertices that are at distance (k) from (s) before it finds *any* vertices that are a distance (k+1). One good way to visualize what the breadth first search algorithm does is to imagine that it is building a tree, one level of the tree at a time. A breadth first search adds all children of the starting vertex before it begins to discover any of the grandchildren.

A. The Algorithm (Heading 2)

Let us take as example [4] an 8-vertex graph, $V = \{0,1, ...,7\}$ shown in Fig. 2, with the following adjacency matrix:

	0	1	2	3	4	5	6	7
v0	0	1	1	0	0	0	0	0
v1	1	0	0	1	1	0	0	0
v2	1	0	0	0	0	1	0	1
v3	0	1	0	0	1	1	0	1
v4	0	1	0	1	0	0	1	0
v5	0	0	1	1	0	0	1	0
v6	0	0	0	0	1	1	0	1
v7	0	0	0	1	0	0	1	0

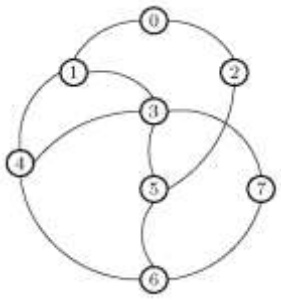

Figure 2. The 8-vertex graph used as example

The result of applying the BFS is the tree described by the following two vectors:

```
d = 0 1 2 3 4 5 6 7  // destination vector
s = 1 3 5 8 3 3 4 3  // source vector
```

where each pair of vertexes stands for an edge in the resulting tree. The vertex 8 is a virtual one, it is used to pinpoint the root of the tree.

The algorithm desined for our MapReduce engine is:

```
N = |V|;              // N = 8
acc <= initial_root; // acc = 3
s <= N+1 N+1 ...;     // s = 9 9 9 9 9 9 9 9: no source
b <= 0 0 ...;         // b = 0 0 0 0 0 0 0 0: branches
r <= 0 0 ...;         // r = 0 0 0 0 0 0 0 0: roots
where (d=acc)
     r <= 1;          // r = 0 0 0 1 0 0 0 0: initial root
     s <= N;          // s = 9 9 9 8 9 9 9 9: points initial root as 8
endWhere
while (redMax(s)>N) {
          while (redSum(r)>0) {
          where ((r=1)&(first))
               acc(first) <= ix;
               acc <= redOr;      // acc = 3
               r <= 0;            // r = 0 0 0 0 0 0 0 0
          endWhere

          where ((v(acc)=1)&(s=9))
              s <= acc;             // s = 9 3 9 8 3 3 9 3
              b <= 1 ;              // b = 0 1 0 0 1 1 0 1
          endWhere
   }
   r <= b;
   b <= 0 0 ...;
}
```

B. The Program

The program is presented in Appendix (as an example of how our engine is programmed at the lowest level).

The maximum execution time is:

$$T_{BFS\_DENSE} = (N-1)logP + 33N - 17 \quad \in O(NlogP)$$

with $N \leq P$, where P is the number of cells and N the number of vertices in the graph.

The actual time depends on the final form of the tree. The maximum time is obtained when each vertex is only followed by one other vertex.

C. The Test Program and the Results

Example 3. Let us consider the graph from Fig. 2. The initial state of the vector memory contains the adjacency matrix as follows:

```
vect[8]  = 0 1 1 0 0 0 0 0 x ...
vect[9]  = 1 0 0 1 1 0 0 0 x ...
vect[10] = 1 0 0 0 0 1 0 0 x ...
vect[11] = 0 1 0 0 1 1 0 1 x ...
vect[12] = 0 1 0 1 0 0 1 0 x ...
vect[13] = 0 0 1 1 0 0 1 0 x ...
vect[14] = 0 0 0 0 1 1 0 1 x ...
vect[15] = 0 0 0 1 0 0 1 0 x ...
```

The TEST program is listed in Appendix. The result is stored in the following two vectors:

```
vect[0] = 0 1 2 3 4 5 6 7 x ... = destination
vect[1] = 1 3 5 8 3 3 4 3 x ... = source
```

whose content is interpreted as the adjacency list:

```
((1 0)(3 1)(5 2)(x 3)(3 4)(3 5)(4 6)(3 7))
```

where the source x points to the root of the tree.
The execution time is: $T_{BFS\_DENSE}(8) = 223$ cycles < 275 which is the maximal theoretical prediction.

The resulting tree is shown in Figure 4:

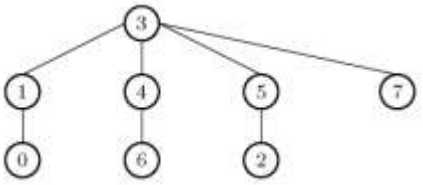

Figure 3. The result of applying BFS on the graph from Fig.2

IV. BREADTH-FIRST-SEARCH ALGORITHM FOR SPARSE MATRIX REPRESENTATION OF GRAPHS

The representation of the sparse non-weighted adjacency matrix consists of two vectors, one for lines, another for columns.

Example 4. Let us revisit the example form [4], where V = {0,1, ...,7} and |E| = 11. The length of the two vectors, `l` and `c`, is |E| :

	0	1	2	3	4	5	6	7
v0	0	1	1	0	0	0	0	0
v1	1	0	0	1	1	0	0	0
v2	1	0	0	0	0	1	0	1
v3	0	1	0	0	1	1	0	1
v4	0	1	0	1	0	0	1	0
v5	0	0	1	1	0	0	1	0
v6	0	0	0	0	1	1	0	1
v7	0	0	0	1	0	0	1	0

```
l = 0 0 1 1 2 3 3 3 4 5 6  // line
c = 1 2 3 4 5 4 5 7 6 6 7  // column
```

A. *The Algorithm*

The algorithm is similar to the algorithm for representation using dense matrix. The only difference refers to the search of the vertexes pointed by the currently considered vertex, i.e., the sequence of steps:

```
where (v(acc)=1)
   where (s=9)
        s <= acc;   // s = 9 3 9 8 3 3 9 3
        b <= 1  ;   // b = 0 1 0 0 1 1 0 1
   endWhere
endWhere
```

which will be substituted with a more complex one, as follows:

```
N = |V|;                // (N = 8)
acc <= initial_root;    // (acc = 3)
s <= N+1 N+1 ...;       // s = 9 9 9 9 9 9 9 9: no source
b <= 0 0 ...;           // b = 0 0 0 0 0 0 0 0: branches
r <= 0 0 ...;           // r = 0 0 0 0 0 0 0 0: roots
where (d=acc) r <= 1;   // r = 0 0 0 1 0 0 0 0: initial root
              s <= N;   // s = 9 9 9 8 9 9 9 9:
endWhere
while (redMax(s)>N) {                   // while there is 9 in s
    while (redSum(r)>0) {               // while roots unexplored
        where ((r=1)&(first))
              acc(first) <= ix;
              acc <= redOr;             // acc = 3
              r <= 0;                   // r = 0 0 0 0 0 0 0 0
        endWhere
        cr <= acc;                      // currentRoot <= acc
        while (redFlag(where (l=acc)) {  // while l has current_root
            where (first)
                 acc <= c;
                 l <= 9;
            endwhere
            where ((d=acc))&(s=9))
                 s <= acc;              // because |V| =< |E|
                 b <= 1 ;
            endwhere
        endwhere
       }
       acc <= cr;
       while (redFlag(where (c=acc)) {  // while c has current_root
            where (first)
                 acc <= l;
                 c <= 9;
            endwhere
            where ((d=acc))&(s=9))      // because |V| =< |E|
                 s <= acc;
                 b <= 1 ;
            endwhere
      endwhere
     }
   }
   r <= b;
   b <= 0 0 ...;
}
```

The search in the dense matrix representation is substituted with the search in the two vectors of the sparse matrix representation, where each edge is represented only once.

B. *The Program*

The program for sparse matrix representation, written in the assembler language presented in chapter II.*B.2*, was used to determine the maximum execution time (the maximum execution time depends on the actual shape of the graph):

$$T_{BFS\_SPARSE\_max} = 3.5(N-1)logP + 85N - 66 \quad \in O(NlogP)$$

with $N \leq P$, where P is the number of cells and N the number of vertices in the graph.

C. *The Test Program and the Results*

Let us revisit the same example as for dense matrix representation. The TEST program provided the following result, stored in the two vectors:

```
vect[0] = 0 1 2 3 4 5 6 7 0 ... = destination
vect[1] = 1 3 5 8 3 3 4 3 0 ... = source
```

which corresponds to the following list of pairs of vertexes:

```
((1 0)(3 1)(5 2)(x 3)(3 4)(3 5)(4 6)(3 7))
```

where vertex 3 is the root of the resulting tree. The measured running time is $T_{BFS\_SM}(8) = 769$. It is smaller then the maximal theoretical prediction, 837, because, in the case we considered, the depth of the resulting tree is not maximal.

V. Concluding Remarks

BFS algorithms on hypercube architecture are evaluated (see [6]) as working, for both sparse and dense matrix representation, in $O(NlogP)$, where: N is the number of vertices and P the number of processors. Our architecture provides performances in the same magnitude order, but the advantages we offer are: our engine size is smaller, it is in $O(P)$ compared with a hypercube organization's size which is in $O(PlogP)$. Another advantage of our solution is that the cells in our engine are *execution units*, while in the hypercube engines, evaluated in [6], the cells are *processing units*. 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 once in the Controller's program memory.

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.

Appendix

A. *The Program for BFS_Dense Matrix Representation*

The program is stored in the file named `03_BFS.v`:

```
     cNOP;       IXLOAD;
     cNOP;       STORE(0); // dest : 0 1 2 3 4 5 6 7
     cNOP;       VLOAD(9);
     cNOP;       STORE(1); // source : 9 9 9 9 9 9 9 9
     cNOP;       VLOAD(0);
     cNOP;       STORE(2); // branches : 0 0 0 0 0 0 0 0
     cNOP;       STORE(3); // roots: 0 0 0 0 0 0 0 0
     cVLOAD(3); LOAD(0);  // acc <= 3 (the initial vertex);
                          // acc[i] <= dest
     cNOP;       CSUB;
     cNOP;       WHEREZERO;
     cNOP;       VLOAD(1);
     cLOAD(0);   STORE(3);
     cNOP;       CLOAD;
     cNOP;       STORE(1);
     cNOP;       ENDWHERE;
     cNOP;       NOP;

LB(2); cNOP;    LOAD(1);  // acc[i] <= source[i]
     cCLOAD(2); NOP;      // acc <= redMax
     cSUB(0);   NOP;      // acc <= redMax - N
     cBRZ(1);   NOP;      // redMax = N, jump to end

// the inner loop

LB(4); cNOP;    LOAD(3);   // acc[i] <= roots
     cCLOAD(0); NOP;
     cBRZ(3);   NOP;       // test end inner loop
     cNOP;      VSUB(1);   // acc[i] <= roots[i] - 1
     cNOP;      WHEREZERO; // where roots = 1
     cNOP;      NOP;
     cNOP;      WHEREFIRST;  // where first (roots = 1)
     cNOP;      IXLOAD;      // acc[i] <= index
     cCLOAD(0); VLOAD(0);    // acc <= redSum;
                             // acc[i] <= 0
     cVADD(8); STORE(3);     // acc <= acc + 8;
                             // roots[i] <= 0
     cNOP;     ENDWHERE;
     cNOP;     NOP;
     cNOP;     ENDWHERE;
     cNOP;     CALOAD;       // acc[i] <= v[acc]
     cNOP;     VSUB(1);
     cNOP;     WHEREZERO;
```

```
    cNOP;     LOAD(1);      // acc[i] <= source
    cNOP;     VSUB(9);      // acc[i] <= source - 9
    cVSUB(8); WHEREZERO;
    cNOP;     CLOAD;        // acc[i] <= acc
    cNOP;     STORE(1);     // source <= acc[i]
    cNOP;     VLOAD(1);     // acc[i] <= 1
    cNOP;     STORE(2);     // branches <= 1
    cNOP;     ENDWHERE;
    cNOP;     NOP;
    cNOP;     ENDWHERE;
    cJMP(4);  NOP;

// end inner loop

LB(3); cNOP;  LOAD(2);     // acc[i] <= branches[i]
    cNOP;    STORE(3);     // roots[i] <= branches[i]
    cNOP;    VLOAD(0);     // acc[i] <= 0
    cJMP(2); STORE(2);     // branches[i] <= 0

// LB(1); cSTOP; NOP;
          cHALT; NOP;
```

B. Test Program for BFS_Dense Matrix Representation

The TEST program for `03_BFS.v` is:

```
    cNOP;     ACTIVATE;
    cVLOAD(8);IXLOAD;     // acc <= N; acc[i] <= index
    cSTORE(0);CSUB;       // mem[0]<= N; acc[i]<= index - N
    cNOP;     WHERECARRY; // select only the first N cells

    `include "03_matrixLoad.v"

    cSTART;   NOP;

    `include "03_BFS.v"

LB(1); cSTOP; NOP;
       cHALT; NOP;
```

The program `03_matrixLoad.v`, which loads the adjacency matrix, is:

```
    cVPUSHL(0); NOP;
    cVPUSHL(0); NOP;
    cVPUSHL(0); NOP;
    cVPUSHL(0); NOP;
    cVPUSHL(0); NOP;
    cVPUSHL(1); NOP;
    cVPUSHL(1); NOP;
    cVPUSHL(0); NOP;
    cNOP;       SRLOAD;
    cNOP;       STORE(8); // v0
    cVPUSHL(0); NOP;
    cVPUSHL(0); NOP;
    cVPUSHL(0); NOP;
    cVPUSHL(1); NOP;
    cVPUSHL(1); NOP;
    cVPUSHL(0); NOP;
    cVPUSHL(0); NOP;
    cVPUSHL(1); NOP;
    cNOP;       SRLOAD;
    cNOP;       STORE(9); // v1
    cVPUSHL(0); NOP;
    cVPUSHL(0); NOP;
    cVPUSHL(1); NOP;
    cVPUSHL(0); NOP;
    cVPUSHL(0); NOP;
    cVPUSHL(0); NOP;
    cVPUSHL(0); NOP;
    cVPUSHL(1); NOP;
    cNOP;       SRLOAD;
    cNOP;       STORE(10); // v2
    cVPUSHL(1); NOP;
    cVPUSHL(0); NOP;
    cVPUSHL(1); NOP;
    cVPUSHL(1); NOP;
    cVPUSHL(0); NOP;
    cVPUSHL(0); NOP;
    cVPUSHL(1); NOP;
    cNOP;       SRLOAD;
    cNOP;       STORE(11);  // v3
    cVPUSHL(0); NOP;
    cVPUSHL(1); NOP;

    cVPUSHL(0); NOP;
```

REFERENCES

[1] K. Asanovic, R. Bodik, B.C. Catanzaro, et al., “The landscape of parallel computing research: A view from Berkeley”, Technical Report No. UCB/EECS-2006-183, December 18, 2006.

[2] 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.

[3] 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, 2013, pp. 569-574 .

[4] V. Dragomir, “Graph Traversal on One-Chip MapReduce Architecture”, Proceedings of the 18th International Conference on Computers (part of CSCC 14), Advances in Information Science and Applications, Vol. II, Santorini, Greece, 2014, pp. 559-563 .

[5] 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.

[6] V. Kumar, A. Grama, A. Gupta, G. Karypis, Introduction to Parallel Computing. Design and Analysis of Algorithms, The Benjamin/Cummings Pub. Comp. Inc., 1994.

[7] Q. Lianghong, F. Lei, L. Jianhua, “Implementing QuasiParallel Breadth-First Search in MapReduce for LargeScale Social Network Mining”, IEEE Conference on Computational Aspects of Social Networks (CASoN), Fifth International Conference, pp. 714 ,2013.

[8] A. Papagiannis, D.S. Nikolopoulos, “MapReduce for the Single-Chip- Cloud Architecture”, Institute of Computer Science (ICS), Foundation for Research and Technology Hellas (FORTH), Greece, 2011.

[9] G. Revesz, “Parallel Graph-Reduction With A Shared Memory Multiprocessor System”, IEEE Computer Languages, New Orleans, LA, 1990, pp. 33-38.

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

[11] G.M. Stefan, M. Malita, “Can One-Chip Parallel Computing Be Liberated from Ad Hoc Solutions? A Computation Model Based Approach and Its Implementation”, Proceedings of the 18th International Conference on Computers (part of CSCC 14), Advances in Information Science and Applications Vol. II, Santorini Island, Greece, 2014, pp. 582-597.

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

[13] M. Yasugi, T. Hiraishi, S. Umatani, 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, 2010, pp. 406-414.