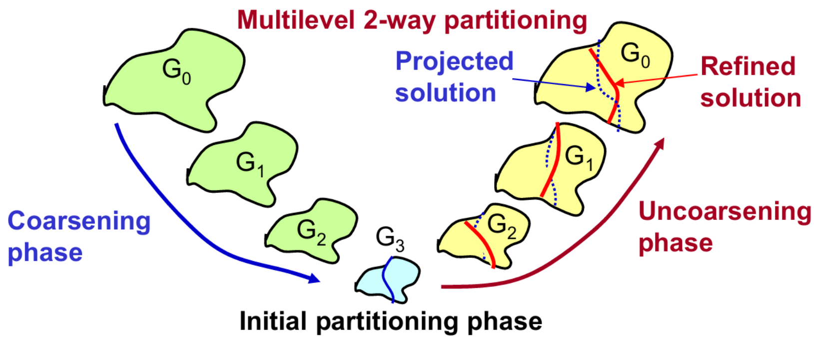
PA1 Report

R13943144 吳紹宇

Method Concept:

I use multilevel partition technique for this programming assignment. I separate the partition into three phases. One phase is “coarsening phase” that I keep using edge coarsening to reduce the cell number until the cell number less than 2500 nodes. Another phase is “initial partition phase” which I perform FM heuristic to partition the small circuit that coarsened from the original one. The other phase is “uncoarsening phase” which I uncoarsening the graph and do FM heuristic for refinement until the circuit go back to its original size. The figure below shows the flow of the multilevel partition.



In this way, we can reduce the input size of the initial partition. Moreover, for the further refinement in the “uncoarsening phase”, we can have fewer iterations due to the better initial partition. Hence, we can get better quality or even lower runtime.

Data Structure:

There are four classes in my code, Node, Cell, Net, and Partitioner.

The Node contains two pointers and an Id. It represents a corresponding cell is used in bucketlist to connect to other nodes. The Cell represents a single cell or a clustering cell group. It contains the corresponding netlist and the number of the cells and so on.

```
class Node
{
    friend class Cell;

public:
    // Constructor and destructor
    Node(const int& id) :
        _id(id), _prev(NULL), _next(NULL) { }
    ~Node() { }

    // Basic access methods
    int getId() const       { return _id; }
    Node* getPrev() const   { return _prev; }
    Node* getNext() const   { return _next; }

    // Set functions
    void setId(const int& id) { _id = id; }
    void setPrev(Node* prev)  { _prev = prev; }
    void setNext(Node* next)  { _next = next; }

private:
    int         _id;    // id of the node (indicating the cell)
    Node*       _prev;  // pointer to the previous node
    Node*       _next;  // pointer to the next node
};
```

```
class Cell // single cell or a cell group

private:
    int             _size;      // size of the cell
    int             _gain;      // gain of the cell
    int             _pinNum;    // number of pins the cell are connected to
    bool            _part;      // partition the cell belongs to (0-A, 1-B)
    bool            _lock;      // whether the cell is locked
    Node*           _node;      // node used to link the cells together
    string          _name;      // name of the cell
    unordered_set<int>      _netList;   // list of nets the cell is connected to
    forward_list<int> _cellGroup; // list of cells of the cell group, store the id of the uncoarsen graph cells
};
```

The partitioner contains a few information about the current circuit, two bucket list data structure and a few stacks to store the coarsening circuits, and so on. For the bucket list, I also implemented the cell insert function and the delete function. The bucket list is implemented using a map. Therefore, we can find the max gain cell in the bucket list simply by getting the reverse_iterator rbegin(). The following code snap shows the interface of the data members of the partitioner and the implementation of the functions relative to bucket lists.

```
class Partitioner

private:
    int                 _cutSize;       // cut size
    int                 _partNum[2];   // cell number of partition A(0) and B(1)
    int                 _partSize[2];   // size of partition A(0) and B(1)
    int                 _netNum;        // number of nets
    int                 _cellNum;       // number of cells
    int                 _maxPinNum;     // Pmax for building bucket list
    double              _bFactor;       // the balance factor to be met
    vector<Net*>        _netArray;      // net array of the circuit
    vector<Cell*>       _cellArray;     // cell array of the circuit
    map<int, Node*>     _bList[2];      // bucket list of partition A(0) and B(1)
    map<string, int>    _netName2Id;    // mapping from net name to id
    map<string, int>    _cellName2Id;   // mapping from cell name to id

    stack<vector<Cell*> > _cellStack;    // stack of cell array
    stack<vector<Net*> > _netStack;      // stack of net array
    stack<int> _cellNumStack;           // stack of cell number
    stack<int> _netNumStack;            // stack of net number

    int                _maxCellSize;   // maximum cell size
    int                _minCellSize;   // minimum cell size

    int                 _accGain;       // accumulative gain
    int                 _maxAccGain;    // maximum accumulative gain
    int                 _moveNum;       // number of cell movements
    int                 _iterNum;       // number of iterations
    int                 _bestMoveNum;   // store best number of movements
    vector<int>         _moveStack;     // history of cell movement

    // Clean up partitioner
    void clear();
```

```
void Partitioner::bListDelete(Cell* cell) {
    int gain = cell->getGain();
    Node* node = cell->getNode();
    int part = cell->getPart();
    Node* prev = node->getPrev();
    Node* next = node->getNext();
    if (prev == nullptr && next == nullptr) {
        // the node is the only node in the list
        _bList[part].erase(gain);
        // _bList[part][gain] = nullptr;
    }
    else if (prev == nullptr) {
        // the node is the head of the list
        _bList[part][gain] = next;
        next->setPrev(nullptr);
        node->setNext(nullptr);
    }
    else if (next == nullptr) {
        // the node is the tail of the list
        prev->setNext(nullptr);
        node->setPrev(nullptr);
    }
    else {
        // the node in the middle of the list
        prev->setNext(next);
        next->setPrev(prev);
        node->setPrev(nullptr);
        node->setNext(nullptr);
    }
    return;
}
```

```
void Partitioner::bListInsert(Cell* cell) {
    int gain = cell->getGain();
    Node* node = cell->getNode();
    int part = cell->getPart();
    if (_bList[part].count(gain) == 0) {
        _bList[part][gain] = node;
        node->setPrev(nullptr);
        node->setNext(nullptr);
    }
    else {
        Node* head = _bList[part][gain];
        node->setPrev(nullptr);
        node->setNext(head);
        if (head != nullptr) {
            // means the the gain is empty
            head->setPrev(node);
        }
        _bList[part][gain] = node;
    }
    return;
}
```

Observation:

The initial partition affects the partition result significantly. For example, if we put the cell with odd Id into part A, the other into part B and perform FM. The final cut number would be about 4000. However, if we go through all the nets, and put the cells in the net we have traversed into part A until part A reaches half of the total cell umber. The result cut number increases to about 6500. Moreover, the initial partition works well for a case does not guarantee to another case. Due to these reasons, I implement the multilevel partition method. In my point of view, the multilevel partition method can have better initial partition of the same circuit size. May lead to better quality. The iteration number of FM at uncoarsening phase can be reduced may further save time.

For the coarsening methods, I have tried edge coarsening and first choice coarsening. I also try different sequences to coarsen the cells. Their performances are close. I finally use edge coarsening. The coarsening order is the max heap order according to the pin number of the cells. Since building max heap is faster than sorting.