Chip Routing In Haskell

James Mastran (jam2454)
Advised by Professor Stephen Edwards
Columbia University

December 2025

1 Abstract

In this paper, we explore solving the chip routing problem in Haskell using the famous A* algorithm inside an algorithm devised in 1995 called **"Pathfinder/Negotiated Congestion"**. We also investigated optimizations, for both sequential and parallel execution, and their impacts on solving an example board as measured through runtime and total wire used in the solved routing. The sample problem we used is a HDMI-to-USB chip represented by a KiCAD S-expression file. While this was a difficult problem to parallelize, there are interesting methods that we employed whose results and trade-offs are discussed throughout this paper.

2 Chip Routing

Chip routing is the process of determining how best to connect components on a PCB, FPGA, or VLSI board/chip. There is a lot to consider while planning the routing, such as preventing shorts between wires, minimizing wire delay, among other metrics. Chip routing is a crucial concept in chip manufacturing and assembly and is important due to the ubiquitousness of chips that has only been hastened by the latest AI boom.

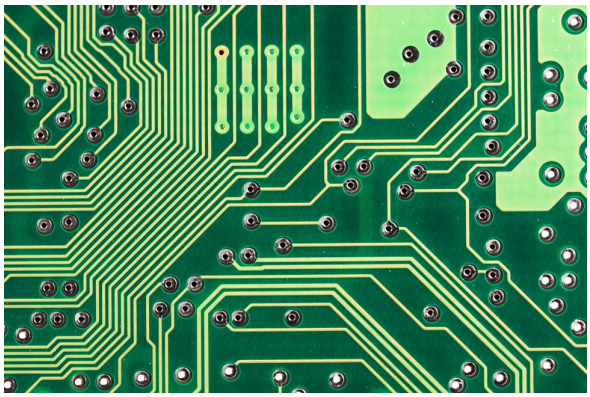

Figure 1: *Manual or interactive routing, advantages and inconveniences*

Identifying how to accomplish all required connections on a chip with tiny wires can be a difficult task, especially since there are many other connections and components. While it ranges, there can be billions of connections on a single chip and thus, in practice, chips often have multiple layers to make solving this problem possible. These connections and components act as barriers for other connections, which is the crux of the problem.

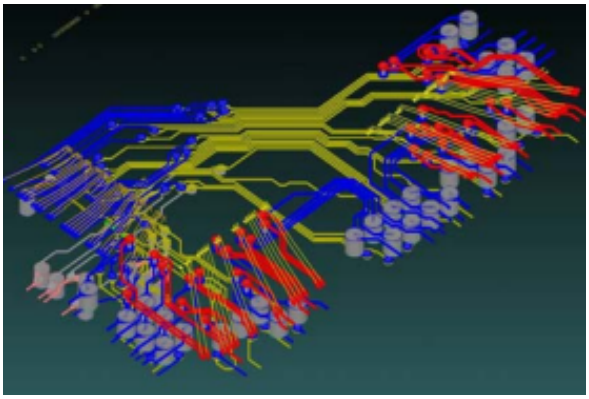

Figure 2: *Electromagnetic and Circuit Co-Simulation and the Future of IC and Package Design*

2.1 Key Terms

The following terms are used to describe chip routing. First, the chip routing problem can be seen as solving a **"netlist"**, which is a list of nets. A **"net"** contains a list of **"pads"**, or components, to be connected. Connections between pads within a net can short, but connections between different nets act as barriers and must avoid each other. Thus, this problem can be seen as maze routing with dynamic barriers.

3 Objective

While routing, there are many objectives that may be prioritized:

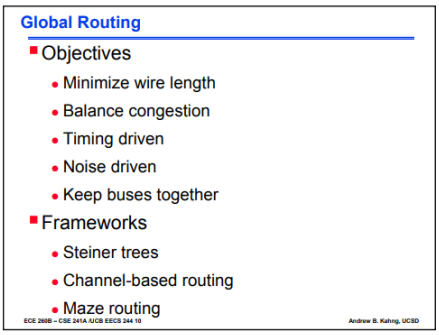

Figure 3: *Routing in Integrated Circuits*

To simplify our project, we are specifically concerned with producing a valid set of connections/segments and investigating ways to speed that process up. A valid set of connections means that we connect all required pads and do not have shorts between inter-nets; only shorts within a net are allowed. A short is an unwanted overlap of wires that must be avoided so the chip functions as expected.

As a fun exercise, we will also be recording and comparing quality of each method by measuring total wire length used in each solved result.

4 Collecting Data & Parsing KiCAD

To begin investigating chip routing, we first had to collect a dataset that would be realistic and challenging enough to test our implementation. We used the dataset found at `https://github.com/timvideos/HDMI2USB-numato-opsis-hardware/blob/master/board/HDMI2USB.kicad_pcb` which is a KiCAD description of an HDMI to USB board:

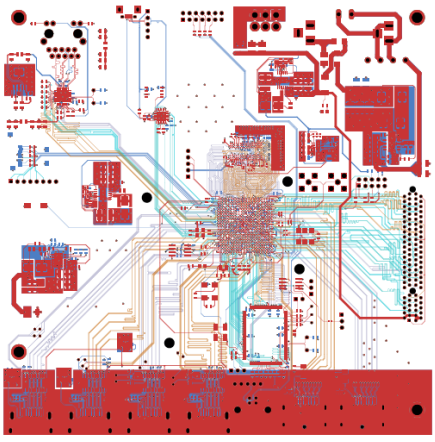

Figure 4: As provided by Professor Edwards; KiCAD Rendering of the HDMI to USB board's S-Expression

The provided file uses S-expressions and nested elements to describe the board. S-expressions are used to provide a rich representation of the board, providing a lot of information.

The relevant components of the KiCAD's S-Expression include:

1. Footprints/modules (depending on the KiCAD version)
2. Pads
3. Nets

We parsed the file's S-expression in Haskell, using the datastructure below. Once populated from processing the file's contents, we walked the datastructure, finding all the pads that belong to each net by iterating through all the footprints. A footprint/module provides a list of net/pad pairs which tells us that a pad belongs to a particular net. Doing so gives us a representation of a netlist.

```
data SExpr = Atom String | SExpr [SExpr]
```

Each footprint has a global/absolute position, (x_f, y_f, θ_f), and the pads within have a relative position to the global, (x_p, y_p). Using this information, we are able to convert the list of pads to a list of coordinates:

$$(x, y) = (x_p{*}\cos\theta \; - \; y_p{*}\sin\theta, \;\; x_p{*}\sin\theta \; + \; y_p{*}\cos\theta)$$

where we convert θ to radians: $\theta = \theta_f * \frac{\pi}{180}$ (as per Wikipedia Contributors, *Rotation matrix*).

As a part of the KiCAD parser, we also allow different grid scalings. For this project, we scaled the data to a 0.1mm grid where each edge is 0.1mm length.

We also had to consider the layer of the pads. Some pads can belong to multiple layers (e.g. *.Fu or *.Cu) or they may only belong to a single layer (either the TOP or BOTTOM layer). After parsing the KiCAD file, we were able to draw the shortest paths between necessary points using SVG:

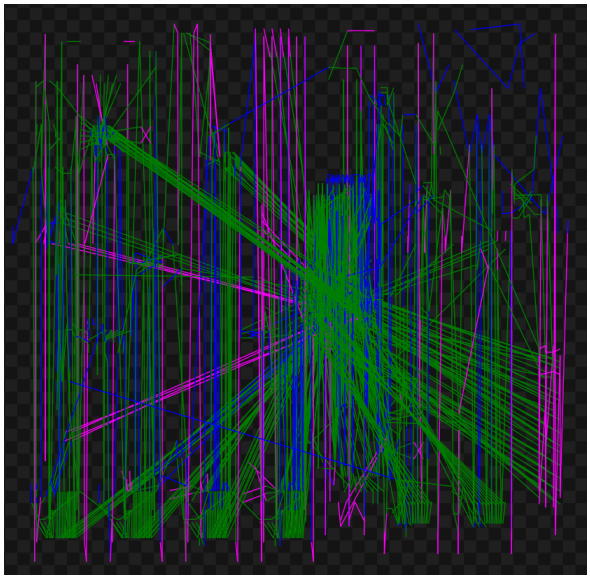

Figure 5: SVG Representation of Points to be Connected

The noticeable similarity between the SVG representation and the KiCAD rendering is a positive sign that our parser is working as intended. There is one notable difference in the bottom right most area of the SVG. The components located there appears to be lower than the KiCAD's rendering, but was insignificantly different and therefore left as-is.

The different colors represent the different layers that the connections belong to. Green is on the TOP layer, blue is on the BOTTOM, and magenta represents pads that belong on multiple layers.

For simplicity, we removed any points that would result in obvious shorts, even if the layer it belongs to is ignored. A subset of the data we parsed is shown below:

hdmi.cleaned.txt

kicad-parser > output > hdmi.cleaned.txt

```
1   1 2093 1131 /DDR3/DDR0_A0 TOP
2   1 2120 1534 /DDR3/DDR0_A0 TOP
3   1 2247 1138 /DDR3/DDR0_A0 BOTTOM
4   10 2085 1171 /DDR3/DDR0_A4 TOP
5   10 2100 1524 /DDR3/DDR0_A4 TOP
6   10 2183 1093 /DDR3/DDR0_A4 BOTTOM
7   100 1909 1195 /FPGA_Bank_0_3/DEBUG_IO0 *.Cu
8   100 2230 1534 /FPGA_Bank_0_3/DEBUG_IO0 TOP
```

Figure 6: Parsed text file

This informs us that points $(2093, 1131, 1)$, $(2120, 1534, 1)$ $(2247, 1138, 0)$ belong to net 1, the */DDR3/DDR0_A0* component, and all must be connected. Similarly, the next 3 lines show that net 10 must connect $(2085, 1171, 1)$, $(2100, 1524, 1)$ and $(2183, 1093, 0)$. The last 2 lines show a pad at $(1909, 1195)$ that belongs to both layers, but for simplicity we will assume it to be on a single layer (TOP). This content acts as input to the chip routing solver project.

This is the sample dataset that we use to benchmark and check our implementation of chip routing against. The difficult task is now figuring out how to arrange these connections avoiding any intersections between net's wiring.

5 The Chip-Routing Algorithm & Implementation

In order to solve the chip routing problem, we followed the Pathfinder/Negotiated Congestion (NC) algorithm presented in "PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs" (1995):

Algorithm: Negotiated Congestion (NC)

```
While shared resources exist (global router)                    [1]
  Loop over all signals i (signal router)                       [2]
    Rip up routing tree RT_i                                    [3]
    RT_i <- s_i                                                 [4]
    Loop until all sinks t_ij have been found                   [5]
      Initialize priority queue PQ to RT_i at cost 0            [6]
      Loop until new t_ij is found                              [7]
        Remove lowest cost node m from PQ                       [8]
        Loop over fanouts n of node m                           [9]
          Add n to PQ at cost c_n + P_im                        [10]
      End                                                       [11]
    End                                                         [12]
    Loop over nodes n in path t_ij to s_i (backtrace)           [13]
      Update c_n                                                [14]
      Add n to RT_i                                             [15]
    End                                                         [16]
  End                                                           [17]
End                                                             [18]
                                                                [19]
```

Figure 7: The NC Algorithm

The chip routing problem is solved by iteratively calling a negotiation loop until all paths calculated do not have intra-net intersections or shorts. Within an iteration of negotiation, there is short-term, "present usage", and long-term, "historical usage", memories.

Present usage is dynamic within an iteration and is initially set to 0 for each node at the start of each negotiation iteration. It keeps track of which nodes have been used by previously solved nets in the iteration. Its purpose is to inform subsequently calculated nets to try and avoid nodes that have been claimed by other nets already.

There is a subtle difference between present usage and the actual cost it contributes. Present usage counts the number of nets using each node within an iteration of NC. The cost is calculated by subtracting the capacity from the present usage of a node:

$$pres\_cost(n) = \max(0,\ pres\_usage(n) - capacity(n))$$

Since we are seeking node-disjoint paths between nets, applying costs at the node-level, rather than at the edge-level, ensures that the routing will converge towards using a single node once per net. Also, to ensure node-disjointness, we set the capacity of each node to 1.

The historical component is a cumulative sum of present cost and is updated at the end (or beginning) of each negotiation cycle. This factor is static within an iteration of NC and keeps track of heavily used paths over iterations.

This short-term and long-term memory dictates the cost of using each node and ultimately drives the algorithm to convergence.

For our problem, the base cost of a node is 1, though in other applications there can be a delay factor for vias (movements between layers) and using different wires, etc..

Using these static and dynamic costs, A* is used to find all paths between required pads.

The equation below describes the cost of each node and is ultimately what defines the NC algorithm:

$$\begin{aligned} cost_n &= (base\_cost_n + hist\_cost_n) \ \cdot \ (pres\_cost_n + 1) \\ &= (1 + hist\_cost_n) \ \cdot \ (pres\_cost_n + 1) \end{aligned}$$

The algorithm is summarized by the flowchart below:

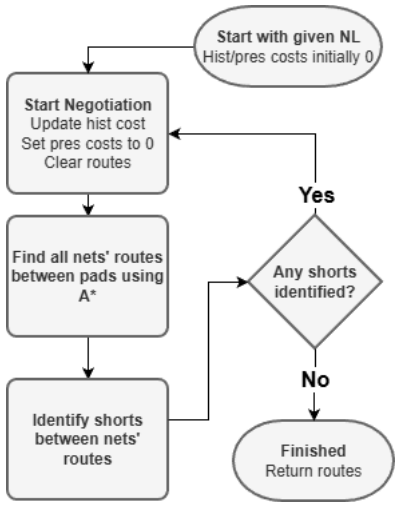

Figure 8: NC Flowchart

5.1 Our Representation

To represent the problem in Haskell, we defined a few types to help explain the problem.

To store costs, we defined a *CostM* type which maps nodes' integer representations of their coordinates to its cost as a *Double*.

```
type Cost           = Double
type CostMKey       = Int
type CostM          = Map.Map CostMKey Cost
```

To explain the problem we are trying to solve, the following types were defined:

```
type Connection     = (Coord, Coord)
type Net            = [Connection]
type NetList        = [Net]
type Problems       = [NetList]
```

These terms have already been explained earlier except for **"Problems"**. Since a netlist is the problem we are trying to solve, it follows that a group of netlist would be "problems" (which is utilized for the batch and parallel variations). For completion, a **"Connection"** is a pair of coordinates, representing pad locations. A **"Net"** is a list of connections, giving an entire list of pad coordinates to be connected within a net. Lastly, a **"NetList"** is a list of nets.

To describe an answer for routing a chip, we also defined the following types:

```
type Segment        = [Coord]
type SegmentList    = [Segment]
type Routing        = [SegmentList]
```

A **"Segment"**, the solution to a connection, is a list of coordinates, representing the solved path of a wire. A segment describes how pads, specifically a single pair of pads, are connected within a net. The layout of a segment is simple, with each coord followed by another that is a unit step away. A **"SegmentList"**, the solution to a net, is a list of segments representing connections between pads within a net. Finally, a **"Routing"** is the answer to solving all the connections within the netlist. The routing contains all the segments for each net in the netlist and is ultimately our answer.

5.2 NC with the A* Algorithm

As used in our implementation of NC, we first implemented the well known A* algorithm. Its purpose is to solve a connection, producing a segment. Since our board is a 3 dimensional grid, we used the 3D Manhattan distance as a heuristic to estimate which order to explore nodes utilizing a priority queue. In addition, seeing that we don't know the actual cost from an unexplored node to the target, we use the heuristic to estimate it. The node that has the minimum cost in the priority queue is the next to be

explored, as based on the node's cost described earlier and the heuristic below (Wikipedia Contributors, *A* search algorithm*). We explore nodes until the target node is reached.

$$h(coord_1, coord_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

While our implementation used an implicit graph, we can conceptualize our problem as a NxM grid like so, but with 2 layers stacked one on-top the other:

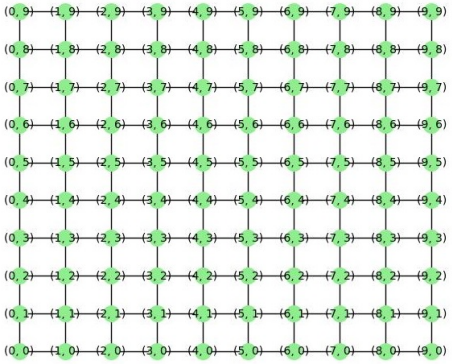

Figure 9: *Graph Theory - Infinite Graphs*

On the grid, diagonal movements are not allowed; only movements in the horizontal/vertical directions and jump between the layers is allowed. Leveraging these rigid rules of our graph, we did not have to use a adjacency matrix, list, or node/edges explicitly. At each node (x, y, l), there are only (5) choices:

1. $(x + 1, y, l)$
2. $(x - 1, y, l)$
3. $(x, y + 1, l)$
4. $(x, y - 1, l)$
5. (x, y, l'), where $l' = \bar{l} \in \{0, 1\}$

Using the A* implementation, we can solve a net by consecutively giving the A* algorithm each connection in a net, which results in a segment list.

To solve the full netlist, we use the process above on each net, but each net uses the present cost/usage as determined by previously solved nets in the iteration.

NC works by iteratively repeating this process on the netlist until all shorts are resolved, updating the historical cost at the end of each "negotiation" iteration by adding present cost. After the historical cost is updated, the present usage and previously calculated routes are cleared for the next iteration to begin.

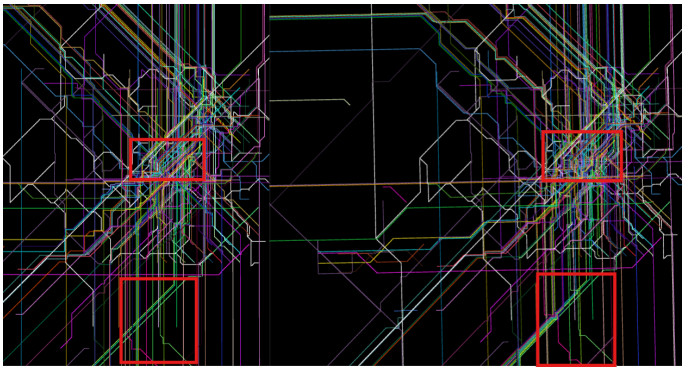

Figure 10: Results after 1 iteration (left) and 2 iterations (right)

The above image shows some routing after one and two iterations, respectively, showing how they make progress between iterations. As highlighted in red boxes, it can be observed that wires sprawl out more in the second iteration's results. This helps show the algorithm working to increase separation, reducing overlaps, from iteration to iteration.

6 Sequential Implementation

When the HDMI to USB output from our KiCAD parser is provided as input to our chip solver, the following X3D images are produced, displaying the solved routing:

Figure 11: Solved chip routing on the HDMI to USB board

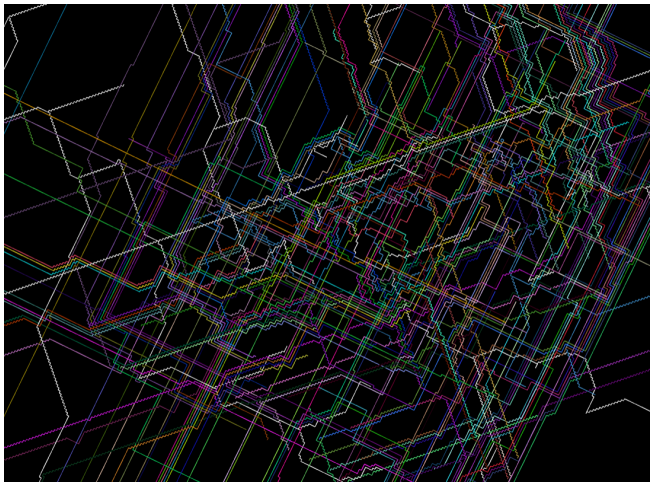

Figure 12: Chip routing zoomed-in

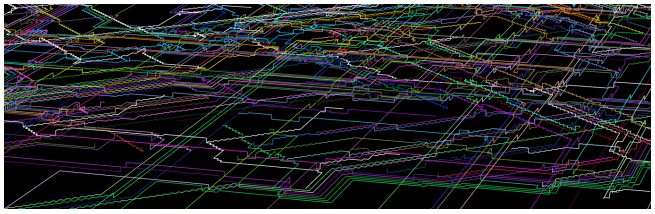

Figure 13: Angled chip routing to see via/layers movement

Due to timing concerns we used only 300 out of the 400 nets, which is plenty difficult. It takes about 5.5 minutes to sequentially solve the above routing with some required optimizations. This is a subset of the data, but it contains most of the nets while limiting the TOP layer nets to 3 pads while those on the BOTTOM layer can have up to 400 pads.

6.1 Tuning and Sequential Optimizations

6.1.1 Tuning Parameters

The naive cost function was described earlier:

$$c_n = (1 + hist\_cost_n) \cdot (pres\_cost_n + 1)$$

For our setup, we weighted each of these differently with some scaling factors **pf** (present factor) and **hf** (historical factor) as suggested by "PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs". We left *hf* as a constant value but *pf* increases by a factor of *pfInc* each time until it hits *pfMax*. These are values that can be provided via command line arguments and should be tuned to specific boards. Therefore, the cost function we used was modified to:

$$c_n = (1 + hf \cdot hist\_cost_n) \cdot (pf \cdot pres\_cost_n + 1)$$

This ensures that the present cost's contribution increases as the historical costs monotonically increase from one iteration to the next.

These values directly impact how quickly the problem converges. It takes some effort to tune the values based on specific inputs. For the output we have presented in this report, the best values for the sequential solution were:

```
pf    = 1.0  // initial pf
pfInc = 1.5
pfMax = 1000 // really no max
hf    = 1.0
maxT  = 2    // 2 max connections per top
maxB  = 400  // 400 max connecs per bottom
```

We will make clear the exact values used for each method described throughout the paper.

Next, we will discuss some of the sequential optimizations that were explored (some inspired from "PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs").

6.1.2 Net-level Optimization

Rather than a random order of pads to be connected, we compute an ordering that minimizes the length between connections, using the ability to short within a net to our advantage. For example, connecting $(1, 1)$ to $(1, 5)$ to $(1, 10)$ to $(5, 5)$ does not make as much as sense as $(1, 1)$ to $(1, 5)$ to $(5, 5)$ and $(1, 10)$ to $(1, 5)$, as depicted below:

Figure 14: Non-optimized vs optimized nets

6.1.3 NetList (NL) Order Optimization

The ordering in which nets are solved on each NC iteration also impacts convergence and runtime. Solving difficult nets first, as deemed by the size of their bounds/area, and allowing them to claim paths first before shorter nets proved to speed up the runtime.

6.1.4 Batched Optimization

One other optimization that came to light while working on the parallel solution is that simply batching nets into groups and running sequentially can have speedups. Batching works by finding isolated netlists that can be solved independently—these are "Problems". Any nets that do not belong to a netlist

at the end of batching are put into a special set of nets that are processed first. After solving each batch, their usages are passed to the subsequently solved batches. This process is described more in-depth in the following sections.

7 Parallel Solutions

The Negotiated Congestion algorithm is inherently sequential due to the necessity of the short-term memory passed between solved nets within a single iteration. Therefore, it was difficult to parallelize.

For all the variations detailed below, the A* algorithm is able to be parallelized since each connection is solved independently within a net and manages its own priority queue. All variations leveraged A* in parallel for each net with at least 4 pads, which contributed to slight speedups (about 3s).

Additionally, for all methods, the *map/'using' parList rdeepseq* parallelization strategy was used to calculate A* connections within a net and to process nets in a given netlist in parallel. To show this, the following code snippets compare how solving connections within a net looks for the sequential vs parallel solutions. First, the sequential version is provided:

```
aStarNetH :: H -> CostM -> CostM -> Args
          -> Net -> SegmentList
aStarNetH _ _ _ _ [] = []
aStarNetH h om hcm args (n:nl)
  = solvedConn : aStarNetH h om hcm args nl
  where solvedConn = aStarConnH h om hcm args n
```

And the parallel version:

```
parAStarNetH :: H -> CostM -> CostM -> Args
             -> Net -> SegmentList
parAStarNetH h om hcm args nl =
  map (aStarConnH h om hcm args) nl
    `using` parList rdeepseq
```

** aStarConnH is the A* algorithm.* The solution for solving nets in parallel is very similar and can be investigated in the code provided at the end of the paper.

7.1 Variation One: Parallel Netlists

The first parallel variation we attempted was based on the idea that if we can find a bunch of isolated nets within the entire netlist, we can group them as independent sub-problems/sub-netlists to be solved in parallel.

Nets that do not belong to a sub-netlist at the end of partitioning are placed in a special netlist group that is solved first. This special group's present usage is then passed to all the other netlists, which are then solved in parallel, since all the other netlists do not overlap.

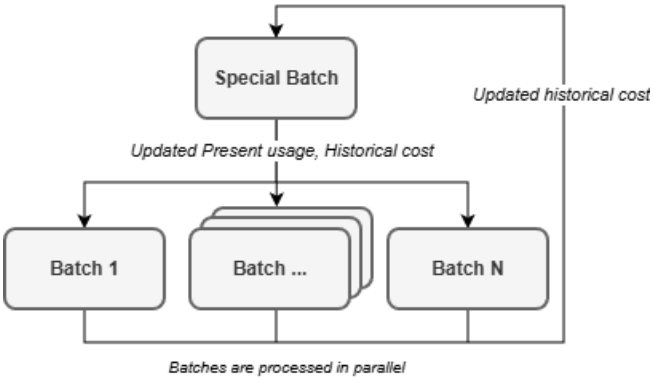

Figure 15: Parallel Netlists method's architecture

To find all our "problems" or list of netlists, we sliced the entire board into NxM slices. The specific choice of the slice size greatly impacts load balancing and parallelism. It takes some effort to tune N and M to the proper values based on the specific board. Once this slice size is determined, we look at each slice to see which, if any, nets fit perfectly within. All nets within a slice can be thought of as an independent netlist to be solved in parallel.

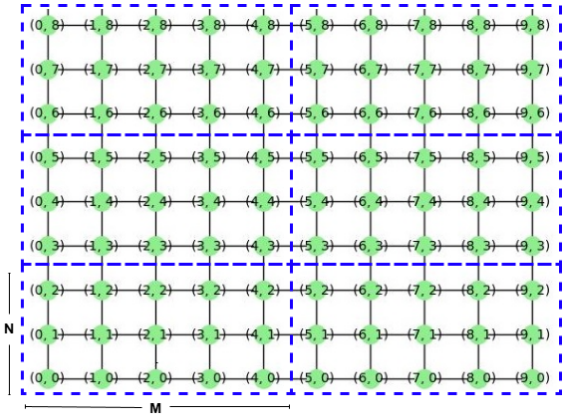

Figure 16: Grid with NxM slices

As it turns out, this process did not result in many groupings. To overcome this issue, we introduced an acceptable amount of overlap. This overlap allows nets that mostly fit within a slice to be a part of that slice, even if it doesn't fit perfectly.

For our solution, we used a varying NxM slicing and varying overlap based on the iteration count. While the solution provided is deterministic, this allowed us to bypass some of the fine tuning and allows some form of "randomness" to help lead to a valid routing.

This is where the idea for using batching as a method of speeding up sequential computation originated.

We used the following parameters for this variation:

pf	pfInc	hf
1	1.5	1

7.2 Variation Two: Seq-Par Alternation

The second parallel variation takes a different approach by alternating between NC iterations that are solved sequentially and in parallel. The present usage from the immediate previous sequential iteration is passed to the following parallel iteration. This helps convergence since the parallel iteration is provided an outdated snapshot of the previous iteration's usage, which is better than an absence of this data altogether. For the parallel case, random batches of nets are solved in parallel. This method allows parallel iterations to explore solutions on small batches of nets quickly while the sequential iteration can use that historical information, as passed through the historical cost, to find non-intersecting paths.

We used a batch size of 25, meaning that on 300 nets we would have 12 batches. The way these are processed in an iteration of negotiation is as follows:

1. Initially start with present usage from the previous sequential execution
2. Take the batch at the front of the list
3. Process the nets in parallel
4. Update present usage for the next batch
5. Repeat steps (2-5) until all batches are processed

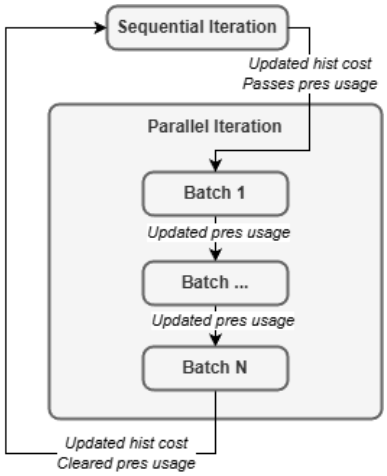

Figure 17: One alternating seq/par iteration

After a parallel run, a normal sequential negotiation iteration would occur. After 15 iterations, the algorithm switches to running only sequential iterations until a valid routing is found, which was necessary for stability in convergence.

We used the following parameters for this variation:

pf	pfInc	hf
1	1.55	1

7.3 Variation Three: Non-Overlapping Batches

For the last variation attempted, we followed a suggestion cited in "Accelerate FPGA Routing with Parallel Recursive Partitioning" for batching non-overlapping nets together and running these batches in parallel. If nets do not have bounding boxes that overlap, then there is little chance they will conflict and as a result they should be able to be safely run in parallel. This is a similar setup to the previously discussed variations, but with slightly different partitioning.

We once again have a group of nets that do not fit nicely in any batch. This special group is processed first on each NC iteration, processing each net in the batch sequentially. This is because each net requires the present usage of the nets calculated before since the nets in this batch may overlap. The resulting present usage cost map is passed to the following batches.

For all other batches, all of the nets' routings are calculated in parallel, one batch at a time. Each consecutive batch uses the present usage information provided by previously calculated batches, updating the usage for subsequently executed batches. The following diagram summarizes a iteration of NC with this setup:

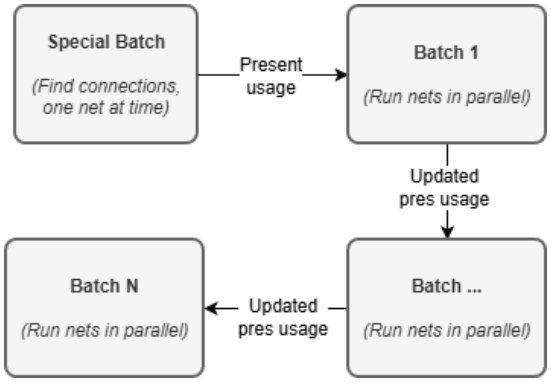

Figure 18: Architecture setup for non-overlapping batches in parallel

There are some small caveats with this setup. First, we introduced some additional padding onto

each net's bounding box, to reduce chances of conflicts between nets within a batch. Additionally, any batch that contains 3 or fewer nets are placed in the special batch. This resulted in many small batches, mostly containing 4-10 nets per batch.

The following parameters were used for this variation:

pf	pfInc	hf
1	1.4	0.9

8 Results and Comparisons

8.1 Sequential Optimization Results

8.1.1 Netlist and Net-Level Results

The netlist and net-level optimizations results show that they are required for convergence within a reasonable amount of time. The following chart shows the runtime we were able to achieve on our sample problem over four runs, using both sequential optimizations:

Run	Time (s)
1	330.787
2	333.375
3	331.132
4	332.291
Average	**331.9**

The solved routing used a total of 170,986 nodes, equivalent to about **170.986** meters of wire.

8.1.2 Batch Results

The batch sequential solution was able to introduce a 31s speedup:

Run	Time (s)
1	299.839
2	298.351
3	303.910
4	301.272
Average	**300.843**

This may have been improved more if time allowed for finding the optimal slice size used for batching.

This method used a total of **170.618**m of wire.

8.2 Parallel Results

8.2.1 Results of Parallel-Netlists

The results of the first parallel variation, **parallel netlists**, are as follows:

Cores	Run 1 (s)	Run 2 (s)	Average (s)
1	137.48	139.171	138.33
2	132.116	133.690	132.9
3	142.537	141.581	142.06
4	148.437	-	148.437
5	157.566	-	157.566
6	160.941	-	160.941
7	164.770	-	164.770
8	168.819	-	168.819

Threadscope shows that there are a good amount of sparks and most sparks are converted, but ultimately the gained speedup was limited. The issue may arise from the method of partitioning groups. Since we must find isolated subproblems/netlists within the entire problem and only a few such groups were found, there could be load balancing issues. Each netlist has a different level of difficulty to solve. However, the biggest issue most likely comes from the inherent nature of the variation: it relies upon a sequential calculation, solving netlists that do not fit nicely inside a single slice. Unfortunately, many nets fell into this sequential group. These are most likely the most difficult ones to solve, since they are likely to occupy a larger area on the board, and thus computation time would become dominated by this sequential work.

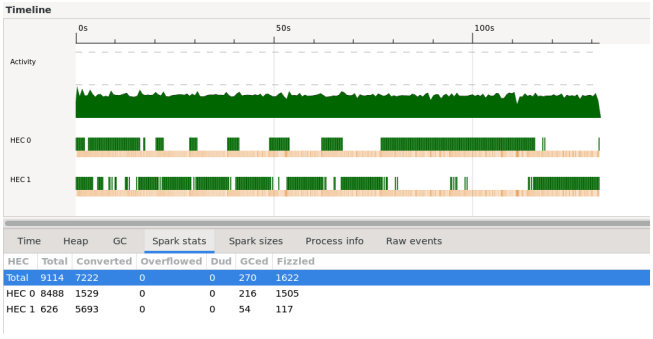

Figure 19: Threadscope chart for parallel-netlists variation with two cores

As it can be observed, sparks are long lived, but time of actual parallelism seems limited.

The next threadscope chart shows three cores and showcases the same issue as above. There are stretches of parallelism, but also prevalent stretches of sequential computation.

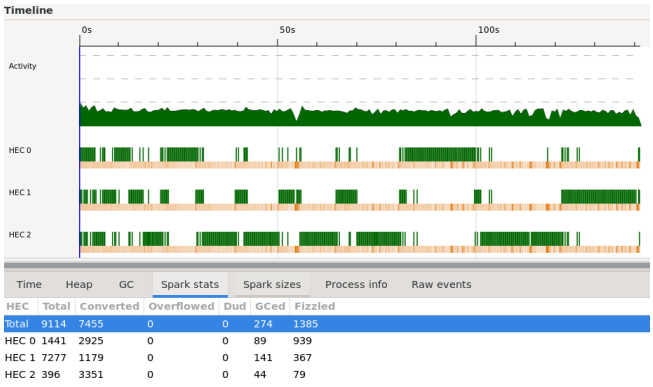

Figure 20: Threadscope chart for parallel-netlists variation with three cores

These charts show that this variation most likely suffers due to load balancing issues and domination of sequential execution.

The resulting chip routing used a wire length of about **171.044** meters.

8.2.2 Results of Sequential/Parallel Alternations

The results of the second parallel variation, **parallel-sequential alternation**, are as follows:

Cores	Run 1 (s)	Run 2 (s)	Average (s)
1	282.983	286.984	284.98
2	256.705	256.632	256.67
3	291.178	-	291.178
4	302.971	-	302.971
5	318.386	-	318.386
6	323.329	-	323.329
7	316.111	-	316.111
8	318.419	-	318.419

The threadscope chart below shows a similar trend to the first parallel attempt. There is some parallelism, specifically in the beginning, but the sparks become shorter lived as the algorithm progresses. This is because the algorithm relies on parallelism to explore quickly, but ultimately requires a sequential end to reach a stable solution.

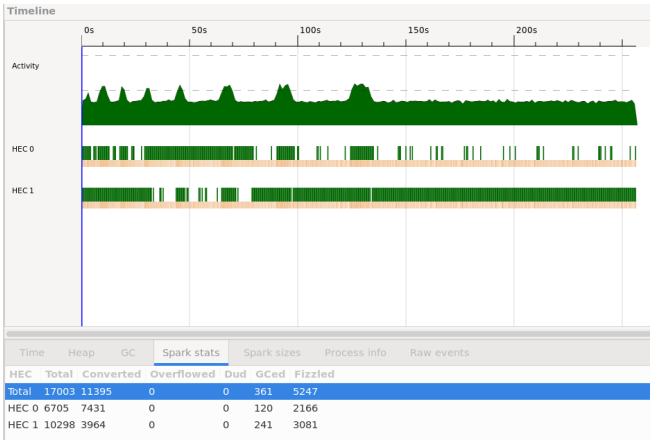

Figure 21: Threadscope chart for the seq/par alt variation with two cores

The parallelism at the beginning comes from a different reason than at the end. In the beginning, where most speedup is gained, we are running batches in parallel, but at the end, we switch to just sequential execution. The parallelism observed near the end is from running A* in parallel on larger nets.

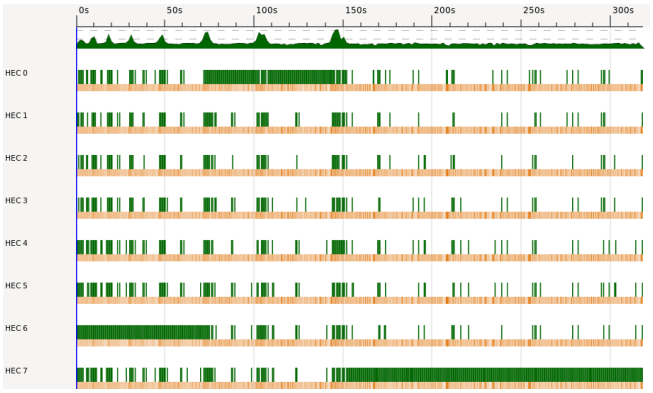

Figure 22: Threadscope chart for the seq/par alt variation with eight cores

Above shows execution over eight cores. There are glimpses of parallelism at the beginning, but once again degrades due to the amount of sequential work required near the end of the algorithm. Overall, the chart appears to be dominated by sequential calculations.

The resulting chip routing from the batch sequential setup used a total of **169.234** meters of wire.

8.2.3 Results of Non-overlapping Batches in Parallel

The results of the third parallel variation, **non-overlapping batches**, are as follows:

Cores	Run 1 (s)	Run 2 (s)	Average (s)
1	174.285	172.236	173.24
2	161.595	161.128	161.362
3	198.621	-	198.621
4	211.561	-	211.561
5	223.143	-	223.143
6	224.918	-	224.918
7	228.285	-	228.285
8	228.076	-	228.076

The following threadscope chart shows a consistent amount of parallelism throughout execution. The sparks are somewhat short lived, but there is more consistency in parallelism than the other variations.

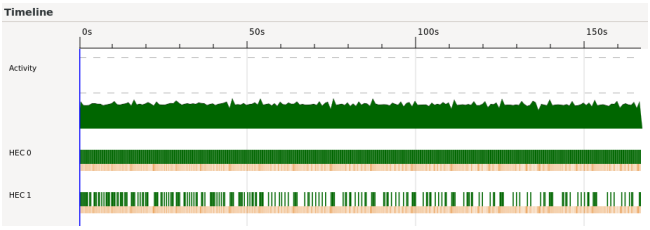

Figure 23: Threadscope chart for the non-overlapping batches in parallel method with two cores

This trend is also apparent with eight cores:

Figure 24: Threadscope chart for the non-overlapping batches in parallel method with eight cores

However, there still seems to be more overhead for parallelism than is gained after 2 cores. After 2 cores, parallelism costs dominate and speed suffers.

The resulting chip routing from the batch sequential setup used a total of **168.032** meters of wire.

8.3 Comparison of Variations

8.3.1 Comparing Speed

For the **sequential optimizations**, we have the following comparisons:

Both Optims	Seq Batch	NL Only	Net Only
331.9s	300.843s	20m+*	20m+*

The sequential batch optimization resulted in a 30s, or 9.36%, speed up. The reason for this speedup is unclear. Batching removes a strict dependency chain and allows nets to explore more without being bogged down by present usage from prior nets. Without batching, netlist ordering is very important, but with batching there's less of a dependency and thus a bad choice in ordering is felt less. Also, there is a chance that the provided pf, pfInc, and hf configuration values were better tuned to this setup.

*Evidently, these results show that both the net-level and netlist optimizations are necessary for convergence on any useful dataset. Without either optimization, computation did not finish within 20 minutes. As a result, outputs and results for all variations (sequential and parallel) are achieved using both of these optimizations turned on.

For the **parallel optimizations**, the following speed-up vs core count chart compares them directly:

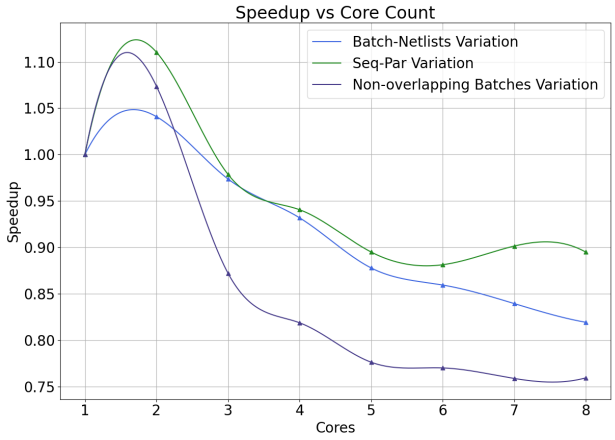

Figure 25: Speedup vs cores for parallel variations

The following graph compares all the parallel outcomes to the ideal speedup. All fall short of the ideal case, which is expected by Amdahl's law. Amdahl's law states that we cannot expect to achieve infinite speedup, but we were hoping for closer curves to the ideal. However, given this was a difficult problem to parallelize, we are happy with the paralleism we were able to accomplish.

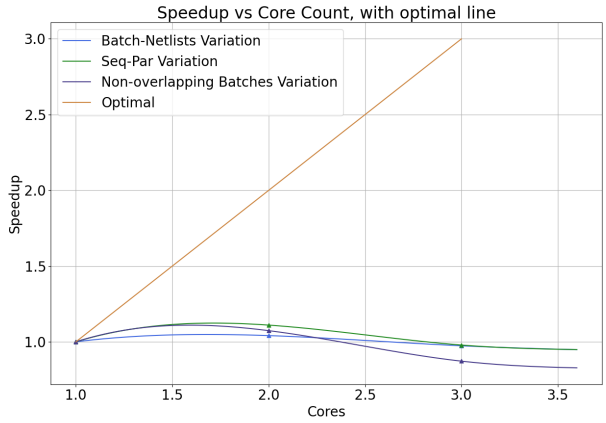

Figure 26: Speedup vs cores for parallel variations, with optimal line

The second variation, Sequential-Parallel, gains about 13% speedup from one to two cores, but then degrades with more threads. This outperforms the first (parallel netlists) and third parallel (non-overlapping batches) attempts, which only observed about a 4% and 6% speed ups with 2 cores, respectively.

While its parallel speed up, when compared to its baseline runtime on a single core, is the weakest, it is impressive that the parallel netlists method had a speed-up of over 50%, when compared against the sequential runs (e.g. 300s vs 132s). As discussed before, since batching allows nets to explore differently, a valid routing was most likely stumbled upon earlier on in the process. Another reason it may decrease runtime is because it keeps the present usage map smaller which in turn makes the A* heuristic more accurate, pruning more of the search space. A similar analysis and results were observed for the third parallel variation explored.

Method	Best Time (s)	Cores	Speedup
Variation 1	**132.9**	2	4.08%
Variation 2	256.67	2	**13%**
Variation 3	161.362	2	6.3%

8.3.2 Comparing Quality

We compare the **quality** of each method by measuring the total wire length used in each resulting chip routing:

Variation	Length (m)
Sequential	170.976
Sequential Batch	170.618
Parallel Netlists (var 1)	171.044
Sequential-Parallel Alt (var 2)	169.234
Non-Overlapping Batch Par (var 3)	**168.032**

The best performer is the non-overlapping batch parallel variation, using only 168.032 meters of wire while the others required about 1-2 meters more. The worst performer was the parallel netlists variation, requiring 171.044m, yielding a similar result to the sequential batch version.

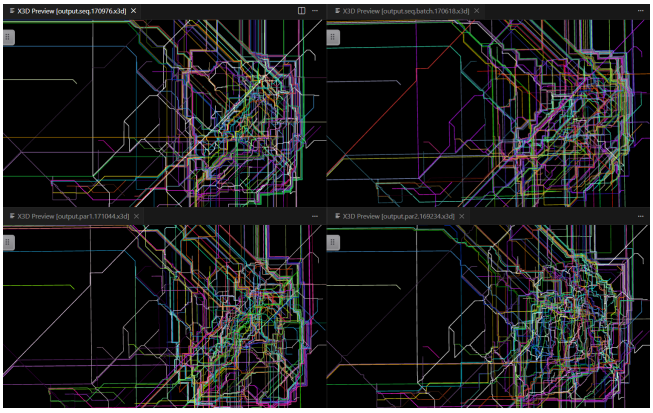

Figure 27: Compares quality between variations. Routings are displayed for the sequential, sequential batch, the first, and the second parallel variations, respectively.

Slight differences in solved routings can be observed in the image above. The seq-par alt (variation 2) method seems to sprawl-out more and be less box-like than the others. The third variation yielded the best quality and is shown below:

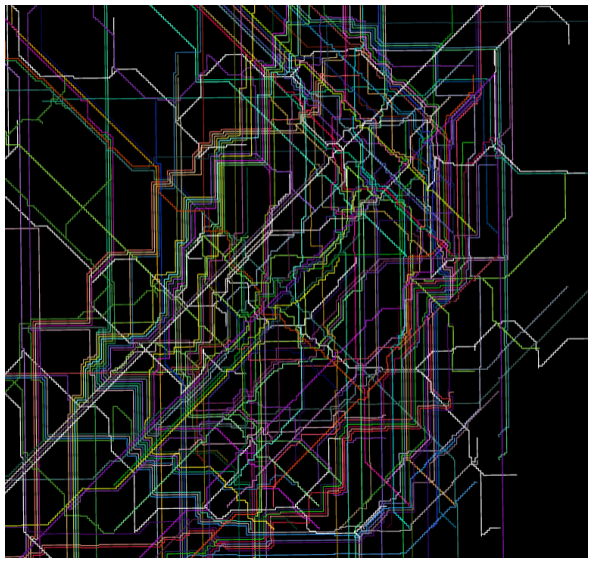

Figure 28: Non-overlapping batch parallel results

8.4 Summary of results

The results are summarized in the following table detailing the best runtime, best speedup (when com-

pared to running on 1 core), and quality/length of routing:

Variation	Time	Speed-up	Quality
Seq	331.9s	-	170.976m
Seq-Batch	300.843s	-	170.618m
Par-Netlists	**132.9s**	4.08%	171.044m
Seq-Par Alt	256.67s	**13%**	169.234m
Non-O Batch	161.362s	6.3%	**168.032m**

Using the following configuration parameters for each:

Variation	pf	pfInc	hf
Seq	1	1.5	1
Seq-Batch	1	1.5	1
Par-Netlists	1	1.55	1
Seq-Par Alt	1	1.28	1
Non-O Batch-Par	1	1.4	0.9

As we can see, each method has positives and negatives and each required different configuration values to work, making comparison difficult. The winner is not quite clear. The parallel netlists variation was able to realize the best overall runtime, but worst quality and smallest speedup from a single to dual core execution. The sequential-parallel alternation method was able to attain the best speedup rate from 1 to 2 cores. Lastly, the non-overlapping batches method produced the best quality. The sequential solutions are the slowest and produce mediocre results.

9 Future Work

There were a few ideas that we were unable to explore due to time constraints.

9.1 Ripping-Up Only Conflicted Nets

After each NC iteration, all paths are ripped up regardless of conflicts. As a potential enhancement to be explored in the future, one could detect which nets have conflicts. Then, the next NC iteration would only have to be executed with conflicted nets. This could substantially speed up execution especially during later iterations.

9.2 Using Different Heuristics

We used the Manhattan distance heuristic, helping ensure reasonable quality and convergence. However, a heuristic that somehow accounts for congestion may also improve speed-up. For example, taking into account density of pads could be beneficial.

9.3 Directional Enforcement

In practice, layers are typically used methodically by designating each layer a particular direction. For example, the top layer could encourage horizontal movements while the bottom encourages vertical movements. Enforcing this via costs could potentially change quality and performance. Another cost that could be enforced to make the problem more realistic would be charging more for via, cross-layer, movements.

9.4 Optimize A*

The A* algorithm was not optimized too much during this project, but there are a few methods to investigate such as:

1. Bidirectional A* search
2. Parallelize A* such that all possible pair combinations are attempted in parallel, using the quickest results as the answer; exiting once all pairs are connected.

10 Code

A subset of code is provided below showcasing the salient parts. All code can be found on GitHub: `https://github.com/mastranj/Chip-Routing/tree/main`, which is the preferred method to ensure most up-to-date code is obtained with the correct spacing and for ease of navigation.

10.1 KiCAD Parser Project

Lib.hs:

```
module Lib
    ( entryPoint
    ) where

import System.Environment (getArgs, getProgName)
import System.Exit (die)
import SExpr
import ProcessSExpr
import Utils
import SvgGenerator
import FileCleaner

entryPoint :: IO ()
entryPoint = do
  args <- getArgs
  run  args

run :: [String] -> IO ()
run [f, out1, mm]              = do
  let path1                    = "output/" ++ out1 ++ ".fulldetails.txt"
  let path2                    = "output/" ++ out1 ++ ".cleaned.txt"
  let path3                    = "output/" ++ out1 ++ ".svg"

  let mmD                      = round (1 / (toDble mm))

  writeFile path1 "" -- clean the file out
  writeFile path2 "" -- clean the file out

  contents                    <- readFile f
  let cleanContents           = [ c | c <- contents, c /= '\r' && c /= '\n' ]

  generateNetPoints mmD path1 $ parseSExpr cleanContents
  (coords, groupedNetList)    <- cleanFile path1 path2
  let roundingError           = 200
  let minX                    = getExtremeX min 100000 coords
  let minY                    = getExtremeY min 100000 coords
  let maxX                    = getExtremeX max 0 coords
  let maxY                    = getExtremeY max 0 coords
  let maxX'                   = maxX-minX+roundingError
  let maxY'                   = maxY-minY+roundingError

  generateSvg path3 (roundingError,minX,maxX',minY,maxY') groupedNetList
  putStrLn "Done"
run _   = do
  progName <- getProgName
  die $ "\n == Usage: " ++ progName
      ++ " <file_to_parse> <out_filename> <mm_in_decimal>\n"
```

Types.hs:

```
module Types
    ( SExpr(..), MDouble, TripleD, PairD, PairS, NetInfo,
      PairI, NetRep
    ) where

data SExpr              = Atom String | SExpr [SExpr] deriving (Show)
type MDouble            = Maybe Double
type TripleD            = (Double, Double, Double)
type PairD              = (Double, Double)
type PairS              = (String, String)
type PairI              = (Int, Int)
type NetInfo            = (String, String, Double, Double, Double, String)
type NetRep             = (String, Int, Int, String, String)
```

SExpr.hs:

```
module SExpr
    ( parseSExpr, SExpr(..)
    ) where

import Types

parseSExpr :: String -> SExpr
parseSExpr              = fst . _parseSExpr

_parseSExpr :: String -> (SExpr, String)
_parseSExpr ('\"':ls)  = parseQuote ls
_parseSExpr ('(':ls)   = (SExpr l, ls2)
  where
    (l,ls2) = getLs ls
    getLs :: String -> ([SExpr],String)
    getLs (')':x)      = ([],x)
    getLs (' ':x)      = getLs x
    getLs x            = (s : ll, lss)
      where
        (ll, lss) = getLs st
        (s,st) = _parseSExpr x
_parseSExpr (')':_)    = error $ "Bad SExpr"
_parseSExpr (' ':ls)   = _parseSExpr ls
_parseSExpr []         = error $ "Bad example"
_parseSExpr s          = parseAtom s

parseQuote :: String -> (SExpr, String)
parseQuote s           = (Atom s1,tail s2)
  where
    (s1,s2) = span (isQuoteChar) s
    isQuoteChar :: Char -> Bool
    isQuoteChar '\"'  = False
    isQuoteChar _     = True

parseAtom :: String -> (SExpr, String)
parseAtom s           = (Atom s1,s2)
  where
    (s1,s2)           = span (isAtomChar) s
```

```
    isAtomChar :: Char -> Bool
    isAtomChar '('    = False
    isAtomChar ')'    = False
    isAtomChar ' '    = False
    isAtomChar _      = True
```

ProcessSExpr.hs:

```
module ProcessSExpr
    ( generateNetPoints
    ) where

import Types
import Utils

generateNetPoints :: Int -> String -> SExpr -> IO ()
generateNetPoints scale f (SExpr s) = do
  _generateNetPoints scale f s
  putStrLn $ "Generated " ++ f

generateNetPoints _ _ _ = error "Bad SExpr to start generating from."

_generateNetPoints :: Int -> String -> [SExpr] -> IO ()
_generateNetPoints scale f (Atom "module":ls)    = _processModule scale f ls
_generateNetPoints scale f (Atom "footprint":ls) = _processModule scale f ls
_generateNetPoints scale f (SExpr s:ls) = do
  _generateNetPoints scale f s
  _generateNetPoints scale f ls
_generateNetPoints scale f (_:ls)                = _generateNetPoints scale f ls
_generateNetPoints _ _ []                        = return ()

_processModule :: Int -> String -> [SExpr] -> IO ()
_processModule scale f s = do
  let netInfos = _findPadNets s
  let moduleAt = _findAt s
  printModule scale f netInfos moduleAt

printModule :: Int -> String -> [NetInfo] -> TripleD -> IO ()
printModule _ _ [] _ = return ()
printModule scale f ((nid, name, x, y, _, layers):ls) moduleAt = do
  let (x',y') = scaleCoord scale $ toGlbalPos moduleAt (x,y)
  appendFile f $ unwords [nid, show x', show y', name, layers]
  appendFile f "\n"
  printModule scale f ls moduleAt

_findAt :: [SExpr] -> TripleD
_findAt (SExpr (Atom "at":Atom x: [Atom y]):_)         =
  (read x, read y, 0)
_findAt (SExpr (Atom "at":Atom x: Atom y: [Atom t]):_) =
  (read x, read y, read t)
_findAt (_:ls)                                         = _findAt ls
_findAt []                                             =
  error $ "At cannot be found"

_findPadNets :: [SExpr] -> [NetInfo]
_findPadNets (SExpr (Atom "pad":pd):ls)
  | _badNet == n    = _findPadNets ls
```

```
  | otherwise       = (nid, name, x, y, t, layers) : _findPadNets ls
  where
    (x,y,t)         = _findAt pd
    n@(nid,name)    = _findNets pd
    layers          = _findLayer pd
_findPadNets (_:ls) = _findPadNets ls
_findPadNets []     = []

_findLayer :: [SExpr] -> String
_findLayer (SExpr (Atom "layers":Atom l1:Atom l2:Atom l3:_):_) =
  unwords [l1,l2,l3]
_findLayer (_:ls) = _findLayer ls
_findLayer []   = ""

_findNets :: [SExpr] -> PairS
_findNets (SExpr (Atom "net":Atom nid:[Atom name]):_) = (nid,name)
_findNets (_:ls)                                      = _findNets ls
_findNets []                                          = _badNet

_badNet :: PairS
_badNet = ("","")
```

SvgGenerator.hs:

```
module SvgGenerator
    ( generateSvg
    ) where

import Types
import Data.List (isPrefixOf)

generateSvg :: String -> (Int,Int,Int,Int,Int) -> [[NetRep]] -> IO ()
generateSvg outp (pad,minX,maxX',minY,maxY') net = do
  writeFile outp $ "<svg xmlns=\"http://www.w3.org/2000/svg\" viewBox=\"0 0 "
    ++ show maxX' ++ " " ++ show maxY' ++ "\" width=\"400\" height=\"400\">\n"
  _parseNetRepLL (pad,minX,minY) outp net
  appendFile outp "</svg>"
  putStrLn $ "Generated " ++ outp

_parseNetRepLL :: (Int,Int,Int) -> String -> [[NetRep]] -> IO ()
_parseNetRepLL _ _ [] = return ()
_parseNetRepLL   ip f (netlist:others) = do
  _parseNetRepL  ip f (head netlist) (tail netlist)
  _parseNetRepLL ip f others

_parseNetRepL :: (Int,Int,Int) -> String -> NetRep -> [NetRep] -> IO ()
_parseNetRepL _ _ _ [] = return ()
_parseNetRepL ip f (_,x,y,_,layer) (to@(_,toX,toY,_,_):ls) = do
  writeLinesToSvg ip f x y toX toY layer
  _parseNetRepL ip f to ls

writeLinesToSvg :: (Int,Int,Int) -> String -> Int -> Int -> Int -> Int
                -> String -> IO ()
writeLinesToSvg (pad,minX,minY) outp x y toX toY layer = do
  let padding = pad `div` 2
  appendFile outp $ "\t<polyline points=\""
    ++ show (x-minX+padding) ++ " " ++ show (y-minY+padding) ++ " "
```

```
    ++ show (toX-minX+padding) ++ " " ++ show (toY-minY+padding)
    ++ "\" fill=\"none\" stroke=\""
    ++ getLayerColor layer ++ "\" stroke-width=\"3\" />\n"

getLayerColor :: String -> String
getLayerColor l
  | l == "TOP"          = "green"
  | "*" `isPrefixOf` l  = "magenta"
  | otherwise           = "blue"
```

10.2 Chip Router Project

Types.hs:

```
module CRTypes.Types
    ( Connection, Net, NetList, Coord, Segment, SegmentList, Routing,
      Bounds, PrevM, Direction(..), MaybeDirection(..), Problems,
      CostM, Cost, CoordRep, Neighbors, CostMKey,
      Args
    ) where

import qualified Data.Map as Map

type ArgInpFile     = String
type ArgPf          = Double
type ArgPfInc       = Double
type ArgMaxPf       = Double
type ArgHf          = Double
type Args           = (ArgInpFile, ArgPf, ArgPfInc, ArgMaxPf, ArgHf)

type Cost           = Double
type CostMKey       = Int
type CostM          = Map.Map CostMKey Cost

type PrevM          = Map.Map Coord Coord

data MaybeDirection = DViaUp | DViaDown
data Direction      = DLeft | DRight | DUp | DDown

type X              = Int
type Y              = Int
type Layer          = Int
type CoordRep       = Int

type Bounds         = ((X,Y), (X,Y))

type Connection     = (Coord, Coord)
type Net            = [Connection]
type NetList        = [Net]
type Problems       = [NetList]

type Coord          = (X, Y, Layer)
type Neighbors      = [Coord]

type Segment        = [Coord]
type SegmentList    = [Segment]
```

```
type Routing        = [SegmentList]
```

Lib.hs:

```
module Lib
    ( entryPoint
    ) where

import CRParallel.ParNegotiation
import CRParallel.ParNegotiationBatch
import CRParallel.ParNonoverlapGroups
import CRTypes.Types
import CRUtils.WriterFuncs
import CRUtils.Args
import Negotiation.BatchNegotiation
import Negotiation.Negotiation
import NetTools.NetOrderer
import NetTools.NetScorer
import NetTools.Partitioners.SlicePartitioner
import NetTools.InputProcessor
import System.Environment (getArgs, getProgName)
import System.Exit (die)

entryPoint :: IO ()
entryPoint = do
  args <- getArgs
  run args

run :: [String] -> IO ()
run [connFile, pf,pfInc,pfMax, hf, maxT,maxB, optN,optNL, parType, isBatch] = do
  let args = (connFile, read pf, read pfInc, read pfMax, read hf) :: Args
  showArgs args maxT maxB optN optNL parType isBatch
  contents <- readFile connFile

  let ls         = lines contents
  let nl         = toSegmentsIgnL "" ls (read maxT) (read maxB)
  let nl'        = optimNetsOrder (read optN) nl
  let nl''       = optimNLOrder (read optNL) nl'

  let parV       = (read parType) :: Int

  if parV == 3 then do
    putStrLn     $ "Running in parallel " ++ parType ++ ": Nonoverlapping batch"
    let routing  = parProblemsNegotiateRoute args nl''
    let score    = scoreRouting routing
    putStrLn     $ "Score: " ++ show score
    routingToX3D routing $ "output.par3." ++ show score++ ".x3d"

  else if parV == 2 then do
    putStrLn     $ "Running in parallel " ++ parType ++ ": Seq-Par Alternations"
    let routing  = parNegotiateRoute args nl''
    let score    = scoreRouting routing
    putStrLn     $ "Score: " ++ show score
    routingToX3D routing $ "output.par2." ++ show score++ ".x3d"

  else if parV == 1 then do
```

```
    putStrLn     $ "Running in parallel " ++ parType ++ ": Slice batching"
    let routing  = parNegotiateRouteB args nl''
    let score    = scoreRouting routing
    putStrLn     $ "Score: " ++ show score
    routingToX3D routing $ "output.par1." ++ show score++ ".x3d"

  else if read isBatch then do
    putStrLn     "Running batch (sequentially)"
    let nlPart   = partsByBounds 0 (0, 1200) (0, 1800) nl'' ((0,0),(9999,9999))
    let routing  = batchNegotiateRoute args nlPart
    let score    = scoreRouting routing
    putStrLn     $ "Score: " ++ show score
    routingToX3D routing $ "output.seq.batch." ++ show score++ ".x3d"

  else do
    putStrLn     "Running sequentially"
    let routing  = negotiateRoute args nl''
    let score    = scoreRouting routing
    putStrLn     $ "Score: " ++ show score
    routingToX3D routing $ "output.seq." ++ show score++ ".x3d"

  putStrLn "\n\nDone"

run _ = do
  progName <- getProgName
  die            $ "==== Usage: " ++ progName
                 ++ " <inpFile> <pf> <pfInc> <pfMax> <hf> <maxTopPads> "
                 ++ "<maxBottomPads> <optimizeNets> <optimizeNetlists> "
                 ++ "<parType={0,1,2}> <isSeqBatch>"
```

NetOrderer.hs:

```
-- Consulted https://dl.acm.org/doi/pdf/10.1145/201310.201328 for ideas
-- about net ordering
-- Consulted https://www.eecg.toronto.edu/~vaughn/papers/iccad96.pdf for
-- bounding box idea in net ordering
--  - Along with https://en.wikipedia.org/wiki/Minimum_bounding_box

module NetTools.NetOrderer
    ( optimNetsOrder, optimNLOrder, getNetBound
    ) where

import CRTypes.Types
import CRUtils.Heuristics
import qualified Data.List as List
import qualified Data.Set as Set

optimNLOrder :: Bool -> NetList -> NetList
optimNLOrder b nl
  | b == False = nl
  | otherwise  = reverse $ List.sortBy (compareNetBound) nl

compareNetBound :: Net -> Net -> Ordering
compareNetBound n1 n2 = compare (getNetArea n1) (getNetArea n2)

getNetBound :: Net -> Bounds
getNetBound n = ((minX,minY),(maxX,maxY))
```

```
  where
    tmpMax = 1000000
    tmpMin = -10
    (maxX, maxY) = getNetExtremes max tmpMin n
    (minX, minY) = getNetExtremes min tmpMax n

getNetArea :: Net -> Int
getNetArea n = (maxX-minX+1)*(maxY-minY+1)
  where
    tmpMax = 1000000
    tmpMin = -10
    (maxX, maxY) = getNetExtremes max tmpMin n
    (minX, minY) = getNetExtremes min tmpMax n

getNetExtremes :: (Int -> Int -> Int) -> Int -> Net -> (Int, Int)
getNetExtremes _ exDef []                        = (exDef, exDef)
getNetExtremes f exDef (((x,y,_),(x2,y2,_)):ns) = (exX,   exY  )
  where
    (exX2, exY2) = getNetExtremes f exDef ns
    exX = f exX2 $ f x $ f exDef x2
    exY = f exY2 $ f y $ f exDef y2

optimNetsOrder :: Bool -> NetList -> NetList
optimNetsOrder b = map (orderNet b)

orderNet :: Bool -> Net -> Net
orderNet b nl
  | not b          = nl
  | length nl >= 2 = getNetOrdering (_getPoints nl)
  | otherwise      = nl

getNetOrdering :: [Coord] -> Net
getNetOrdering cs = (c1,c2) : _getNetOrdering (c1 : [c2]) womiddle2
  where
    c1 = _getMiddleCoord cs
    womiddle = filter (\x -> x /= c1) cs
    c2 = _getMiddleCoord womiddle
    womiddle2 = filter (\x -> x /= c2) womiddle

_getNetOrdering :: [Coord] -> [Coord] -> Net
_getNetOrdering _ []             = []
_getNetOrdering conned (c:cs) = (_attachPoint conned (-10000,-10000,-100) c) :
                                (_getNetOrdering (c : conned) cs)

_attachPoint :: [Coord] -> Coord -> Coord -> Connection
_attachPoint []      bestC c = (c, bestC)
_attachPoint (c2:cs) bestC c
  | dist > distbest          = _attachPoint cs bestC c
  | otherwise                = _attachPoint cs c2 c
  where
    dist                     = manhattan3DBounded (c, c2)
    distbest                 = manhattan3DBounded (c, bestC)

_getPoints :: Net -> [Coord]
_getPoints = Set.toList . Set.fromList . __getPoints
  where
```

```
    __getPoints :: Net -> [Coord]
    __getPoints []           = []
    __getPoints ((c1,c2):nl) = c1 : c2 : _getPoints nl

_orderCoords :: [Coord] -> [Coord]
_orderCoords = List.sortBy (\c c2 -> compareC c c2)

_getMiddleCoord :: [Coord] -> Coord
_getMiddleCoord cs = head $ drop midpt $ _orderCoords cs
  where midpt      = (length cs) `div` 2

compareC :: Coord -> Coord -> Ordering
compareC (x,y,l) (x2,y2,l2) = compare (x+y+l) (x2+y2+l2)
```

Negotiation.hs:

```
-- followed https://dl.acm.org/doi/pdf/10.1145/201310.201328
module Negotiation.Negotiation
    ( negotiateRoute, _negotiateRoute
    ) where

import CRTypes.Types
import CRUtils.HelperFuncs
import Negotiation.AStarNetListAlg
import qualified Data.Map as Map

negotiateRoute :: Args -> NetList -> Routing
negotiateRoute args n = _negotiateRoute args Map.empty n

_negotiateRoute :: Args -> CostM -> NetList -> Routing
_negotiateRoute args@(inp, pf, pfInc, pfMax, hf) hcm nl
  | lofw == 0    = iter
  | otherwise    = _negotiateRoute args' hcm' nl
  where
    iter         = aStarNetList Map.empty hcm args nl
    overflow     = getOverflow iter
    (lofw, hcm') = (length overflow, getNextCost overflow hcm hf)
    pf'          = min pfMax $ pf * pfInc
    args'        = (inp, pf', pfInc, pfMax, hf)
```

AStarNetListAlg.hs:

```
module Negotiation.AStarNetListAlg
    ( aStarNetList, aStarNetListH
    ) where

import CRTypes.Types
import CRUtils.HelperFuncs (accumulateOverflowFromIter)
import CRUtils.Heuristics
import Negotiation.AStarNetAlg

aStarNetListH :: H -> CostM -> CostM -> Args -> NetList -> Routing
aStarNetListH _ _ _ _ []           = []
aStarNetListH h om hcm args (n:nl) = solvedNet : aStarNetListH h om' hcm args nl
  where
    solvedNet = aStarNetH h om hcm args n
```

```
    om'       = accumulateOverflowFromIter om [solvedNet]

aStarNetList :: CostM -> CostM -> Args -> NetList -> Routing
aStarNetList  = aStarNetListH manhattan3DBounded
```

AStarNetAlg.hs:

```
module Negotiation.AStarNetAlg
    ( aStarNet, aStarNetH
    ) where

import CRTypes.Types
import CRUtils.Heuristics
import Negotiation.AStarConnAlg

aStarNetH :: H -> CostM -> CostM -> Args -> Net -> SegmentList
aStarNetH _ _ _ _ []           = []
aStarNetH h om hcm args (n:nl) = solvedConn : aStarNetH h om hcm args nl
  where solvedConn             = aStarConnH h om hcm args n

aStarNet :: CostM -> CostM -> Args -> Net -> SegmentList
aStarNet = aStarNetH manhattan3DBounded
```

AStarConnAlg.hs:

```
-- Citations:
-- 1. Used https://en.wikipedia.org/wiki/A*_search_algorithm for base of A*

module Negotiation.AStarConnAlg
    ( aStarConn, aStarConnH
    ) where

import CRTypes.Types
import CRUtils.Heuristics
import CRUtils.HelperFuncs
-- https://hackage-content.haskell.org/package/psqueues-0.2.8.2/docs/Data-IntPSQ.html
import Data.IntPSQ as Q
import qualified Data.Map as Map

type Q = Q.IntPSQ Cost Cost

aStarConnH :: H -> CostM -> CostM -> Args -> Connection -> Segment
aStarConnH h overuseCost historyCost args n@(s, t) = reconPath n endPrevM
 where
  endPrevM = _aStarConn h t q Map.empty overuseCost historyCost scoreM args
  q        = Q.fromList [(coordToInt s, 0, 0)]
  scoreM   = fromListCostM [(s,0)]

aStarConn :: CostM -> CostM -> Args -> Connection -> Segment
aStarConn  = aStarConnH manhattan3DBounded

_aStarConn :: H -> Coord -> Q -> PrevM -> CostM -> CostM -> CostM
           -> Args -> PrevM
_aStarConn h t q prevM oc hcm scoreM args = case Q.findMin q of
  Nothing        -> error "_aStarConn reached unreachable case."
  Just (curr,_,_)
```

```
    | curr == coordToInt t -> prevM
    | otherwise            -> _aStarConn h t q' pm' oc hcm scrm' args
      where
        (scrm',pm',q') = asStep args q (intToCoord curr) h t oc scoreM hcm prevM

asStep  :: Args -> Q -> Coord -> H -> Coord -> CostM -> CostM
        -> CostM -> PrevM -> (CostM, PrevM, Q)
asStep args q curr = _asStep args (Q.deleteMin q) curr (getAllNeighbors curr)

_asStep  :: Args -> Q -> Coord -> Neighbors -> H -> Coord -> CostM
         -> CostM -> CostM -> PrevM -> (CostM, PrevM, Q)
_asStep _ q _ [] _ _ _ scoreM _ prevM = (scoreM, prevM, q)
_asStep args q curr (neigh:ns) h targ overuseM scoreM hcm prevM =
  _asStep args q' curr ns h targ overuseM scoreM' hcm prevM'
  where
    (scoreM', prevM', q') =
      _asStepOnce h curr targ neigh scoreM overuseM hcm prevM q args

_asStepOnce :: H -> Coord -> Coord -> Coord -> CostM -> CostM -> CostM
            -> PrevM -> Q -> Args -> (CostM, PrevM, Q)
_asStepOnce h curr targ neigh scoreM oc hcm prevM q (_,pf,_,_,hf)
  | tentativeScore < scoreOfNeigh = ( scoreM', prevM', q' )
  | otherwise                     = ( scoreM,  prevM,  q  )
  where
    dcost = 100000

    scoreOfNeigh   = getCostD dcost scoreM neigh
    boardCostNeigh = 1 + (hf*getCost hcm neigh)
    overuseFac     = pf * getCostD 0.0 oc neigh
    neighborCost   = boardCostNeigh*(overuseFac+1)
    costToCurr     = getCostD dcost scoreM curr

    tentativeScore = costToCurr + neighborCost

    fscore         = tentativeScore + (h (neigh, targ))
    scoreM'        = insertCostM neigh tentativeScore scoreM
    prevM'         = Map.insert neigh curr prevM
    q'             = Q.insert (coordToInt neigh) fscore fscore q

-- Reconstruct the path
reconPath :: Connection -> PrevM -> Segment
reconPath n@(_,t) prevM = reverse $ t : (_reconPath n prevM)

_reconPath :: Connection -> PrevM -> Segment
_reconPath (s,curr) prevM = case Map.lookup curr prevM of
  Just prev -> if prev == s then [s] else prev : _reconPath (s,prev) prevM
  _         -> error "_reconPath reached unreachable case."
```

ParNegotiation.hs:

```
-- https://ceca.pku.edu.cn/media/lw/f8937474bc4352fa545e32f55ae8e7be.pdf
-- mentions partitioning with nonoverlapping bouding boxes and run those
-- in parallel. They cite https://dl.acm.org/doi/10.1145/2435264.2435322
-- which I am unable to access.
module CRParallel.ParNegotiation
    ( parNegotiateRoute
```

```
    ) where

import Control.Parallel.Strategies
import CRParallel.ParAStarNetListAlg
import CRParallel.ParAStarNetAlg
import CRTypes.Types
import CRUtils.HelperFuncs
import NetTools.Partitioners.SlicePartitioner
import qualified Data.Map as Map

parNegotiateRoute :: Args -> NetList -> Routing
parNegotiateRoute a n = _parNegotiateRoute 100000 0 a Map.empty Map.empty n

_parNegotiateRoute :: Int -> Int -> Args -> CostM -> CostM -> NetList -> Routing
_parNegotiateRoute psc i args@(inp, pf, pfInc, pfMax, hf) pOcm hcm n
  | lofw == 0 = iter
  | isNextPar = _parNegotiateRoute lofw (succ i) args' overflow hcm' n
  | otherwise = _parNegotiateRoute lofw (succ i) args' Map.empty hcm' n
    where
      isParRun :: Int -> Int -> Bool
      isParRun sc it = it `mod` 2 /= 0 && sc > 15

      isPar                = isParRun psc i
      isNextPar            = isParRun psc $ i + 1
      ps       | isPar     = toGroupsSizeN _getGroupS n
               | otherwise = [n]
      usedArgs | isPar     = (inp, pf*0.5, pfInc, pfMax, hf*2)
               | otherwise = args
      lps                  = (length ps > 1)
      iter                 = _processProblems ps lps usedArgs pOcm hcm
      overflow             = getOverflow iter
      lofw                 = length overflow
      hcm'                 = getNextCost overflow hcm hf
      pf'                  = min pfMax $ pf * pfInc
      args'                = (inp, pf', pfInc, pfMax, hf)

_getGroupS :: Int
_getGroupS = 25

_processProblems :: Problems -> Bool -> Args -> CostM -> CostM -> Routing
_processProblems [] _ _ _ _ = []
_processProblems (nl:nls) isPar args pOcm hcm
  | not isPar = parAStarNetList pOcm hcm args nl
  | otherwise = next ++ iter
  where
    iter     = map (parAStarNet pOcm hcm args) nl
               `using` parList rdeepseq :: Routing
    pOcm'    = _accumulateCost pOcm $ iter
    next     = _processProblems nls isPar args pOcm' hcm

_accumulateCost :: CostM -> Routing -> CostM
_accumulateCost om []                 = om
_accumulateCost om (solvedNet:others) = _accumulateCost c others
  where c = accumulateOverflowFromIter om [solvedNet]
```

ParNonoverlapGroups.hs:

```
module CRParallel.ParNonoverlapGroups
    ( parProblemsNegotiateRoute
    ) where

import Control.Parallel.Strategies
import CRParallel.ParAStarNetAlg
import CRParallel.ParAStarNetListAlg
import CRTypes.Types
import CRUtils.HelperFuncs
import NetTools.NetOrderer
import NetTools.Partitioners.OverlapPartitioner
import qualified Data.Map as Map


parProblemsNegotiateRoute :: Args -> NetList -> Routing
parProblemsNegotiateRoute a nl = _parProblemsNegotiateRoute a Map.empty sp gp
  where
    ov    = 150
    lim   = 3
    ps    = partsByNonOverlap ov nl
    subP  = take (length ps - 1) ps
    gp    = optimNLOrder True $ concat $
              (drop (length ps - 1) ps) ++ (filter (\x -> length x <= lim) subP)
    sp    = map (\x -> optimNLOrder True x) $ filter (\x -> length x > lim) subP

_parProblemsNegotiateRoute :: Args -> CostM -> Problems -> NetList -> Routing
_parProblemsNegotiateRoute args@(inp, pf, pfInc, pfMax, hf) hcm ps globalPs
  | lofw == 0  = allIters
  | otherwise  = _parProblemsNegotiateRoute args' hcm' ps globalPs
  where
    globalIter = parAStarNetList Map.empty hcm args globalPs
    otherIters = _processProblems ps args (seglistToCostM globalIter) hcm
    allIters   = globalIter ++ otherIters
    overflow   = getOverflow allIters
    lofw       = length overflow
    hcm'       = getNextCost overflow hcm hf
    pf'        = min pfMax $ pf * pfInc
    args'      = (inp, pf', pfInc, pfMax, hf)

_processProblems :: Problems -> Args -> CostM -> CostM -> Routing
_processProblems [] _ _ _ = []
_processProblems (nl:nls) args pOcm hcm = next ++ iter
  where
    iter  = map (parAStarNet pOcm hcm args) nl
                `using` parList rdeepseq :: Routing
    pOcm' = Map.unionWith (+) pOcm (seglistToCostM iter)
    next  = _processProblems nls args pOcm' hcm
```

ParAStarNetListAlg.hs:

```
module CRParallel.ParAStarNetListAlg
    ( parAStarNetListBatch, parAStarNetList
    ) where

import CRParallel.ParAStarNetAlg
import CRTypes.Types
import CRUtils.HelperFuncs
```

```
import CRUtils.Heuristics

parAStarNetListH :: H -> CostM -> CostM -> Args -> NetList -> Routing
parAStarNetListH _ _ _ _ [] = []
parAStarNetListH h om hcm args (n:nl) =
  solvedNet : parAStarNetListH h om' hcm args nl
  where
    solvedNet = parAStarNetH h om hcm args n
    om'       = accumulateOverflowFromIter om [solvedNet]

parAStarNetList :: CostM -> CostM -> Args -> NetList -> Routing
parAStarNetList om hcm args nl         =
  parAStarNetListH manhattan3DBounded om hcm args nl

parAStarNetListBatch :: CostM -> CostM -> Args -> NetList -> (CostM, Routing)
parAStarNetListBatch om hcm args nl    = (seglistToCostM ans, ans)
  where ans = parAStarNetListH manhattan3DBounded om hcm args nl
```

ParAStarNetAlg.hs:

```
module CRParallel.ParAStarNetAlg
    ( parAStarNet, parAStarNetH
    ) where

import Control.Parallel.Strategies
import CRTypes.Types
import CRUtils.Heuristics
import Negotiation.AStarConnAlg

parAStarNetH :: H -> CostM -> CostM -> Args -> Net -> SegmentList
parAStarNetH h om hcm args nl
  | length nl <= 3  = _parAStarNetH h om hcm args nl
  | otherwise       = map (aStarConnH h om hcm args) nl `using` parList rdeepseq

_parAStarNetH :: H -> CostM -> CostM -> Args -> Net -> SegmentList
_parAStarNetH _ _ _ _ []           = []
_parAStarNetH h om hcm args (n:nl) = solvedConn : _parAStarNetH h om hcm args nl
  where solvedConn                 = aStarConnH h om hcm args n

parAStarNet :: CostM -> CostM -> Args -> Net -> SegmentList
parAStarNet = parAStarNetH manhattan3DBounded
```

11 References

A. Kahng K. Keutzer, A. R. Newton. *Routing in Integrated Circuits*. URL: https://people.eecs.berkeley.edu/~keutzer/classes/244fa2004/pdf/4-routing.pdf.

Cendes, Zoltan. *Electromagnetic and Circuit Co-Simulation and the Future of IC and Package Design*. URL: https://ewh.ieee.org/soc/cpmt/tc12/fdip06/Cendes_Part_I.pdf.

Larry McMurchie, Carl Ebeling. "PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs". In: (1995).

Manual or interactive routing, advantages and inconveniences. URL: https://www.proto-electronics.com/blog/manual-or-interactive-routing.

Minghua Shen, Guojie Luo. "Accelerate FPGA Routing with Parallel Recursive Partitioning". In: ().

Point, Tutorials. *Graph Theory - Infinite Graphs*. URL: https://www.tutorialspoint.com/graph_theory/graph_theory_infinite_graphs.htm.

Wikipedia Contributors. *A* search algorithm*. URL: https://en.wikipedia.org/wiki/A*_search_algorithm.

— *Rotation matrix*. URL: https://en.wikipedia.org/wiki/Rotation_matrix.