ECE5960 Physical Design Algorithms Programming Assignment 1 Report

Cheng-Hsiang Chiu
Department of Electrical and Computer Engineering
University of Utah, Salt Lake City, UT-84112
{cheng-hsiang.chiu}@utah.edu

I. Encountered Challenges

The challenge we encounted was the long runtime. We used `gprof` to profile the executions and found out that the operation of updating the gains after a base cell was moved was the bottleneck. To reduce the runtime, we came up with two improvements. One is to implement the operation of updating the gains using the `critical net` idea [1]. The other is to avoid the repetitive iterations of locked cells in the bucket data structure while locating the next base cell.

II. Ways to Overcome Challenges

To overcome the challenge, we used `gprof` to profile the exectuion and realized the bottleneck was the operation of updating the gains after a base cell was moved. There are two improvements we did in the programming assignment.

The first improvement is to adopt `critical net` idea introduced by the authors of FM algorithm [1]. Our initial implementation was to calculate the `fs` and `te` of each cell. The complexity of this way is $O(N^3)$. The `critical net` technique could reduce the complexity from $O(N^3)$ down to $O(N^2)$.

The second improvement is to avoid iterating the locked cells in the bucket data structure. Our initial implementation of updating a cell between linked lists to reflect the gain change did not exclude the base cell. That is, we also updated the base cell between linked lists in the bucket. We ended up with iterating many locked cells while finding the next base cell. The current implementation removes the locked cells in the bucket. In this way, the number of cells in the bucket is decreasing gradually.

III. Availability

The whole implementation is available in the Github repository, link.

IV. Build and Run Executable

There is a README in the uploaded compressed file and in the Github repository mentioned above. README provides very detailed information. In this section, we highlight the instructions to build and run the executable `fm`. Please make sure CMake and clang++ is installed.

To build the executable `fm`, use the following commands,

```
mkdir build
cd build
cmake ../
make
```

To run the executable with an input file, use the following commands in folder `build`,

```
./fm [input_file] [output_file]
```

To verify the correctness, use the following commands in folder `build`,

```
../checker_linux [input_file] [output_file]
```

In addition to running `fm` with one benchmark, we provide a script `run.sh` to run `fm`, measure the runtime and verify the correctness with all of the benchmarks. The script will be copied to folder `build` automatically when building `fm`. In folder `build`, simply use the following commands,

```
chmod 744 run.sh
./run.sh
```

Input	Cutsize	Runtime (second)
Input_0.dat	43360	1.865
Input_1.dat	1667	0.012
Input_2.dat	2957	0.025
Input_3.dat	35075	0.59
Input_4.dat	61866	1.311
Input_5.dat	187937	4.595
Input_6.dat	2	0.001

TABLE I: Cutsize and runtime of each benchmark.

V. Results

The initial partion we used is a random partition. The executable is built using clang++ v10.2 with -std=c++17 and -O3 on a Linux machine with Intel i7-9700K 8 Cores at 3.6 GHz and 32 GB RA. All results are obtained in a single run.

Table I shows the cutsize and runtime for each benchmark. The executable runs multiple passes for each benchmark. When the maximum prefix gain is zero in a pass, the executable will not run the next pass.

References

[1] C. M. Fiduccia and R. Mattheyse, "A linear-time heuristic for improving network partitions," *Design Automation Conference (DAC)*, 1982.