

**CSE2003**  
**Data Structures and Algorithms**  
**J Component Report**

A project report titled  
**Graph Partition for VLSI**

*By*

19BEC1412	Yeshwanth Reddy P
19BEC1129	Akash Gowda K R
19BEC1073	Sai Kumar K

BACHELOR OF TECHNOLOGY  
IN  
ELECTRONICS AND COMMUNICATION ENGINEERING



**VIT<sup>®</sup>**  
**Vellore Institute of Technology**  
(Deemed to be University under section 3 of UGC Act, 1956)

*Submitted to*

**Dr. R. KARTHIK**

*April 2022*

**School of Electronics Engineering**

**DECLARATION BY THE CANDIDATE**

I hereby declare that the Report entitled “**Graph Partition for VLSI**” submitted by me to VIT Chennai is a record of bonafide work undertaken by me under the supervision of **Dr. R. Karthik, Senior Assistant Professor, SENSE, VIT Chennai.**

Signature of the Candidate

*Yeshwanth*

**Yeshwanth Reddy**

*Akash*

**Akash Gowda K R**

*Sai Kumar*

**Sai Kumar K**

Chennai

23/04/2022.

## ACKNOWLEDGEMENT

We wish to express our sincere thanks and deep sense of gratitude to our project guide, **Dr. R. Karthik**, School of Electronics Engineering for his consistent encouragement and valuable guidance offered to us in a pleasant manner throughout the course of the project work.

We are extremely grateful to **Dr. Sivasubramanian. A**, Dean of the School of Electronics Engineering (SENSE), VIT University Chennai, for extending the facilities of the School towards our project and for his unstinting support.

We express our thanks to our **Head of The Department Dr. Vetrivelan. P (for B.Tech-ECE)** for his support throughout the course of this project.

We also take this opportunity to thank all the faculty of the School for their support and their wisdom imparted to us throughout the courses till date.

We thank our parents, family, and friends for bearing with us throughout the course of our project and for the opportunity they provided us in undergoing this course in such a prestigious institution.

## **BONAFIDE CERTIFICATE**

Certified that this project report entitled “**Graph Partition for VLSI**” is a bonafide work of **YESHWANTH REDDY (19BEC1412), AKASH GOWDA K R (19BEC1129) and SAI KUMAR K (19BEC1073)** carried out the “J”-Project work under my supervision and guidance for CSE2003 Data Structures and Algorithms.

**Dr.R.Karthik**

School of Electronics Engineering

VIT University, Chennai

Chennai – 600 127.

**TABLE OF CONTENTS**

<b>S.NO</b>	<b>Chapter</b>	<b>PAGE NO.</b>
1	Chapter -1 Introduction	7-8
2	Chapter – 2 Requirements and proposed system	9
3	Chapter -3 Module description	10-19
4	Chapter 4 – Results and	20-27
5	Conclusion	28
	References	29

## ABSTRACT

The graph partitioning problem is that of dividing the vertices of a graph into sets of specified sizes such that few edges cross between sets. This NP-complete problem arises in many important scientific and engineering problems. Prominent examples include the decomposition of data structures for parallel computation, the placement of circuit elements and the ordering of sparse matrix computations. We present a multilevel algorithm for graph partitioning in which the graph is approximated by a sequence of increasingly smaller graphs. The smallest graph is then partitioned using a spectral method, and this partition is propagated back through the hierarchy of graphs. A variant of the Kernighan-Lin algorithm is applied periodically to refine the partition. The entire algorithm can be implemented to execute in time proportional to the size of the original graph. Experiments indicate that, relative to other advanced methods, the multilevel algorithm produces high quality partitions at low cost.

Move-based iterative improvement partitioning methods such as the Fiduccia-Mattheyses (FM) algorithm and Krishnamurthy's Look-Ahead (LA) algorithm are widely used in VLSI CAD applications largely due to their time efficiency and ease of implementation. This class of algorithms is of the "local improvement" type. They generate relatively high quality results for small and medium size circuits. However, as VLSI circuits become larger, these algorithms are not so effective on them as direct partitioning tools. We propose new iterative-improvement methods that select cells to move with a view to moving clusters that straddle the two subsets of a partition into one of the subsets. The new algorithms significantly improve partition quality while preserving the advantage of time efficiency. Experimental results on 25 medium to large size ACM/SIGDA benchmark circuits show up to 70% improvement over FM in cutsize, with an average of per-circuit percent improvements of about 25%, and a total cut improvement of about 35%. They also outperform the recent placement-based partitioning tool Paraboli and the spectral partitioner MELO by about 17% and 23%, respectively, with less CPU time. This demonstrates the potential of iterative improvement algorithms in dealing with the increasing complexity of modern VLSI circuitry.

## CHAPTER – 1

### INTRODUCTION

VLSI world completely revolves around integrated circuits (IC). Manufacturing process of an IC follows through a lot of different steps, out of which typically IC design cycle involves following steps:

1. System Specification.
2. Architectural Design.
3. Logic Design.
4. Logic Synthesis.
5. Physical Design.
6. Physical verification and Tapeout.
7. Wafer Fabrication.
8. Packaging.

Amongst all these steps, we will be dealing with Physical Design. Once the logical description of the design has been completely synthesized into a netlist, the physical design converts this netlist based circuit representation into a geometrical representation.

Physical design further involves following steps:

1. Partitioning.
2. Floor-planning.
3. Placement.
4. Clock-tree synthesis.
5. Routing.
6. Timing verification.

It would be very time consuming or even impossible to hand-craft the circuit design of large complexity without any assistance of computer programs during all the steps discussed above. According to Moore's law, number of transistors on a single chip doubles every 2 years, the state of the art chips consists of an entire system with millions of transistors. For example, Pentium processors from Intel has 3 million transistors. Computer programs are

used widely into the industries to automate these physical design processes with no or very little human intervention. As a part of Design automation of VLSI systems course. We have implemented automatic partitioning and placement engines.

### **Importance of Partitioning:**

Partitioning is the process of dividing a chip into different functional blocks and aim is to keep the frequently communicating blocks together. This reduces the cost and efforts of further design stages. In case of FPGA based design, when the design is too large to map on a single FPGA board, the circuit is partitioned and multiple blocks are mapped on different boards. The aim of the partitioning process is to divide the design in such a way that, the interconnection between the blocks are minimized. If the interconnections between two partitions are not minimized, then the functional blocks which are supposed to be together will be pushed away from each other which will affect the overall performance of the system. In FPGA based implementation, the interconnections between the boards will increase thereby increasing the complexity of interfacing the boards together. The partitioning problem takes in a netlist and divides the circuit represented by the netlist into two parts with a target to minimize the interconnection between them.



## CHAPTER – 2

### PROPOSED SYSTEM

#### **Circuit Partitioning using Fiduccia and Mattheyses algorithm:**

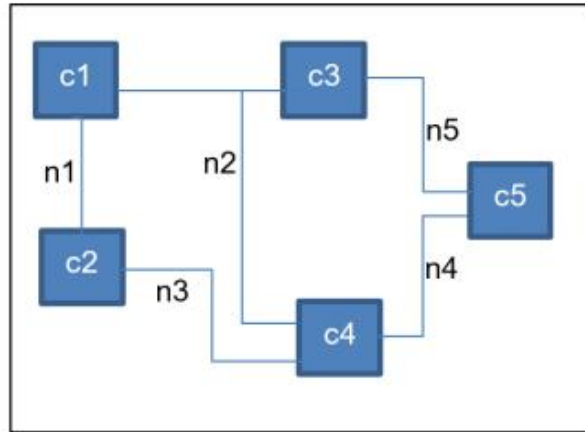
In the world of VLSI Design automation, the circuit partitioning is the most fundamental step that is followed during the design of the backend of the EDA tools. With the help of partitioning, the circuit is divided into two parts in such a way that the cells which interact together more frequently are kept close by i.e. in the same partition. In other words, the number of nets across the boundary of cut is minimized. For partitioning purpose, an electronic circuit is converted into graph-based model  $G(V,E)$ . A graph partitioning method was proposed by Kernighan-Lin (KL method) in 1970.

In this method nodes from two partitions are swapped with each other and the swap which increases the current cutset between two partitions is rejected. This was proven to be effective for circuits with relatively smaller solution space but it gets stuck in local minima for the bigger circuits. Another algorithm especially for circuit partitioning was proposed by Fiduccia Mattheyses in 1980 known as FM method. The FM method is an improvement over KL method in terms of both, execution time and final cut-set. Unlike KL method which makes the move only if there is an improvement, FM method encourages to make some bad moves in a hope that this will lead to a better solution (cut-set) in the future also known as Hill-Climbing. FM method proved to be the most effective method for circuit partitioning also, it makes use of efficient data structures to avoid unnecessary searching for the best cell to move and to minimize unnecessary updating of the cells affected by each move.

## CHAPTER – 3

### MODULE DESCRIPTION

#### Details of FM algorithm:



**Figure 1: Hypergraph model of a circuit**

Conversion of a circuit into graph model:

First, we convert the circuit into Hypergraph-based model  $G(V,E)$  as shown in the figure1

Where,

V: Set of cells in the circuit.  $\{c1, c2, c3, c4, c5\}$

E: Set of interconnect in the circuit.  $\{n1, n2, n3, n4, n5\}$

Let,

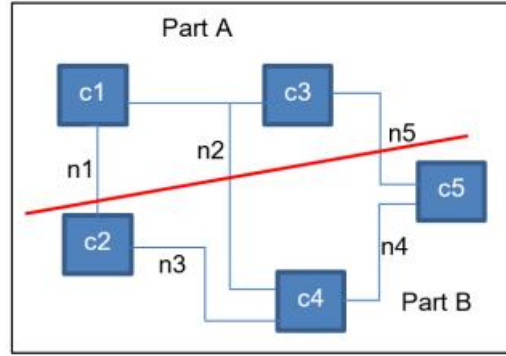
$n(i)$  be the number of cells on net  $n_i$ ,

$s(i)$  be the size of cell  $c_i$ ,

$p(i)$  be the no. of pins on cell  $c_i$ ,

Total number of pins in the circuit  $P = \sum_{i=1}^{cells} p(i)$

### Random partition and initial cutset:



**Figure 2: Random partition**

First step is to partition the circuit into random two parts such that it satisfies balance criteria. Figure 2 illustrates this on an example circuit. The concept of partitioning for minimum cutset is meaningless unless we put some restriction for the partition otherwise we could achieve an empty cutset by moving all the cells in one partition. In FM this restriction is implied with the ratio “r” for which,

Let  $s(A)$  and  $s(B)$  be the total sizes of part A and B respectively.

$r \approx \frac{s(A)}{s(A)+s(B)}$ , The partition satisfying  $0 < r < 1$  are accepted. After random partition the total number of nets crossing the partition boundary becomes the cutset of the circuit.

### Gain calculation:

Now we start making the moves towards minimizing this cutset.

The moves are based on the gain of a cell which is defined as, the change in cutset when the cell moves from its initial partition to the other partition.

For all nets connected to cell  $c(i)$ , the nets crossing the partition boundary are called external nets “E”, and remaining nets are internal nets “I” and gain  $g(i) = E - I$ .

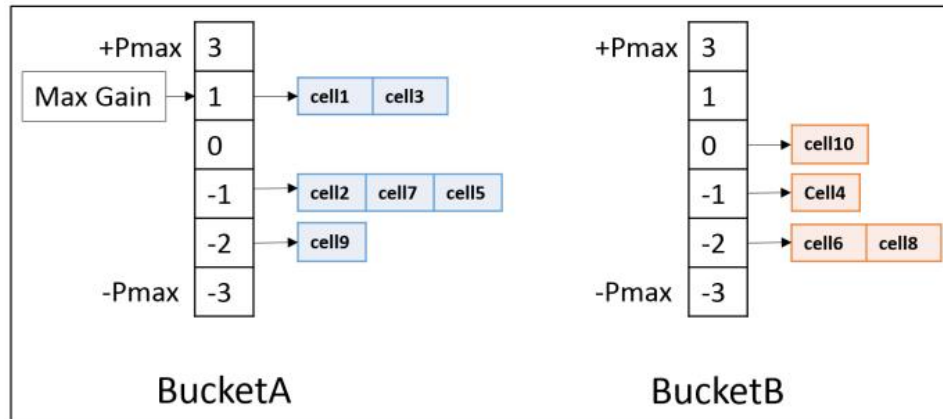
e.g. For cell  $c_4$ ,  $E=1$  and  $I=2$  as,  $n_2$  is an external net and  $n_3$  and  $n_4$  are internal nets.

$$g(4) = 1 - 2 = -1$$

The gain is negative/positive if the cutset increases/decreases after the move. The maximum gain of a cell  $c(i)$  can be  $+p(i)$  and minimum can be  $-p(i)$ .

## The Bucket Array:

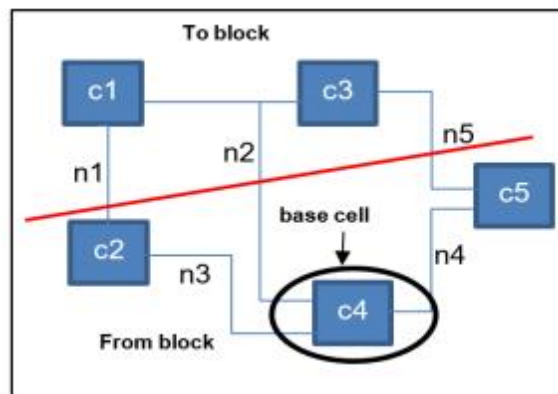
Once the initial random partitioning of the cells is done, their initial gains are calculated. An array type data structure is used to create two buckets, bucketA and bucketB one each for both partitions. Buckets store the list of cells with same gain as shown in the figure



**Figure 3: The Bucket Array**

A base cell of maximum gain is selected to move and along with buckets, a list of locked cells is kept. Whenever the base cell is moved from one partition to another, that cell is added to the locked list and is removed from the bucket list. Once it is moved to complimentary partition, the gains of all neighbors of the base cell may change. Their gains are updated along with their respective buckets. when no cells are left with a positive gain, then a cell with negative gain is still moved in a hope that it will improve the cutset after the move.

## Fast computation of gains:



**Figure 4: Illustration of 'From' and 'To' blocks**

The gain calculation discussed previously is a naive approach and take a lot of time when it comes to updating the gains after the move. To speed up this process a new method is proposed. Let,  $F$  be the current (from) block of the base cell and  $T$  be the complimentary (to) block as shown in figure 4.

For every net ' $n$ ' connected to the base cell  $F(n)$  and  $T(n)$  be the number of cells the net ' $n$ ' has in the 'from' and 'to' blocks respectively. The net participates in the cutset if  $T(n)$  is non-zero and the net is critical before the move if  $T(n)=0$  or  $1$  or  $F(n) = 1$  which is base cell. Similarly, the net is critical after the move if  $T(n)= 1$  or  $F(n)= 0$  or  $1$ . While updating the gains of the neighbors of the base cell, all these conditions are checked, and the gains of the respective cells are incremented or decremented according to the conditions. Updating the gains in this way saves a lot of time completing the entire process in linear time.

#### End criteria:

The process of moving a cell and updating the gains is continued until either all cells have been moved and locked or no more moves are possible. At this point, a pass is said to have completed and the next pass starts from the state where minimum cut was obtained in this pass

#### Implementation Details:

The FM algorithm discussed above was implemented in C++ and tested on the ISPD2016 benchmarks. The details of implementation are discussed below

#### Data structures:

As C++ is an object-oriented programming language. It offers object oriented constructs such as classes which can be used to store the parameters related to a specific object. In our implementation we used two classes, namely, cell and net and are described below

##### 1. Cell

CELL SIZE	CELL GAIN	LOCK STATUS	CELL PARTITION	CELL TYPE	NETLIST
Stores the size of the cell	The gain of the cell is stored in this member	Stores '1' if the cell is locked else stores '0'	Stores '0' if cell is in partition A else stores '-1' for partition B	Stores the type of cell e.g. LUT, DSP, RAM.	List of nets connected to the cell

**Table 1: Cell class**

Table 1 shows the members of the class ‘cell’. All cell related data from the cell class can be accessed using an object of that cell. The objects are stored in another construct i.e. map named as cell map. Map stores the data in (key, value) pair where key is a pointer to the value of the map. The data access through map can be done in  $O(1)$  time.

Cell No.(key)	Object of class cell (Value)
0	Object of cell corresponding to cell 0
1	Object of cell corresponding to cell 1
2	Cell size   Cell Gain   Lock Status   Cell Partition   Cell Type   Netlist
3	Object of cell corresponding to cell 0
.	.
.	.
.	.
n	Object of cell corresponding to cell n

**Table 2: Structure of map of cell**

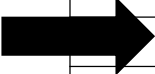
Table 2 shows the structure of cell map which stores the cell number as key and value is the object of the cell class corresponding to that cell with an example to illustrate how this map will be accessed for cell number 2.

## 2. Net

Cell List	A Size	B Size
Stores the list of cells connected to the net	Total number of cells connected to the net from partition A	Total number of cells connected to the net from partition B

**Table 3: Net class**

Table 3 shows the members of the class ‘net’. All net related data from the net class can be accessed using an object of that net. The objects are stored in another construct i.e. map named as net map.



Net name (key)	Object of class cell (Value)
net_0	Object of cell corresponding to cell 0
net_1	Object of cell corresponding to cell 1
net_2	Cell List   A size   B size
net_clk_buff	Object of cell corresponding to net clock buffer
.	.
.	.
.	.
i	Object of cell corresponding to cell i

**Table 4: Structure of map for nets**

Table 4 shows the structure of net map which stores the net name as a key and value is the object of the net class corresponding to that net with an example to illustrate how this map will be accessed for net 'net\_2'.

### 3. Buckets A and B

As discussed in section 2.1.3 we used maps to store the data of buckets. In this map the key is the gain of the cell and value is the list of cells corresponding to that gain. Two separate buckets bucketA and bucketB are used for partition A and B. Keys for the maps are always stored in a sorted order, so the key at the top always points at the list of cells with maximum gain one of which can be selected as a base cell. Again data access in  $O(1)$  time speeds up this process

### 4. Locked cells

A vector array is used to keep a list of locked cells. A cell is pushed into this vector when it moves from its partition to the complimentary partition and gets locked. This list also provides the information about the sequence in which the cells were moved and locked. We keep track of the cutset and the index of the cell, which was pushed into this list at the occurrence of minimum cutset, is stored. This index will be used to undo the moves made after mincut to start the next pass.

## Implementation:

Implementation starts with reading the nodes file which stores the information of cell number and cell type for all the cells from the circuit. While reading this file, the number

and its type is stored in the data structure. At the same time, the cells are pushed into one of the partition randomly. The pseudo code below shows this flow

**/\* cell list file \*/**

FOR each cell  $c = 1 \dots N$

    store the cell number cell  $c$ ;

    store cell type for cell  $c$ ;

    store size of cell  $c$ ;

    randomly pick a partition for cell  $c$  without violating balance criteria;

    unlock cell  $c$ ;

END for

Once the nodes file is done reading, netlist file read. This file contains the names of each net and the cell numbers connected to it. Using this information, the cell list of each net as well as the net list of each cell is updated in the data structure.

**/\* net list file \*/**

FOR each net  $n = 1 \dots N$

    FOR each cell  $c$  on the net  $n$

        update cell list of net  $n$ ;

        update net list of cell  $c$ ;

        if (cell  $c$  in partition A)

            A size++;

        else if (cell  $c$  in partition B)

            B size++;

    END for

END for

Now that all the information necessary for gain calculation is available, initial gains for all cells are calculated as follows:



**/\* Initial gain calculation \*/**

FOR each cell = 1 ... N

    FOR each net n in the netlist of cell c

        F <- current partition of cell c (FROM partition)

        T <- complimentary partition of cell c (TO PARTITION)

        if ( F(n) == 1 )

            gain(c)++;

        if ( T(n) == 0)

            gain(c)--;

END for net list

update bucket for FROM partition of cell c

END for cell

After this, the whole data structure is updated and ready to be used for actual partitioning process. This partitioning process follows the following steps:

**/\* MAIN LOOP\*/**

WHILE (!exit\_criteria)

{

    DO

        {

            Select a base cell to move

        } WHILE ( balance criteria not met )

        remove base cell from current bucket;

        lock base cell;

        push to locked cell list;

        cutset = cutset – gain(base cell);

        change partition of base cell;

        update\_gains();

    }

The `update_gains()` is the function which is called in the main function to update the gains of all neighboring cells to base cell. The update gain routine goes as follows:

```
/* UPDATE GAINS */
```

```
F <- current partition of base cell (FROM partition)
```

```
T <- complimentary partition of base cell (TO PARTITION)
```

```
FOR each net n in the netlist of base cell
```

```
  if ( base cell from A ) //assuming base cell is from partition A
```

```
    F(n) = A_size(n);
```

```
    T(n) = B_size(n);
```

```
  /* Check criticality before the move */
```

```
  if (T(n) == 0)
```

```
    Increment gains of all free cells on net n;
```

```
    Update_buckets();
```

```
  else if (T(n) == 1)
```

```
    Decrement the gain of the only cell (if unlocked) present in T partition on net n;
```

```
  Update_buckets();
```

```
  /* Making the move */
```

```
    F(n) --;
```

```
    T(n)++;
```

```
  /* After the move */
```

```
    A_size(n) = F(n);
```

```
    B_size(n) = T(n);
```

```
  /* Check criticality after the move */
```

```
  if (F(n) == 0)
```

```
    Decrement gains of all free cells on net n;
```

```
    Update_buckets();
```

```
  else if (F(n) == 1)
```

```
    Increment the gain of the only cell (if unlocked) present in T partition  
    on net n;
```

**Update\_buckets();**

END for cell

Each time when the gains were updated in update\_gains() function, update\_buckets() function is called. It was necessary to reflect the gain change in the bucket arrays. The function takes the cell whose gain has been changed as well as its previous gain as input and updates the bucket arrays accordingly. The flow of update\_gains() goes as follows:

**/\* UPDATE BUCKETS \*/**

F <- current partition of cell whose gain has been changed (FROM partition)

in bucket(F)

Remove the cell from the list pointed by previous gain;

Add the cell to the list pointed by new gain;

**/\*END UPDATE BUCKETS\*/**

After this the program returns to the main loop and selects next cell to move. This process continues until exit criteria is met.

### **Exit criteria:**

Exit criteria is same as discussed before, i.e. the program ends when no more cells are left unlocked or no more cells can be moved satisfying the balance criteria.

## CHAPTER – 4

### RESULTS AND DISCUSSION

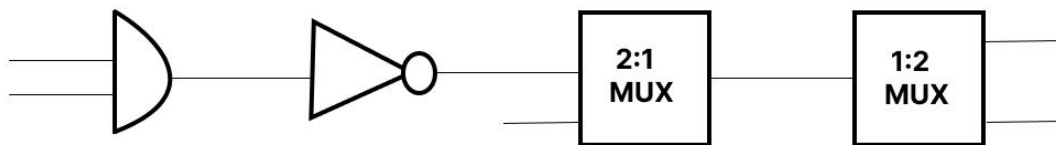
#### ➤ Graph partitioning for simple graphs

a) AND Gate

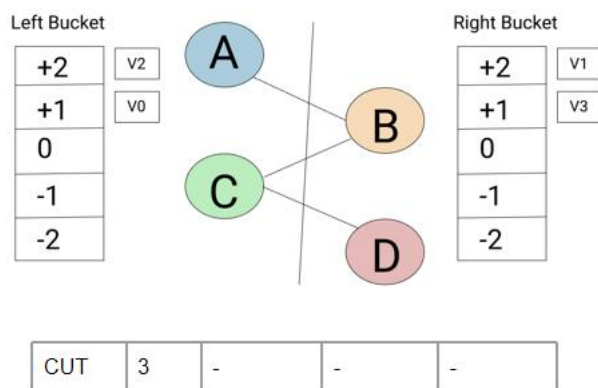
b) NOR gate

c) 2:1 MUX

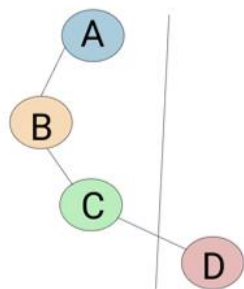
d) 1:2 MUX



**Step 1:**

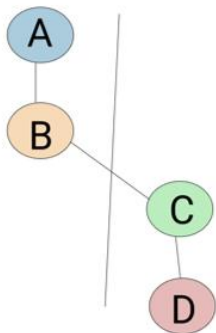


Step 2:



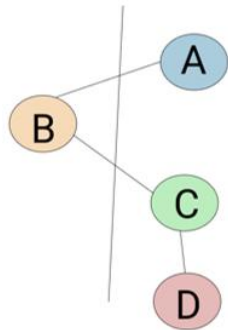
		V1	V2	V0	V3
CUT	3	1	-	-	-

Step 3:



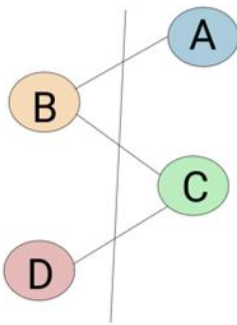
		V1	V2	V0	V3
CUT	3	1	1	-	-

Step 4:



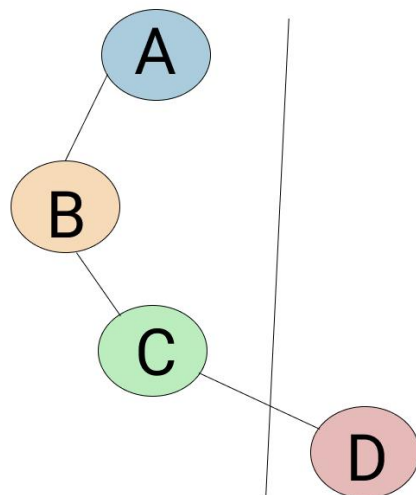
		V1	V2	V0	V3
CUT	3	1	1	2	-

Step 5:



		V1	V2	V0	V3
CUT	3	1	1	2	3

Final optimal solution with constraints min 1 and max 3



So overall gain  $\Rightarrow 3 - 1 = 2$

## C++ Output:

```

Number of Nodes:      4
Number of Nets: 3
Net 1   :2 0 1
Net 2   :2 1 2
Net 3   :2 2 4
constraint (min,max) : 1 3
Initial Random Partition :
P1 = {a c }
P2 = {b d }

Gain : 2      1      1      0
Node : b      a      c      d
move 'b' from partition 2 to partition 1
gain = 2
P1 = {a b c }
P2 = {d }

Gain : 2      0      -1     -1
Node : b      d      a      c
move 'a' from partition 1 to partition 2
gain = -1
P1 = {b c }
P2 = {a d }

Gain : 0      0      -1     -1
Node : b      d      a      c
move 'd' from partition 2 to partition 1
gain = 0
P1 = {b c d }
P2 = {a }

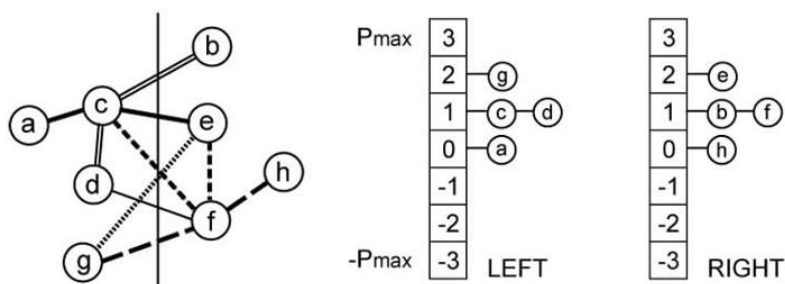
Gain : 0      0      -1     -1
Node : b      d      a      c
move 'c' from partition 1 to partition 2
gain = -1
P1 = {b d }
P2 = {a c }

Optimal Gain = 2
P1 = {a b c }
P2 = {d }

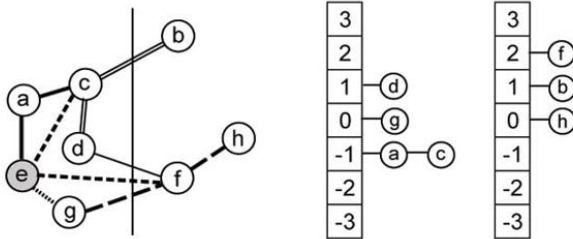
```

## ➤ Graph partitioning for hyper graphs

### Initial Partitioning:

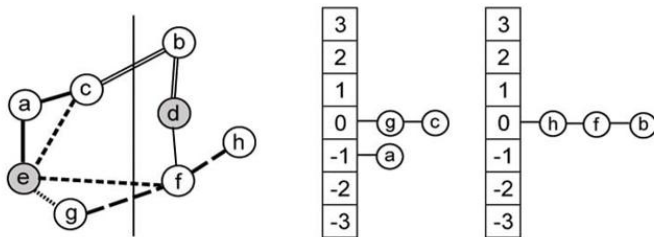


**First move:**



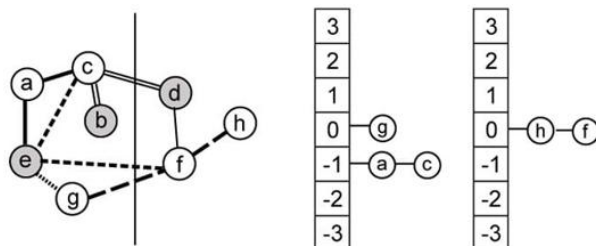
		E							
C U T	6	4							

**Second move:**



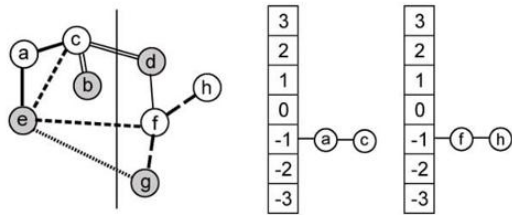
		E	D					
C U T	6	4	3					

**Third move:**

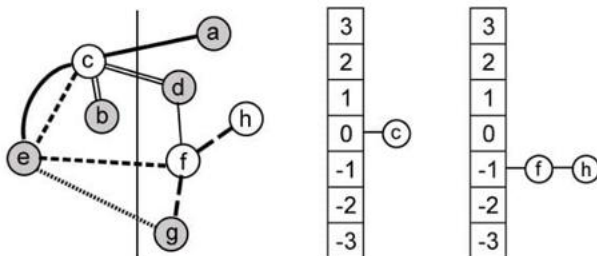


		E	D	B						
C U T	6	4	3	3						

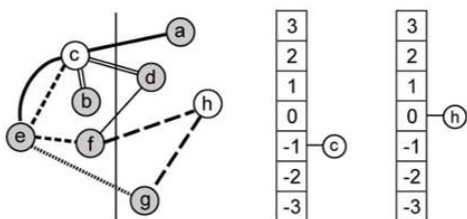


**Fourth move:**

		E	D	B	G				
CUT	6	4	3	3	3				

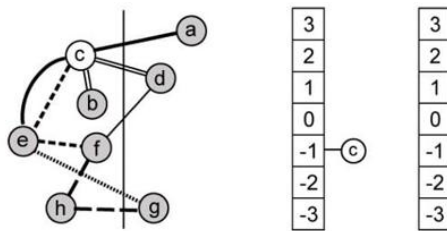
**Fifth move:**

		E	D	B	G	A			
CUT	6	4	3	3	3	4			

**Sixth move:**

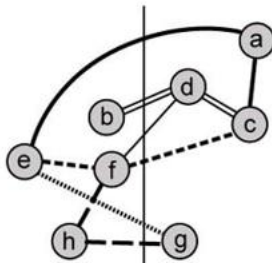
		E	D	B	G	A	F		
CUT	6	4	3	3	3	4	5		

**Seventh move:**



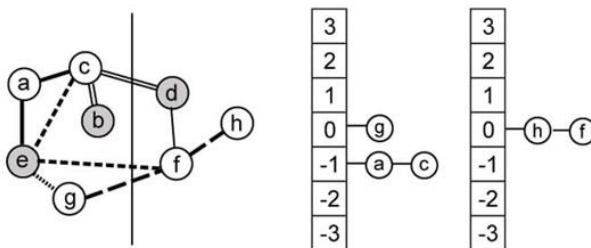
		E	D	B	G	A	E	H	
C	6	4	3	3	3	4	5	5	
U									
T									

**Eighth move:**



		E	D	B	G	A	E	H	C
C	6	4	3	3	3	4	5	6	6
U									
T									

Final optimal solution with constraints min 3 and max 5



		E	D	B					
C	6	4	3	3					
U									
T									

**Overall gain  $\Rightarrow 6 - 3 = 3$**

## C++ Output:

```

Number of Nodes:      8
Number of Nets: 6
Net 1 : 3 0 2 4
Net 2 : 3 1 2 3
Net 3 : 3 2 4 5
Net 4 : 3 5 6 7
Net 5 : 2 3 5
Net 6 : 2 4 6
constraint (min,max) : 3 5
Initial Random Partition :
P1 = {a c e g }
P2 = {b d f h }

Gain : 0      0      0      0      0      -1      -1      -2
Node :  b      c      f      g      h      a      d      e
move 'b' from partition 2 to partition 1
gain = 0
P1 = {a b c e g }
P2 = {d f h }

Gain : 0      0      0      0      0      -1      -1      -2
Node :  b      d      f      g      h      a      c      e
move 'g' from partition 1 to partition 2
gain = 0
P1 = {a b c e }
P2 = {d f g h }

Gain : 0      0      0      0      -1      -1      -1      -1
Node :  b      d      e      g      a      c      f      h
move 'd' from partition 2 to partition 1
gain = 0
P1 = {a b c d e }
P2 = {f g h }

Gain : 1      0      0      0      -1      -1      -1      -2
Node :  f      d      e      g      a      b      h      c
move 'e' from partition 1 to partition 2
gain = 0
P1 = {a b c d }
P2 = {e f g h }

Gain : 0      0      0      0      0      -1      -1      -2
Node :  a      c      d      e      f      b      h      g
move 'a' from partition 1 to partition 2
gain = 0
P1 = {b c d }
P2 = {a e f g h }

Gain : 1      0      0      0      -1      -1      -1      -2
Node :  c      a      d      f      b      e      h      g
move 'f' from partition 2 to partition 1
gain = 0
P1 = {b c d f }
P2 = {a e g h }

Gain : 0      0      0      0      0      -1      -1      -2
Node :  a      c      e      f      h      b      g      d
move 'c' from partition 1 to partition 2
gain = 0
P1 = {b d f }
P2 = {a c e g h }

Gain : 1      0      0      0      -1      -1      -1      -2
Node :  f      b      c      h      a      d      g      e
move 'h' from partition 2 to partition 1
gain = 0
P1 = {b d f h }
P2 = {a c e g }

Optimal Gain = 3
P1 = {a b c e g }
P2 = {d f h }

```

## CHAPTER – 5

### CONCLUSION

We have presented a multilevel algorithm for graph partitioning. The algorithm generates a sequence of smaller graphs that approximate the original graph, partitions the smallest graph in the sequence, and propagates the resulting partition back to the original graph while periodically performing a local refinement. This approach is very general and is applicable with a variety of different metrics on the partition quality. Our implementation of this method demonstrates excellent performance on a variety of graphs typical of those encountered in parallel scientific computing. It found better partitions than spectral bisection in a small fraction of the time. One shortcoming of the algorithm is that the construction of a sequence of graphs is memory intensive. A more aggressive coarsening strategy which generates fewer intermediate graphs would be one way of reducing this problem. It is worth noting that a parallel implementation of the algorithm described here is problematic because Kernighan{Lin is known to be P-complete. Furthermore, one attempt at a parallel implementation proved disappointing in performance. However, there are local refinement techniques that do parallelized well, and they could easily take the place of Kernighan{Lin in our implementation. We are currently studying these and related issues. Multilevel ideas have proved extremely powerful in continuous mathematics but have not yet made a large impact on discrete problems. We hope that our results will inspire further work on related algorithms for other combination problems.

## REFERENCES

- Abou-Rjeili, A., Karypis, G.: Multilevel algorithms for partitioning power-law graphs. In: 20th International Parallel and Distributed Processing Symposium (IPDPS). IEEE (2006)
- Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: 6th Symposium on Operating System Design and Implementation (OSDI), pp. 137–150. USENIX (2004)
- Andersen, R., Lang, K.J.: An algorithm for improving graph partitions. In: 19th ACM-SIAM Symposium on Discrete Algorithms, pp. 651–660 (2008)
- Hendrickson, B., Leland, R., Driessche, R.V.: Enhancing data locality by using terminal propagation. In: 29th Hawaii International Conference on System Sciences (HICSS 2009), vol. 1, p. 565. Software Technology and Architecture (1996)
- Andreev, K., Räcke, H.: Balanced graph partitioning. *Theory Comput. Syst.* 39(6), 929–939 (2006)