ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY
Volume **18**, Number 2, 2015, 126–143

Minimum Spanning Tree Algorithm on MapReduce One-Chip Architecture

Voichiţa MAICAN

DCAE Department, Politehnica University of Bucharest, Romania
E-mail: `voichita.dragomir@upb.ro`

Abstract. A parallel algorithm for minimum spanning tree and its implementation on a *one-chip many-core structure with a MapReduce architecture* is presented. The generic structure's main features and performances are described, but also new general purpose features added for upgrading the existing generic structure in order to perform better in running the proposed algorithms. As the developed algorithm uses the representation of the graph as a matrix, both , dense and sparse cases are considered. A Verilog based simulator is used for evaluation. The main outcome of the presented research is that, compared with the hyper-cube architecture (having p processors and the size in $O(p\,log\,p)$), our MapReduce architecture (with p execution units and the size in $O(p)$) has the same theoretical time performance: $O(NlogN)$ for $p = N = |V|$. Also, the actual energy performance of our architecture is $7\,pJ$ for 32-bit integer operation, compared with the $\sim 150\,pJ$ per operation of the current many-cores.

Key-words: parallel computing; MapReduce; many core; minimum spanning tree; parallel algorithm

1. Introduction

Processing large graphs is becoming increasingly important nowadays, because a graph can be used to represent a large number of real life problems such as road networks, computer networks and social networks. So graph algorithms can be applied to solve a multitude of real life problems: Computer Networks - peer to peer applications need to locate a file that the client is requesting. GPS Navigation systems - navigation systems, which can give directions to reach from one place to another, 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). Facebook - treats each user profile as a node on the graph and two nodes are said to be connected if they are each other's friends.

The graph algorithms are fundamentally the same and consists of visiting all of the vertices and edges in the graph in a particular manner, updating and checking their values along the way, each with it's own optimal application.

Minimum Spanning Tree is one of these graph algorithms with lots of applications in the design of networks, including computer networks, telecommunications networks, transportation networks, water supply networks, and electrical grids. Some practical applications based on minimal spanning trees are: taxonomy, cluster analysis, constructing trees for broadcasting in computer networks (on Ethernet networks this is accomplished by means of the Spanning tree protocol), image registration and segmentation, curvilinear feature extraction in computer vision, handwriting recognition of mathematical expressions, circuit design (implementing efficient multiple constant multiplications, as used in finite impulse response filters), regionalization of socio-geographic areas (the grouping of areas into homogeneous, contiguous regions).

So far, parallel algorithms for minimum spanning tree 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) or distributed computing where the MapReduce approach is limited by the latency introduced by the communication network. These two main existing parallel structures are shown in Fig. 1.

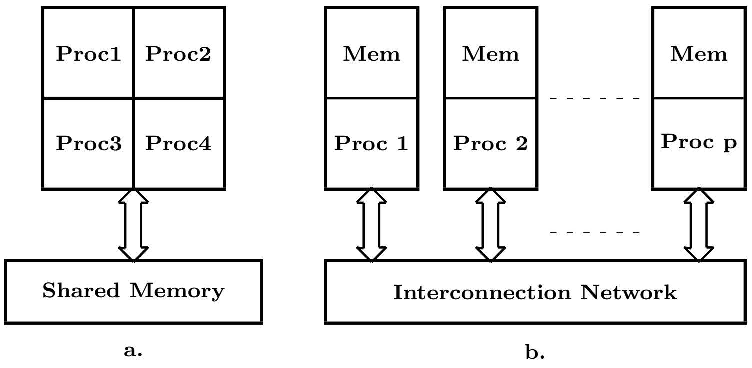

Fig. 1. Main existing parallel structures: **a.** Multi-core architecture; **b.** Distributed architecture. The most efficient interconnection network is a hyper-cube network.

The problem is that both of the existing structures have limitations:

1. the cores compete for the shared resource (the external memory) known as the bottleneck effect in multi-core architectures
2. the individual cores have access to their own memory, in distributed architectures, but there is a latency in communicating between the machines, so there is a significant increase in energy and time use.

What is new in our approach is that we are going to use a *one chip many-core structure* not on multi-core or distributed computing, which implies multi-threading (dividing the problem into threads and processing them simultaneous on multiple cores). This technique has its limitations due to the communication and power issues. For example, if the interconnection network used is a hyper-cube, then the size of the entire system belongs to $O(p \, log \, p)$ with a latency in communication in $O(log \, p)$.

There are other one-chip MapReduce approaches. For example, the Intel SCC family. In [5] and [9] two different MapReduce applications are presented. The use of this Intel general purpose array of processors has a much slower response 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 versions for Prim's Minimum Spanning Tree algorithm [6] developed for this new, *one-chip many-core MapReduce architecture*. Then, in the fifth section, we present the new general features and characteristics added for upgrading this generic MapReduce architecture.

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 hyper-cube parallel computer.

2. Generic One-Chip MapReduce Architecture

The research presented in this paper is part of a larger project having as the main goal to tune, a generic new architecture, using the 13 "dwarfs" emphasized in the seminal Berkeley research report on parallel computation [1]. The "dwarf" considered in this paper is *Graph traversal*.

2.1. The Organization

Our *one-chip many-core MapReduce structure* is shown in Fig. 2. 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 [7].

The structure supports two data domains:

- the S domain: a linear array of scalars $S =< s_0, s_1, \dots, s_{n-1} >$ (see External Memory in Figure 2)
- the V domain: a linear array of vectors: the content of the vector buffer

$$V =< v_0, v_1, \dots, v_{m-1}, ix >$$

(see the Local Memory modules represented in Fig. 2), a two-dimension array containing m p-scalar ***horizontal vectors*** distributed along the linear array of cells:

$$\begin{gathered} v_0 =< x_{00}, \ldots, x_{0\ p-1} > \\ v_1 =< x_{10}, \ldots, x_{1\ p-1} > \\ \ldots \\ v_{m-1} =< x_{m-1\ 0}, \ldots, x_{m-1\ p-1} > \end{gathered}$$

and tree special vectors:

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

The vector *indexVector* is used to identify the position of each cell in the linear array of cells. The vector *activeVector* is used to activate the cells. If the i-th component of the *activeVector* is 1, than the i-th cell is active, *i.e.*, the current instruction is performed. The accumulator vector, $accVector$, is the vector associated to the distributed accumulator along the array of cells.

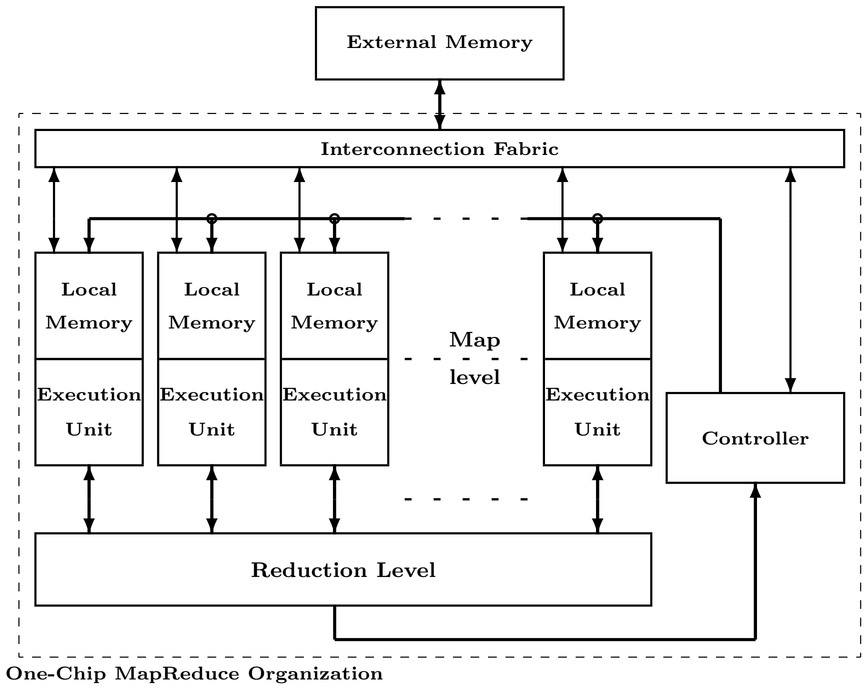

Fig. 2. The one-chip many-core MapReduce architecture.

The previously described structure was implemented in a few versions. The last, implemented in 2008, in 65 nm standard cells technology [8] provides the following performances:

- $100\,GOPS/Watt$ (Giga Operations Per Second / Watt)
- $5\,GOPS/mm^2$, while the current sequential engines (x86 architecture) have, in the same technology:
- $\sim 1\,GOPS/Watt$
- $\sim 0.25\,GOPS/mm^2$

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 $O(log\,p)$.

2.2. 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 Array –, the instruction set architecture, $ISA_{mapReduce}$, is a dual one:

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

where:

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

In each clock cycle from the program memory of the controller a pair of instructions is read: one from $ISA_{controller}$, to be executed by Controller, and another from ISA_{array} 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; 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 (zero)` instruction (it corresponds to the `endif` instruction form the $SS_{control}$ subset)

2.2.1. The Instruction Structure

The instruction format for the MapReduce engine allows issuing two instructions 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:
$\Sigma_0^p \, acc_i$

`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:
$\Sigma_0^p \, a_i$

2.2.2. The Assembler Language

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).

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;       // controller's accumulator <= the 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; carryVector <= <1 1 1 0 0 ...>
cNOP; WHERECARRY; // activeVector <= <1 1 1 0 ... 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, non-standard versions of this architecture are used for generating efficiently pseudo-random number sequences [3].

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 in developing the parallel algorithm were: choosing the sequential Prim's algorithm for finding the minimum spanning tree from a graph, 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) and upgrading with few new general features our generic MapReduce structure to make it more efficient for further general purpose applications.

3. Dense Minimum Spanning Tree Algorithm

We consider our graph has n vertices. Therefore the dense matrix used for representing the graph is $n \times n$ and it is stored in n vectors $v(0), \ldots v(n-1)$.

Definition 1 *Spanning tree (ST) of an undirected graph G: a tree containing all the vertices of G.*
◇

Definition 2 *Minimum spanning tree (MST) of a weighted undirected graph is the ST with minimum weight.*
◇

3.1. The Algorithm

Let be $G = (V, E, w)$. Consider V_T the set of vertices already added to the result. The final result is $G' = (V, E', w')$, where $E' \subseteq E$ with $\Sigma\, w'$ minimal.

1. select the starting vertex: $V_T = \{r\}$, with $r \in V$
2. select from $(V - V_T)$ the vertex pointed by the minimum weighted edge starting from V_T , add it and the corresponding edge to the final result
3. if $V_T \neq V$ go the step 2, else stop.

Let be the example form the section 7.2 in [4] (see Fig. 3). The associated storage resources are:

```
dest : a b c d e f : destination vector
v(a) : 0 1 3 9 9 2 : distance vector from a
```

```
v(b) : 1 0 5 1 9 9 : distance vector from b
v(c) : 3 5 0 2 1 9 : distance vector from c
v(d) : 9 1 2 0 4 9 : distance vector from d
v(e) : 9 9 1 4 0 5 : distance vector from e
v(f) : 2 9 9 9 5 0 : distance vector from f

INITIALLY:
dist  : 9 9 9 9 9 9 : distance vector; all distances are infinite
source: x x x x x x : source vector; initially unspecified
active: 1 1 1 1 1 1 0 0 ... 0 : Boolean vector used to activate cells
acc   : x           : controller's accumulator; initially unspecified
```

where: the weighted adjacency matrix is $(v(a), v(b), ...v(f))$; the value 9 stands for ∞.

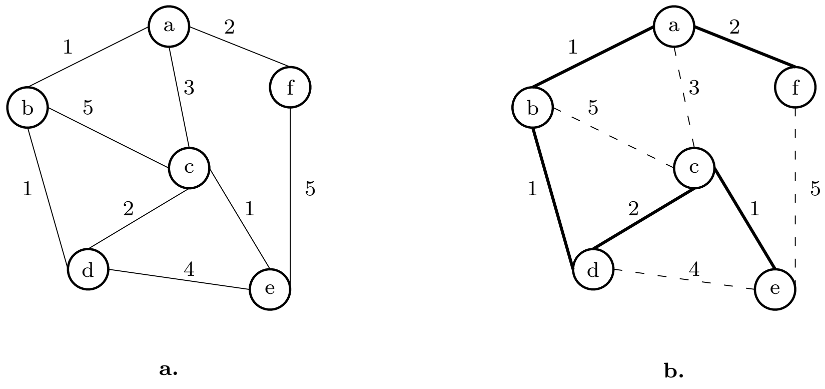

Fig. 3. a. The initial form of the graph used as example.
b. The result (dashed lines are used to represent the removed edges).

The parallel Minimum Spanning Tree Algorithm on the MapReduce engine for dense matrix representation is:

```
acc <= inital_vertex;
where (dest = acc)
    dist <= 0;                  \\ set distance 0 to itself
    active <= 0;                \\ inactivate the associated column
endwhere
while (reductionOr(active) = 1){ \\ do until all the cells are inactive
    where (v(acc) < dist)       \\ only for vertexes connected to acc
        source <= acc;          \\ the possible sources are indicated
        dist <= v(acc);         \\ the associated values are considered
    endwhere
    acc <= reductionMin(dist);  \\ selects the shortest edge
    where (first(dist = acc))   \\ select the associated column
        acc <= reductionOr(dest);\\ selects a new vertex in dest vector
    endwhere
    at (dest = acc) active <= 0; \\ inactivate the associated column
}
```

The execution time, evaluated according to the program presented in Appendix 1, is:

$$T_{PRIM}(N) = (N-1)logN + 25N - 15 \in O(NlogN)$$

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

```
vect[1] = 0 1 3 9 9 2 x ...
vect[2] = 1 0 5 1 9 9 x ...
vect[3] = 3 5 0 2 1 9 x ...
vect[4] = 9 1 2 0 4 9 x ...
vect[5] = 9 9 1 4 0 5 x ...
vect[6] = 2 9 9 9 5 0 x ...
```

The value 9 stands for ∞*.*

The TEST program is listed in Appendix 1. The result is:

```
destination = vect[7] = 1 2 3 4 5 6 x ... == a b c d e f
distance    = vect[8] = 1 0 2 1 1 2 x ...
source      = vect[9] = 2 x 4 2 3 1 x ... == b x d b c a
```

The result is interpreted as follows: the edges and their weights of the resulting graph are:

$$(b, a, 1), (d, c, 2), (b, d, 1), (c, e, 1), (a, f, 2)$$

The execution time, provided by the simulator, is: $T_{PRIM}(6) = 156$ *cycles.*

$\diamond$

4. Sparse Minimum Spanning Tree Algorithm

4.1. The Algorithm

The parallel Minimum Spanning Tree Algorithm on the MapReduce engine for *sparse matrix* representation is:

```
acc <= initial_value;       // set current vertex
w <= v;
r <= (0 0 ... 0);           // reset the vector result
a <= (0 0 ... 0);           // inactivate cells involved in computation
while (redMin(a) = 0) {
    activate (l = acc);     // activate where l = acc; NEW FEATURE
    activate (c = acc);     // activate where c = acc; NEW FEATURE
    vx <= acc;              // save current vertex
    acc <= reductionMin(w); // acc loaded with minimum from selected
    where (w = acc)
        where (first)
```

```
            w <= 9; \\ 9 is our "infinite"
            r <= 1;
            if (reductionAdd(l) = vx)
                    acc <= reductionAdd(c); // next vertex
             else   acc <= reductionAdd(l); // next vertex
        endWhere
    endWhere
}
```

The execution time for a graph with $|V| = N$ is:

$$T_{PRIM\_SM\_max} = 2(N-1)logN + 334N - 26 \in O(NlogN)$$

for $|E|$ execution units (according to the program and evaluation done in Appendix 2).

Example 4 *Let us take the same example as for the dense matrix representation and turn it into a sparse matrix. The graph we use is shown in Fig. 3. The program uses a numerical representation for the vertexes ($a \to 1, b \to 2, \ldots, f \to 6$) of the graph. Therefore, the representation looks as follows:*

```
   1 2 3 4 5 6        1 2 3 4 5 6

1  0 1 3 9 9 2     1  0 0 0 0 0 0
2  1 0 5 1 9 9     2  1 0 0 0 0 0
3  3 5 0 2 1 9     3  3 5 0 0 0 0
4  9 1 2 0 4 9     4  0 1 2 0 0 0
5  9 9 1 4 0 5     5  0 0 1 4 0 0
6  2 9 9 9 5 0     6  2 0 0 0 5 0
```

The sparse representation is:

```
l: 2 3 3 4 4 5 5 6 6 \\ line
c: 1 1 2 2 3 3 4 1 5 \\ column
v: 1 3 5 1 2 1 4 2 5 \\ value
```

For each edge there is an element in the three vectors used to represent the graph. The test program is listed in Appendix 2.
The first lines of the program initializes the local memory in cells and controller:

```
vect[0] = 2 3 3 4 4 5 5 6 6 0 0 0 0 0 0 0 // line
vect[1] = 1 1 2 2 3 3 4 1 5 0 0 0 0 0 0 0 // column
vect[2] = 1 3 5 1 2 1 4 2 5 0 0 0 0 0 0 0 // value
vect[3] = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
vect[4] = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 // result
```

The final result is:

```
vect[0] = 2 3 3 4 4 5 5 6 6 0 0 0 0 0 0 0
vect[1] = 1 1 2 2 3 3 4 1 5 0 0 0 0 0 0 0
vect[2] = 1 3 5 1 2 1 4 2 5 0 0 0 0 0 0 0
vect[3] = 9 3 5 9 9 9 4 9 5 0 0 0 0 0 0 0
vect[4] = 1 0 0 1 1 1 0 1 0 0 0 0 0 0 0 0
```

The value 1 in `vect[4]` *point to the edges of the resulting graph, the same as in the result provided by the dense matrix representation algorithm.*

The execution time, provided by the simulator, is: $T_{PRIM\_SM}(6) = 198$ *cycles.*
$\diamond$

5. Upgraded Version of MapReduce Architecture and Organization

The Prim's algorithm based on sparse matrix representation requires, for simplicity and performance, additional forms of spatial selection. This new features, marked in the algorithm description with the comment NEW FEATURE, meant adding three new instructions for the array:

actwhere : activate the cells where the local accumulator is equal with the controller's accumulator (`acc = acc[i]`)

saveact : save the current activation configuration

restact : restore the saved activation configuration

The implementation of these new instructions meant reconsidering the relation between the two vectors used for managing the activation of cells:

```
    reg [a-1:0] actVect[0:(1<<x)-1] ; // activation vector
    reg         boolVect[0:(1<<x)-1]; // Boolean vector
```

according to the following relation:

```
    boolVect[k] = (actVect[k] == 0) || (actVect[k][a-1] == 1);
```

The actions on the activation vector `actVect[0:(1<<x)-1]` are described as follows:

```
if (arrayOperand == ctl)
    case(arrayOpCode)
        ...
        actwhere: actVect[i] <= (!boolVect[i] && (accVect[i] == acc))
                              ? 0 : actVect[i]                   ;
        saveact : actVect[i] <= actVect[i] - 1                   ;
        restact : actVect[i] <= actVect[i] + 1                   ;
    endcase
```

At the assembly language level these instructions act as follows:

```
cNOP;      ACTIVATE;    // bv = (1 1 1 1 ... 1 1)
...        ...
cNOP;      WHERECARRY;  // bv = (0 0 0 1 ... 1 1)
...        ...
cNOP;      ACTWHERE;    // bv = (0 1 0 1 ... 1 1)
```

```
...        ...
cNOP;      RESTACT;      // bv = (0 0 0 1 ... 1 1)
...        ...
cNOP;      ENDWHERE;     // bv = (1 1 1 1 ... 1 1)
```

These new instructions allows the programmer to conditionally activate cells, not only to conditionally deactivate cells.

6. Concluding Remarks

Prim's algorithm runs on sequential computers in $O(N^2)$, where $N = |V|$ (number of vertices of the graph). On a *hyper-cube* engine with p ***processors*** and size in $O(NlogN)$ the execution time is in $O(NlogN)$ with $p = N$. On our MapReduce engine, with N ***execution units*** and size in $O(N)$, the execution time is in the same magnitude order, $O(NlogN)$. *So, the execution time is the same but the engine is smaller and simpler.*

Besides the theoretical results, the actual measurements are very favourable for our approach. While the Intel's Xeon Phi provides $\sim 150pJ/op$ in 22 nm technology, our approach provides $7pJ/op$ in 28 nm technology.

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

References

[1] ASANOVIC K., BODIK R., CATANZARO B. C., GEBIS J. J., HUSBANDS P., KEUTZER K., PATTERSON D. A., PLSKER W. L., SHALF J., WILLIAMS S. W., YELICK K. A., *The Landscape of Parallel Computing Research: A View from Berkeley*, Tech. Rep. No. UCB/EECS-2006-183, 2006.

[2] BÎRA C., HOBINCU R., PETRICĂ L., CODREANU V., COŢOFANĂ S., *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, Advances in Information Science and Applications - volume **II**, Santorini, Greece, 2014, pp. 432–437.

[3] GHEOLBĂNOIU A., MOCANU D., HOBINCU R., PETRICĂ L., *Cellular Automaton pRNG with a Global Loop for Non-Uniform Rule Control*, Proceedings of the 18th International Conference on Computers, Advances in Information Science and Applications - volume **II**, Santorini, Greece, 2014, pp. 415–420.

[4] KUMAR V., GRAMA A., GUOTA A., KARYPIS G., *Introduction to Parallel Computing. Design and Analysis of Algorithms*, The Benjamin/Cummings Pub. Comp. Inc., 1994.

[5] PAPAGIANNIS A., NIKOLOPOULOS D.S., *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.

[6] PRIM R. C., *Shortest connection networks and some generalizations*, Bell System Technical Journal **36** (6), 1957, pp. 1389–1401.

[7] ŞTEFAN G. M., MALIŢA M., *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 - volume **II**, Santorini Island, Greece, 2014, pp. 582–597.

[8] ŞTEFAN G. M., *One-Chip TeraArchitecture*, Proc. of the 8th Applications and Principles of Information Science Conference, Okinawa, Japan, 2009.

[9] TRIPATHY A., PATRA A., MOHAN S., MAHAPATRA R., *Distributed Collaborative Filtering on a Single Chip Cloud Computer*, IEEE Conference on Cloud Engineering (IC2E), 2013, pp. 140–145.

Appendix 1: Dense Matrix Representation Program

The Program

The program for dense matrix representation is the following, named `03_PRIM.v`:

```
        cNOP;           IXLOAD;
        cNOP;           VADD(1);
        cNOP;           STORE(7);   // dest : 1 2 3 4 5 6
        cNOP;           VLOAD(9);
        cNOP;           STORE(8);   // dist : 9 9 9 9 9 9
        cVLOAD(2);      LOAD(7);    // acc <= 2; acc <= dest
        cNOP;           CSUB;       // acc[i] <= dest[i] - acc
        cNOP;           WHEREZERO;  // where dest[i] = acc
        cNOP;           STORE(8);   // dist[i] <= 0
        cSTORE(3);      ELSEWHERE;  // inactivate the cell
LB(1);  cLOAD(3);       NOP;
        cNOP;           CALOAD;     // acc[i] <= mem[acc][i] = mem[2][i]
        cNOP;           SUB(8);
        cNOP;           WHERECARRY;
        cNOP;           CLOAD;
        cNOP;           STORE(9);
        cNOP;           CALOAD;
        cNOP;           STORE(8);
        cNOP;           ENDWHERE;
        cNOP;           LOAD(8);
        cNOP;           NOP;
        cNOP;           NOP;
        cNOP;           NOP;
        cCLOAD(1);      NOP;
        cNOP;           CSUB;
        cNOP;           WHEREZERO;
        cNOP;           NOP;
        cNOP;           WHEREFIRST;
        cNOP;           LOAD(7);
```

```
cNOP;           NOP;
cNOP;           ENDWHERE;
cNOP;           ENDWHERE;
cCLOAD(0);      NOP;
cNOP;           CSUB;
cSTORE(3);      WHERENZERO;
cLOAD(4);       NOP;
cVSUB(1);       NOP;
cSTORE(4);      NOP;
cBRNZ(1);       NOP;
```

The Evaluation

The test program for 03_PRIM.v is:

```
cNOP;       ACTIVATE;
cVLOAD(6);  IXLOAD;     // acc <= N; acc[i] <= index
cVSUB(1);   CSUB;       // acc <= acc - 1; acc[i] <= index - N
cSTORE(4);  WHERECARRY; // select only the first N cells
    'include "03_matrixLoad.v"
cSTART;     NOP;
    'include "03_PRIM.v"
cSTOP; NOP;
cHALT; NOP;
```

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

```
cVPUSHL(2);     NOP;
cVPUSHL(9);     NOP;
cVPUSHL(9);     NOP;
cVPUSHL(3);     NOP;
cVPUSHL(1);     NOP;
cVPUSHL(0);     NOP;
cNOP;           SRLOAD;
cNOP;           STORE(1);
cVPUSHL(9);     NOP;
cVPUSHL(9);     NOP;
cVPUSHL(1);     NOP;
cVPUSHL(5);     NOP;
cVPUSHL(0);     NOP;
cVPUSHL(1);     NOP;
cNOP;           SRLOAD;
cNOP;           STORE(2);
cVPUSHL(9);     NOP;
cVPUSHL(1);     NOP;
cVPUSHL(2);     NOP;
cVPUSHL(0);     NOP;
cVPUSHL(5);     NOP;
cVPUSHL(3);     NOP;
cNOP;           SRLOAD;
```

```
cNOP;           STORE(3);
cVPUSHL(9);     NOP;
cVPUSHL(4);     NOP;
cVPUSHL(0);     NOP;
cVPUSHL(2);     NOP;
cVPUSHL(1);     NOP;
cVPUSHL(9);     NOP;
cNOP;           SRLOAD;
cNOP;           STORE(4);
cVPUSHL(5);     NOP;
cVPUSHL(0);     NOP;
cVPUSHL(4);     NOP;
cVPUSHL(1);     NOP;
cVPUSHL(9);     NOP;
cVPUSHL(9);     NOP;
cNOP;           SRLOAD;
cNOP;           STORE(5);
cVPUSHL(0);     NOP;
cVPUSHL(5);     NOP;
cVPUSHL(9);     NOP;
cVPUSHL(9);     NOP;
cVPUSHL(9);     NOP;
cVPUSHL(2);     NOP;
cNOP;           SRLOAD;
cNOP;           STORE(6);
```

Appendix 2: Sparse Matrix Representation Program

The Program

We use a full numeric representation. Thus, each edge is numbered from 1 to N. The program `03_PRIM_SM.v` is:

```
        cLOAD(1);       VLOAD(0);   // acc <= mem[1]; acc[i] <= 0
        cNOP;           STORE(4);   // mem[i][4] = r[i] <= 0
        cNOP;           LOAD(2);    // acc[i] <= value[i]
        cNOP;           STORE(3);   // mem[i][3] = w[i] <= v[i];
        cNOP;           ENDWHERE;   // act[i] <= 1; b[i] <= 0
        cNOP;           WHERECARRY; // to desactivate all cells
        cNOP;           SAVEACT;    //
LB(1);  cLOAD(1);       LOAD(0);    // acc <= vx; acc[i] <= line[i]
        cNOP;           RESTACT;    //
        cNOP;           ACTWHERE;   // activate where acc = acc[i]
        cNOP;           SAVEACT;    //
        cNOP;           LOAD(1);    // acc[i] <= column[i]
        cNOP;           RESTACT;    //
        cSTORE(1);      ACTWHERE;   // activate where acc = acc[i]
        cNOP;           LOAD(3);    // acc[i] <= w[i] = mem[i][3]
        cNOP;           NOP;        //
        cNOP;           NOP;        //
        cNOP;           NOP;        //
        cCLOAD(1);      NOP;        // acc <= min(w)
        cNOP;           CSUB;       // acc[i] <= acc[i] - acc
        cNOP;           WHEREZERO;  // select where w = acc = min(w)
        cNOP;           NOP;        // TO BE SOLVED!
        cNOP;           WHEREFIRST; // select first selected
        cNOP;           VLOAD(9);   // acc[i] <= 9; 9 = "infinite"
        cNOP;           STORE(3);   // w <= 9, where selected
        cNOP;           VLOAD(1);   // acc[i] <= 1
        cNOP;           STORE(4);   // r <= 1, where selected
        cNOP;           LOAD(0);    // acc[i] <= line
        cNOP;           NOP;        //
        cNOP;           NOP;        //
        cNOP;           NOP;        //
        cCLOAD(0);      NOP;        // acc <= l selected
        cSUB(1);        NOP;        // acc <= acc - vx
        cBRZ(2);        NOP;        // if (acc=0) jmp to LB(2)
        cCLOAD(0);      NOP;        // reload acc <= l selected
        cSTORE(1);      NOP;        // vx <= next vertex
        cJMP(3);        NOP;        //
LB(2);  cNOP;           LOAD(1);    // acc[i] <= column
        cNOP;           NOP;        //
        cNOP;           NOP;        //
        cNOP;           NOP;        //
        cCLOAD(0);      NOP;        // acc <= c selected
        cSTORE(1);      NOP;        // vx <= next vertex
LB(3);  cNOP;           ENDWHERE;
        cNOP;           ENDWHERE;
        cNOP;           VLOAD(1);   // acc[i] <= 1
        cNOP;           NOP;        //
        cNOP;           NOP;        //
        cNOP;           NOP;        //
        cCLOAD(0);      NOP;        // acc <= redAdd
        cSUB(0);        SAVEACT;    // acc <= acc - N;
        cBRNZ(1);       NOP;        //
```

The Evaluation

The TEST program is:

```
cNOP;           ACTIVATE;   // activate all cells
cNOP;           VLOAD(0);   //
cNOP;           STORE(0);   // mem[i][0] <= 0 for simulation
cNOP;           STORE(1);   // mem[i][1] <= 0 for simulation
cNOP;           STORE(2);   // mem[i][2] <= 0 for simulation
cNOP;           STORE(3);   // mem[i][3] <= 0 for simulation
cVLOAD(9);      STORE(4);   // |E| = 9; mem[i][4] <= 0 for simulation
cSTORE(0);      IXLOAD;     // mem[0] <= N; acc[i] <= index
```

```
cVSUB(1);       CSUB;       // acc <= acc - 1; acc[i] <= index - N
cSTORE(4);      WHERECARRY; // select only the first N cells
cVLOAD(2);      NOP;        // vertex 2 is the starting vertex
cSTORE(1);      NOP;        // mem[1] <= starting vertex
    ‘include "03_matrixLoad.v"
cSTART;         NOP;
    ‘include "03_PRIM_SM.v"
cSTOP;          NOP;
cHALT;          NOP;
```

The program `03_matrixLoad.v`, which loads the weighted adjacency matrix in sparse form, is:

```
cVPUSHL(6); NOP;
cVPUSHL(6); NOP;
cVPUSHL(5); NOP;
cVPUSHL(5); NOP;
cVPUSHL(4); NOP;
cVPUSHL(4); NOP;
cVPUSHL(3); NOP;
cVPUSHL(3); NOP;
cVPUSHL(2); NOP;
cNOP;       SRLOAD;
cNOP;       STORE(0); // v0: line
cVPUSHL(5); NOP;
cVPUSHL(1); NOP;
cVPUSHL(4); NOP;
cVPUSHL(3); NOP;
cVPUSHL(3); NOP;
cVPUSHL(2); NOP;
cVPUSHL(2); NOP;
cVPUSHL(1); NOP;
cVPUSHL(1); NOP;
cNOP;       SRLOAD;
cNOP;       STORE(1); // v1: column
cVPUSHL(5); NOP;
cVPUSHL(2); NOP;
cVPUSHL(4); NOP;
cVPUSHL(1); NOP;
cVPUSHL(2); NOP;
cVPUSHL(1); NOP;
cVPUSHL(5); NOP;
cVPUSHL(3); NOP;
cVPUSHL(1); NOP;
cNOP;       SRLOAD;
cNOP;       STORE(2); // v2: value
```