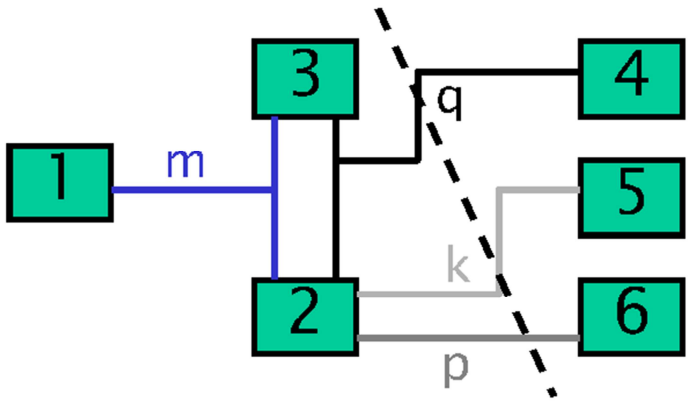
Partitioning (continued)

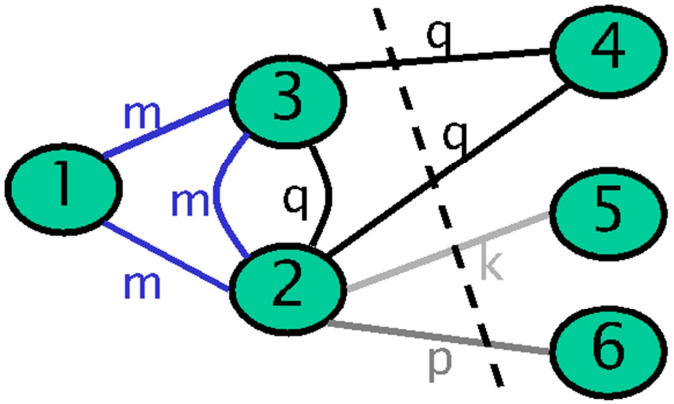
Fiduccia-Mattheyses (FM) Algorithm

- Modified version of KL
- A single vertex is moved across the cut in a single move
  - –Unbalanced partitions
- Vertices are weighted
- Concept of cutsize extended to hypergraphs
- Special data structure to improve time complexity
  - –(Main feature)
- Can be extended to multi-way partitioning

Why Hyperedges?

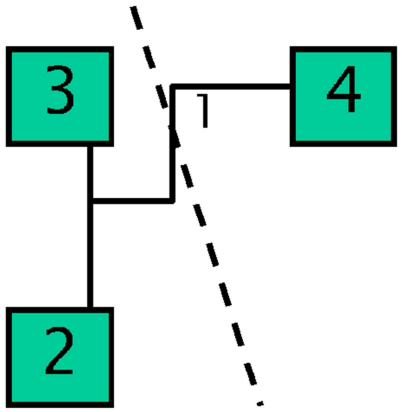
- For multi terminal nets, K-L may decompose them into many 2-terminal nets, but not efficient!
- Consider this example:
- If A = {1, 2, 3} B = {4, 5, 6}, graph model shows the cutsize = 4 but in the real circuit, only 3 wires cut
- Reducing the number of nets cut is more realistic than reducing the number of edges cut



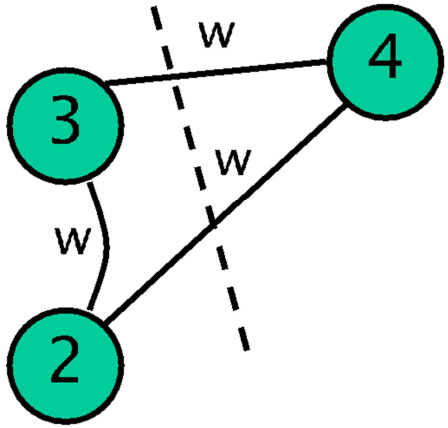


Hyperedge to Edge Conversion

- A hyperedge can be converted to a “clique”.



“Real” cut=1



“net” cut=2w

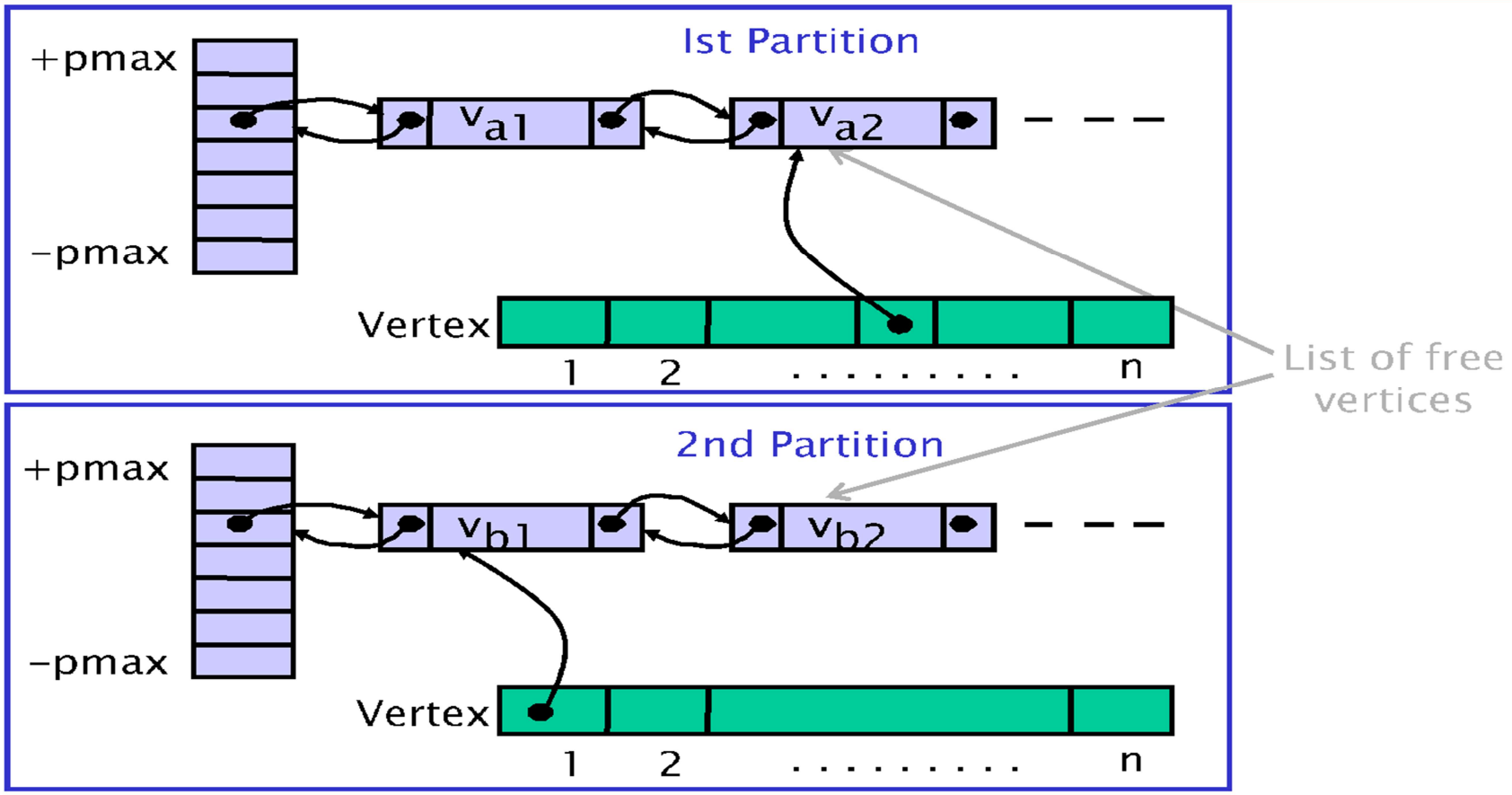
The FM Algorithm: Data Structure

- Pmax
  - Maximum gain
  - $p_{max} = d_{max} \cdot w_{max}$, where
    d_{max} = max degree of a vertex (# edges incident to it)
    w_{max} is the maximum edge weight
  - What does it mean intuitively?

- -Pmax .. Pmax array
  - Index *i* is a pointer to the list of unlocked vertices with gain *i*.

- Limit on size of partition
  - A maximum defined for the sum of vertex weights in a partition (alternatively, the maximum ratio of partition sizes might be defined)

The FM Algorithm: Data Structure



Each pointer in the array list A (or B) points to a linear list of unlocked vertices inside A (or B) with the corresponding gain

The FM Algorithm: Balanced Partition

- Definition. A maximum vertex weight W which satisfies the relation $W > w(V)/2 + \max_{v \varepsilon V}\{w(v)\}$,

  where w(v) = weight of vertex v,

  w(V) = sum of the weights of all vertices

- Balanced partition {A|B} is one having a total vertex weight of at most W, i.e., w(A), w(B) <W.

Differences from KL:

Move only one cell each time.

Cells can have different sizes.

Nets can be multi-terminal.

Maintain a balanced partition after every move.

The FM Algorithm: Overview

- Initialize
  - Start with a balanced partition A, B of G
    (can be done by sorting vertex weights in decreasing order, placing them in A and B alternatively)

- Iterations
  - Similar to KL
  - A vertex cannot move if it violates the balance condition
  - Choosing the node to move:
    pick the max gain in the partitions
  - Moves are tentative (similar to KL)
  - When no moves possible or no more unlocked vertices available, the pass ends
  - When no move can be made in a pass, the algorithm terminates

The FM Algorithm

Start with a balanced partition P = {X,Y}.

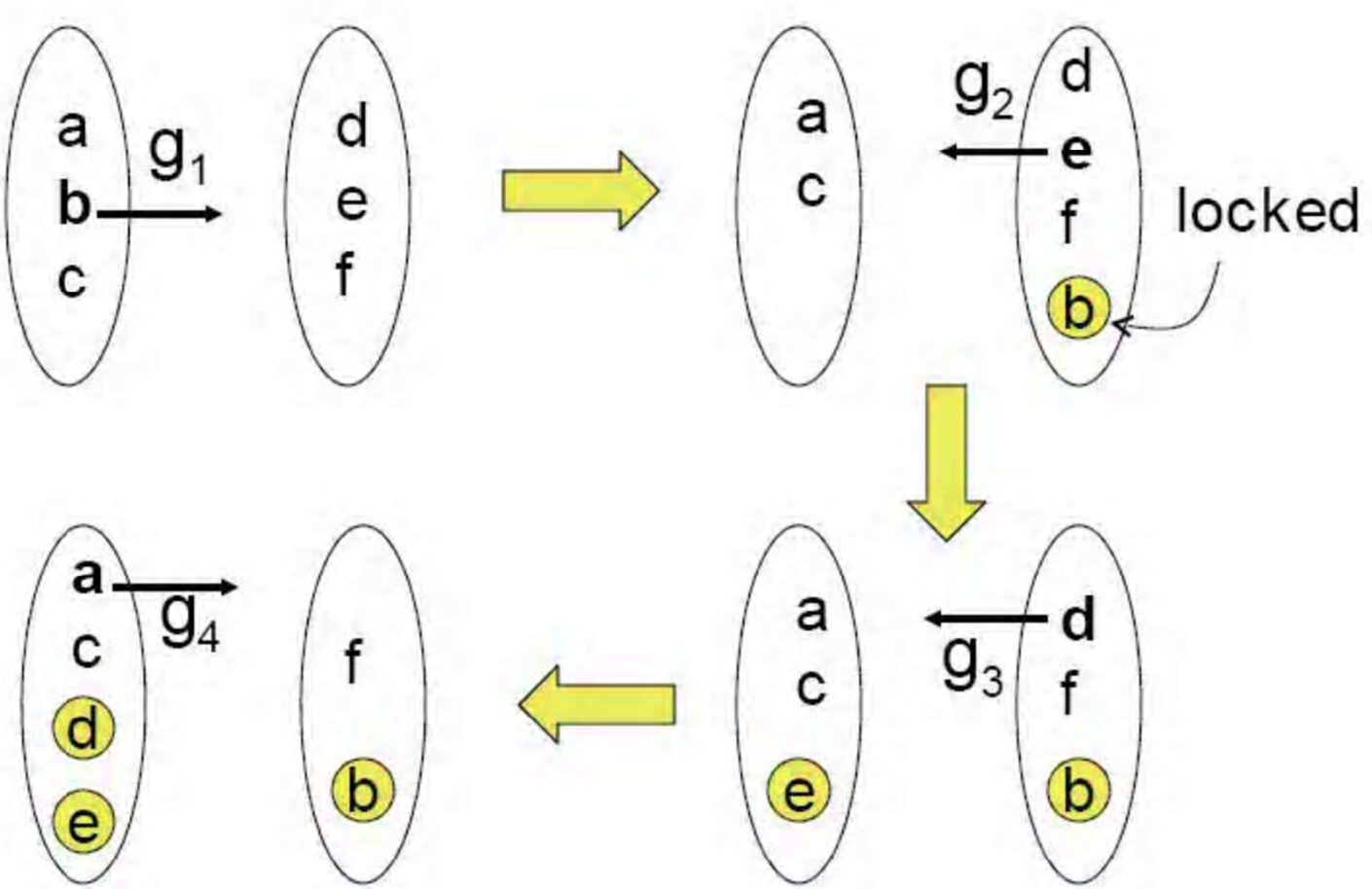
Repeat

- For i = 1 to n:
  - Choose a free cell b $\in$ X$\cup$Y s.t. moving b to the other side gives the highest gain, gain(b), and moving b preserves balance in P.
  - Move and lock b.
  - Let gi = gain(b).
- Find k s.t. G = g1 + g2 + ….. + gk is maximized and shuffle the cells up to this kth step.

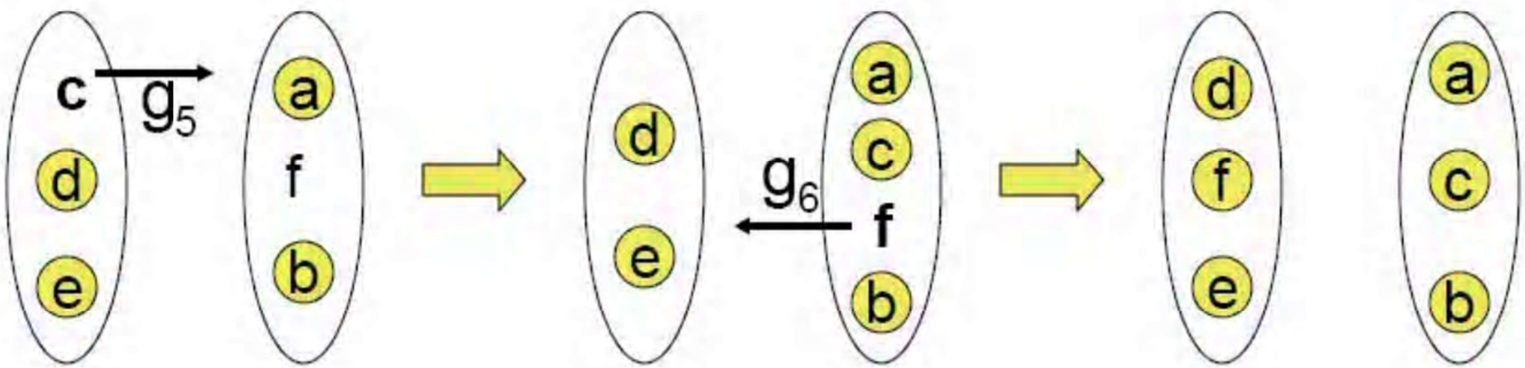
Until G = 0.

An initial balanced partition can be obtained by sorting vertex weights in decreasing order, and placing them in A and B alternatively

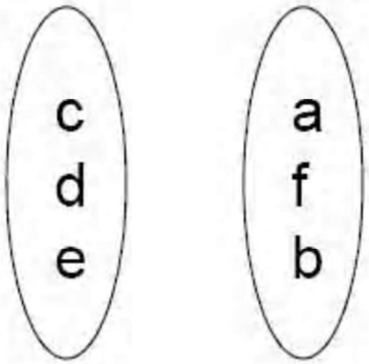
The FM Algorithm: Example



The FM Algorithm: Example



If $G = g_1 + g_2 + g_3 + g_4$ is the largest partial sum, the partition after this pass is:



FM Algorithm: Time complexity

Data structures list A and B are initialized by traversing all hyperedges one by one

Consider the hyperedge $e_a=\{M_1,M_2,.....,M_d\}$, when e_1, e_2, e_3,……,e_{a-1} are already visited

Let num(A) is the no of modules connected by e_a that are in A.

If num(A)=0, then gain of each module M_i in $\{M_1,M_2,.....,M_d\}$ is decreased by $w(e_a)$.

If num(A)=1, the gain of one module M_i in $\{M_1,M_2,.....,M_d\}$ that is in A is increased by $w(e_a)$.

If num(A)>1, no action takes place unless num(B)= 0 or num(B)=1.

The same analysis holds for num(B).

Time spent for processing ea is O(d). Σd= the total number of terminals (t)

Finding the modules with maximum gain in list A and list B takes a constant time, if this value is maintained and updated while processing e_a.

Thus the time complexity of building data structures list A and list B is O(t+n), where n = the number of modules

FM Algorithm: Time complexity

In one pass, find a module with maximum gain using doubly linked pointers

Let a module M_i (in hyperedge e_a) is in A and is moved to B

Case 1: if all modules of e_a are in A (before moving M_i), the gain of all these modules will be increased by $w(e_a)$.

Case 2: if all modules of e_a are in B (right after moving M_i), the gain of all these modules will be decreased by $w(e_a)$.

Time taken to update the gain in each case is $O(d_a)$, where d_a is the number of modules connected to e_a

Since each hyperedge will be considered twice (case 1 and 2), it takes time proportional to number of terminals to update the list

Total time is O(ct), where c= the number of passes (independent of t)

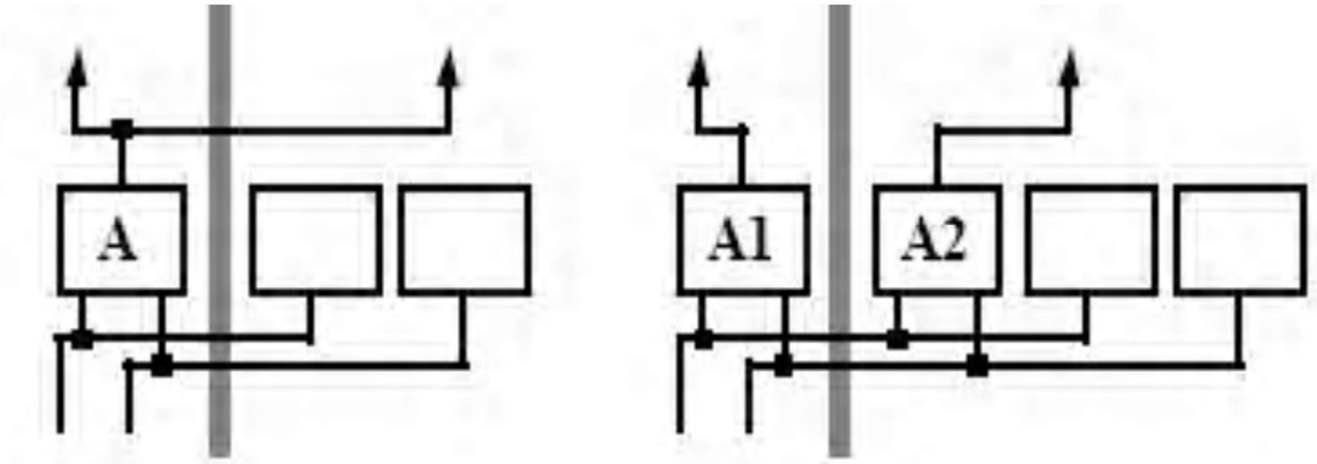
Helpful-Set: A Generalization of FM

- As in FM heuristic, based on local rearrangements.
- Searches for two sets of equal size, one in each part, which will improve cut size if they both change the parts.
- It considers not only single vertices, but also whole sets, to take part in the exchange.

Partitioning using Replication

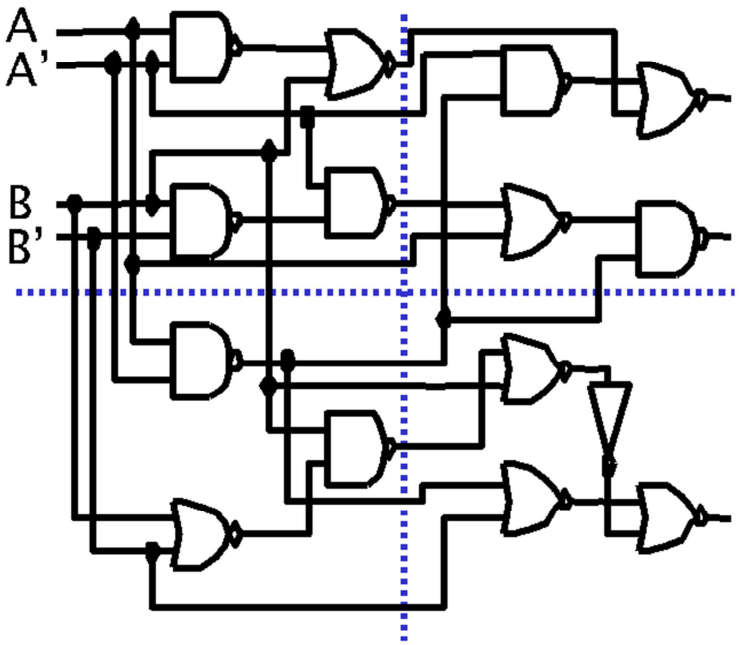
Subgraph Replication to Reduce Cutsize

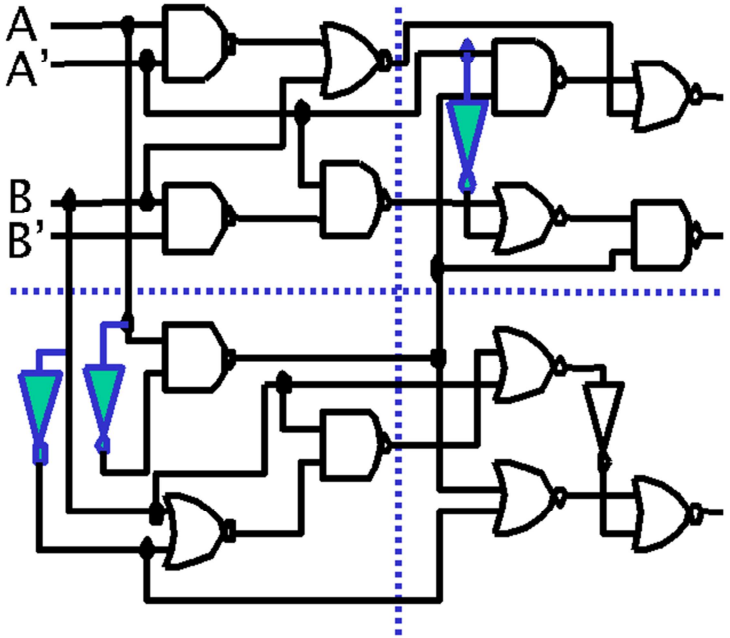
- Vertices are replicated to improve cutsize
- Good results if limited number of components replicated



Node **A** is replicated into **A1** and **A2**, yielding a gain of 1

Subgraph Replication: An Example





Partitioning and Performance

The hypergraph partitioning problem is to divide the nodes of a hypergraph into roughly equal parts; the traditional objective is to minimize cutsize.

In performance-driven partitioning, we also seek to minimize path delay on timing paths.

Motivating Questions

- Can we avoid global timing analysis?
  - Global timing analysis is extremely time-consuming
- Can we improve path delay without significant degrading of cutsize?
  - Need smooth tradeoff between delay and cutsize
- Can we reduce implementation overheads?
  - Previous methods store thousands of critical paths and continuously update them

Partitioning and Delay Problem

If a critical path is cut many times by the partition, the delay on the path may be too large to meet the goals of the high performance systems

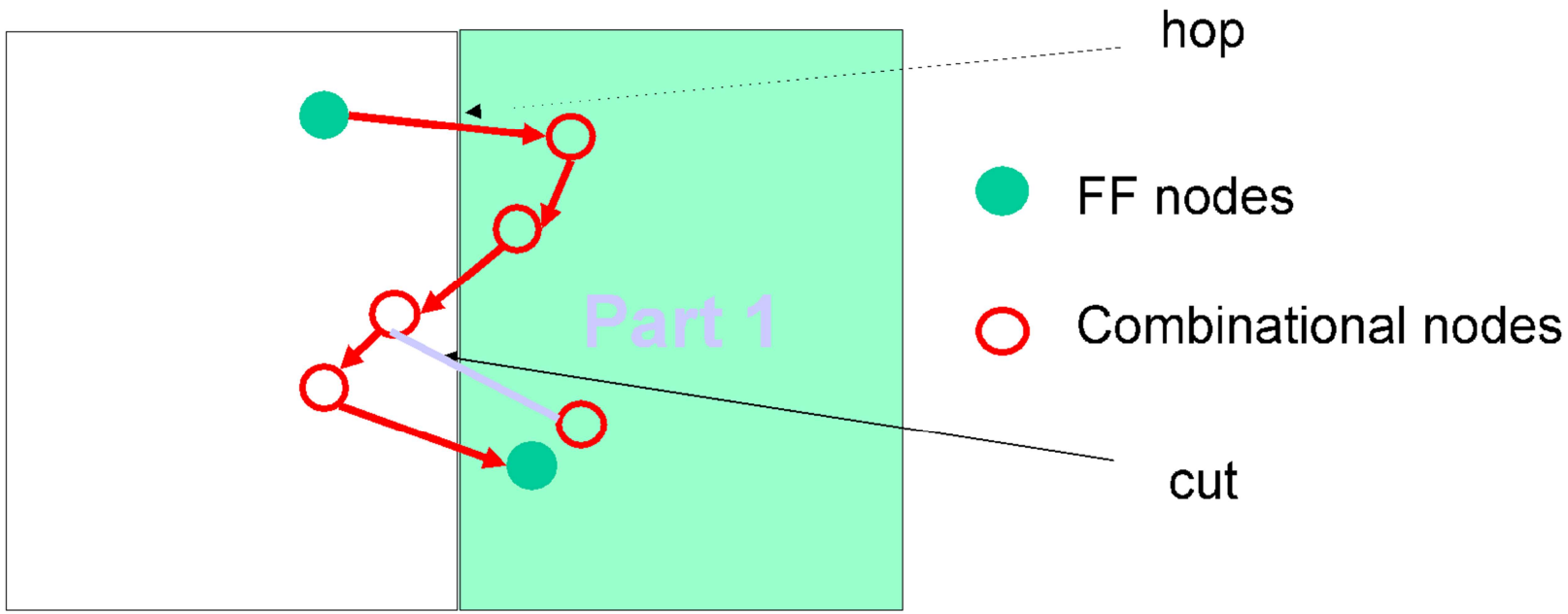
The design of a high performance system requires partitioning algorithms to reduce the cut size as well as to minimize the delay in critical paths

Why Performance Driven Partitioning?

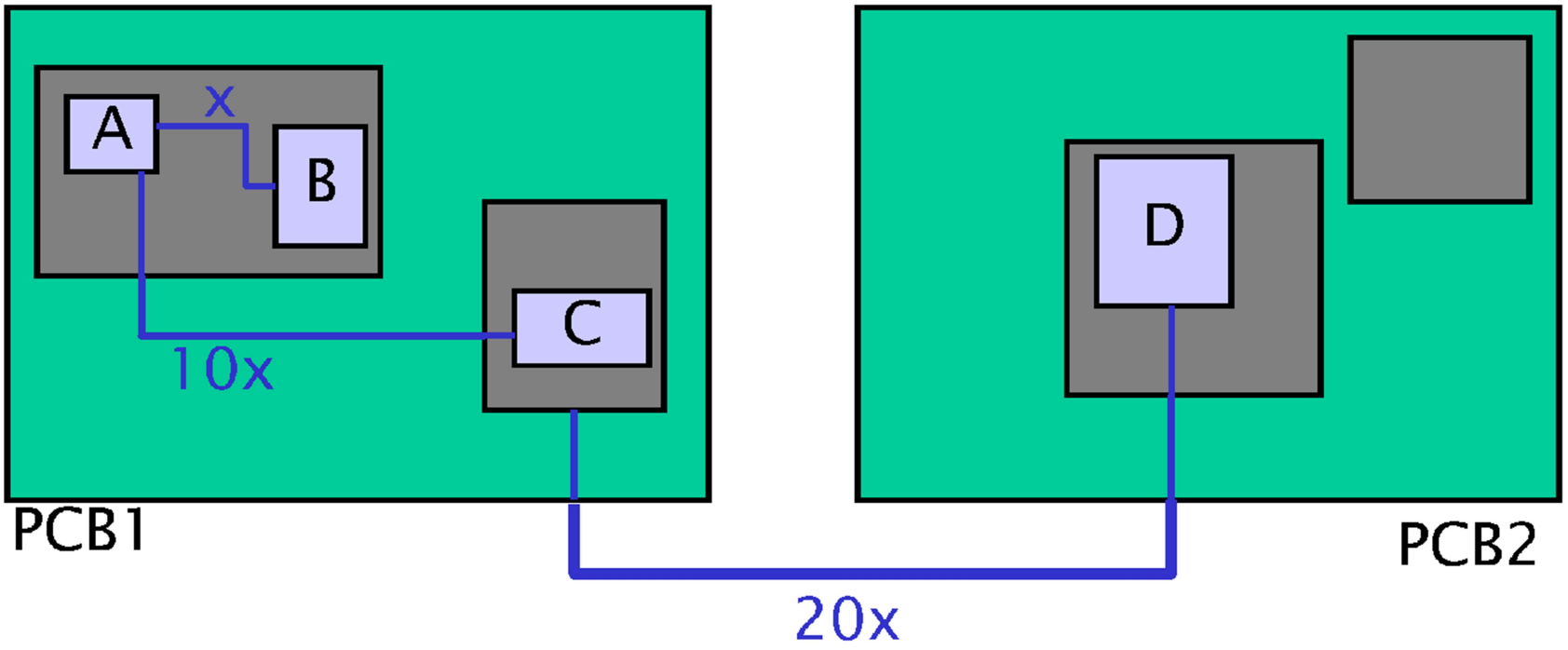
- Achieving timing closure becomes increasingly difficult in deep-submicron technologies due to non-ideal scaling of interconnect delay.
- Routing alone can no longer solve timing problem, even with aggressive optimizations (buffer insertion, buffer/wire sizing,…).
- Timing needs to be addressed at all design stages.
- Partitioning is a critical step in defining interconnect timing properties, but is traditionally driven by cutsize objective.

Delay Model

Delay = hop_delay + node_delay



Delay at Different Levels of Partitions



The on-board delay is three orders of magnitude larger than on-chip delay

Clustering

- Clustering
  - Bottom-up process
  - Merge heavily connected components into clusters
  - Each cluster will be a new “node”
  - “Hide” internal connections (i.e., connecting nodes within a cluster)
  - “Merge” two edges incident to an external vertex, connecting it to two nodes in a cluster
- Can be a preprocessing step before partitioning
  - Each cluster treated as a single node