Jarred Long

EEE6406

EDA Challenge

04/21/2025

Scalable Partitioning Algorithm for Large-Scale Circuit Graphs

Methods & Introduction

I chose to perform an EDA challenge and implement the Fiduccia-Mattheyses algorithm using python and learned a lot by doing so. I would like to start off by first noting that while the assignment linked and provided examples for the PACE2016 and PACE2017 hypergraph format, every other benchmark used a format with a slight difference, and I chose to use that format instead due to its popularity. In that format, line one the number of hyperedges is put before the number of vertices.

Ex. Where 9 is the number of hyperedges and 10 is the number of vertices

```
9 10
1 2 3 4 5
2 6 7
3 7 8
4 8 9
5 9 10
6 7 8
7 8 9
8 9 10
1 10
```

New Format

```
10 9
1 2 3 4 5
2 6 7
3 7 8
4 8 9
5 9 10
6 7 8
7 8 9
8 9 10
1 10
```

Original Format

I initially used ChatGPT as an AI tool to rapidly develop code. Unfortunately due to the current state of LLM's the code needs to be massively edited and checked before it can work properly. Although the use of tools like ChatGPT make the initial code easier to develop, it can lead to longer debug times. The link to the LLM conversation is commented at the top of the provided python code.

In order to check my implementation I installed a popular partitioning framework called KaHyPar (Karlsruhe Hypergraph Partitioning Framework). Unfortunately KahyPar for an unknown reason does not work on small inputs and actually fails on many small test

cases I provided it, for this reason its use was limited to comparing for large partitions. I tried small self-created and large benchmarks on my implementation to test its success the results are shown below in the results section. The implementation shows the final partitions of A and B and the total number of cut edges in the final partitions. My implementation is designed so the cuts have to be 50-50 if even and one off from that if odd, this can be specified as a parameter of the KahyPar framework too. For this reason the main metrics I will compare is the number of edges cut. The final partitions will be provided for the small test case but for the large ones the partitions are too large to reasonably include in this document.

Results

Test.hgr

My Implementation:

Partition A : {1,4,5,9,10} , Partition B: {2,3,6,7,8}, Edges Cut = 4, Total Gain = 4

KahyPar Implementation: N/A

Test2.hgr

My Implementation:

Partition A : {2,3,6,7,8,12,13} , Partition B: {1,4,5,9,10,11}, Edges Cut = 5, Total Gain = 3

KahyPar Implementation: N/A

Test3.hgr

My Implementation:

Partition A : {3,4,7,8,9,11,12,15} , Partition B: {1,2,5,6,10,13,14}, Edges Cut = 6, Total Gain = 2

KahyPar Implementation: N/A

Pd_rhs.mtx.hgr

My Implementation: Total Gain = 845, Cut Edges = 0

KahyPar Implementation: Cut Edges = 0

gemat1.mttx.hgr

My Implementation: Total Gain = 3401, Cut Edges = 1423

KahyPar Implementation: Cut Edges = 35

ibm01.hgr

My Implementation: Total Gain = 6730, Cut Edges = 1541

KahyPar Implementation: Cut Edges = 202

bibd_49_3.mtx.hgr

My Implementation: Total Gain = 0, Cut Edges = 1176, Every Edge is cut

KahyPar Implementation: Cut Edges = 885

Challenges, Insights and Analysis

We can see that KahyPar did many times better than my implementation and while time was not a metric KahyPar was able to perform its partitioning in ~1 second or less while my implementation took up to a minute or two. Let's dive into why I think this occurs. The FM algorithm is a greedy algorithm and is very good at searching for a local minima very efficiently. Although when the problem size grows larger there can be a large number of local minima that FM can find and converge on, not only this but FM also can get stuck in loops where it cannot make any improvements. These problems are amplified in part due to my implementation for a 50-50 partition. In my code I do not allow FM to unbalance more than one so that it always keeps its partition. This will sometimes make it pass over the most optimal move to maintain an even partition. I solved this loop problem partially by adding a randomization if there are multiple optimal vertexes to choose from, in my implementation if the algorithm gets stuck in a loop it will pick a new random "best" value each time to try and get out of the loop. I was able to see this work well, for certain initial partitions there was always a loop where the algorithm would get stuck, after adding the randomization the algorithm converges on the best solution (in small test cases). This also allows the algorithm to explore different paths if one does not work out, but this exploration is not efficient at all and is a mildly positive side effect of the loop fix, it would take way to long to explore using it as an actual method.

From what I can gather, since FM is a greedy algorithm it is not used solely by most tools as it would cause many problems, and instead tools like KaHyPar use many different algorithms and tricks. For example, KaHyPar uses a coarsening algorithm which greatly reduces the problem size so the following algorithms can work much quicker. It also does not use a purely random initial partitioning like I used for basic FM, instead it has an algorithm dedicated to start with promising partitioning. It than uses more complex FM

algorithms like two way FM for local search when initial partitioning and a final FM based algorithm for its local search. Using all these techniques in combination with good coding practices and a fast language like C++ can lead to a much higher quality solution in a lot less time than a slow language implementation of a simple version of FM. As for my implementation I used some methods to cut down on runtime, credit to ChatGPT for the ideas but not for implementing them very well. I only calculate updates for neighbors of the nodes that are moved and use buckets to contain the gain of each of the nodes. I also changed it so instead of copying the partitions when updating the best moves, which could take a long time, I just keep track of the moves and update at the end of a cycle using the moves that lead to the best partition. As for the bad things there are much better data structures that could be used to speed up the processes and I sort the gain buckets each iteration which takes a lot of time. A better method I would say would use better data structures and not sort the buckets before each calculation, but realistically its not using python. After doing this in python I think that C++ is much better suited for this challenge if trying to truly achieve a working tool. Although for prototyping python works well.

Another interesting observation is bibd_49_3.mtx.hgr, where my implementation cuts every edge. When looking at the file we can see each hyperedge has a large amount of vertices. My theory is that the initial partitioning of this large file is more likely than not to split all the hyperedges and the algorithm would need to explore the movement of multiple nodes (hundreds maybe?) before it found a move that caused a positive gain. Since the basic version of FM does not explore that far because of it being greedy is just converges where it started.

Conclusions

My implementation is proven to correctly implement the FM algorithm in its most basic form. It works very well for small sets of data but scales poorly due to many limitations discussed in the previous section. This goes to show how complex, and the underlying understanding required to effectively solve these sorts of problems with high efficiency. The base FM algorithm does not seem to solve the problem well alone but can be an important part of a more complex framework.