EC527 Final: Simulated Annealing

Kestutis Subacius & Joshua Lippai

May 12th, 2017

1. **Description of the Algorithm** 
   1. Background: Place and Route
   2. Background: Fluigi
   3. The SA Algorithm
      1. Conceptual Basis: The Link Between Statistical Mechanics and Combinatorial Optimization.
      2. Base Algorithm: Simulated Annealing
      3. Auxiliary Functions: Cost Analysis.
2. **Serial CPU Implementation**
   1. Implementation and Analysis
   2. Results
3. **Multithreaded CPU Implementation**
   1. Naïve Approach: Simple Multithreading by Inner Loop Parallelism.
      1. Implementation and Analysis
      2. Results
   2. Clever Approach: Parallelism by Graph Partitioning
   3. An Aside: A Brief Overview of Other Parallelization Methods in the Context of PAR.
4. **GPU Considerations**
5. **Closing Discussion**
6. **Supplements**
7. **References**
8. **Description of the Algorithm**

The **Simulated Annealing** algorithm [Kirkpatrik et al. 1983] is a heuristic approach to finding the approximate global optimum of a combinatorial optimization problem. Such subsets of problems often have many variables or a vast solution space with many local optimum solutions among a sole global optimum; hence, traditional gradient descent algorithms often cannot be applied to find the solution to these classes of problems. There are a vast number of real-world problems and applications that can be approached as combinatorial optimization problems, with the canonical example being the *quadratic assignment problem*, of which the traveling salesman problem is a subset.

Simulated Annealing is an algorithm derived from statistical mechanics, wherein the combinatorial optimization problem can be modeled as a system in thermal equilibrium with a large number of degrees of freedom. The global optimum of such a system is found analogously to the slow cooling (or annealing,) of a solid, as such a process results in a final lattice configuration that maintains the lowest energy state of the system. This is analogous to solving a combinatorial optimization problem.

This report focuses on the simulated annealing algorithm in the context of microfluidic place and route, although such a problem is analogous to the placement optimization problems in VLSI and FPGA design as well.

* 1. **Background: Place and Route**

The subset of problems that simulated annealing proves to be very effective on are known as *multivariate combinatorial optimization problems*. There problems involve optimizing a function of a large number of often nonlinear variables in a non-convex search space. These problems are difficult to solve optimally, as traditional gradient descent methods are certain to get trapped in local minima. The immediate approach to solving such a subset of problems is to use *heuristic* approaches. A heuristic is an algorithm implementation that incorporates randomness in generating a solution. Instead of converging on a single optimum, heuristic algorithms are able to search a vast solution space while avoiding the trap of non-optimum local minimums. There are a number of well-known heuristic approaches including genetic algorithms, swarm inteligence, and the focus of this report: simulated annealing.

As it happens, one of the most significant applications that match the subset of problems described above is the problem of *place and route*. Place and route is one stage in the design pipeline of creating ICs, PCBs, and FPGA circuits from high level specifications. For example, in the context of FPGAs, the place and route problem involves determining the most optimum configuration of logic elements placed in the FPGA grid. In such a problem, the user will have specified the functional description of their circuit in a high level description such as Verilog or VHDL, and the goal of the place and route algorithms is to realize this description by placing logic blocks in the most optimal configuration; this involves minimizing a cost function, where cost itself is a function of many variables. There are many possible ways to place and route the circuit, with some solutions more favorable than others. Hence this is this type of problem that simulated annealing can effectively address.

To be clear on vocabulary, the *placement* problem is to solve the question “What is the most optimal way to place all components on a graph?” The *routing* problem is to solve the question “What is the most optimal way to connect all components on a graph, after they have been placed?”

* 1. **Background: Fluigi**

While the issue of place and route has been thoroughly explored in the context of ICs, PCBs and FPGAs, the idea has just recently gained traction in the context of medium-to-large scale microfluidic integration *(mlsi)*. This problem is quite analogous to the problem in programming FPGAs: users specify a high level description of a microfluidic device; its fluid inputs, fluid outputs, and fluid manipulation “logic blocks” in between. From this high level description, a pipeline of algorithms and electronic design tools realizes a microfluidic schematic with the most optimal placement and connection of the described device.

In our work, we focus on the software workflow for microfluidics called Fluigi [Huang 2016]. This software workflow includes many components that come together to realize a microfluidic design from a high level specification. ***Our focus is on implementing and optimizing the simulated annealing algorithm for placement of components****.* I should emphasize that this is only a smaller part of the greater pipeline, and our implementation focuses on the placement component, forgoing the routing algorithm (while some routing algorithms do use SA, Fluigi uses Hadlocks algorithm). As a brief aside, it is well known that the placement SA algorithms are much more favorable to optimization through parallelization than routing SA algorithms, and the placement SA algorithms are generally much more intensive than the routing algorithms. For this reason, in the literature you will find most parallelization’s of SA focus on the placement aspect.

* 1. **The SA Algorithm**

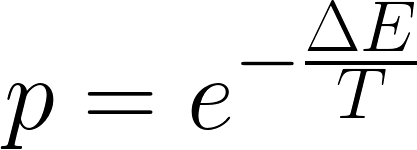
This section of the report focuses on fully describing the baseline SA algorithm. Please refer to this section for details on technical implementation and design considerations, as the remainder of the report will focus on optimization methods and results.

* + 1. **Conceptual Basis: The Link Between Statistical Mechanics and Combinatorial Optimization**

First we describe the conceptual basis for the SA algorithm. The simulated annealing algorithm can be described at the highest level as an implementation of the annealing (or slow cooling) process of a system with a large number of degrees of freedom at equilibrium. The idea is as follows:

First, the system begins at a high thermal energy; programmatically this is monitored by the **temperature** variable. This variable is initialized to a “high” temperature (the definition of “high” depends on the application, and we will discuss the specific values we use in a later section). Just as in an annealing process in thermodynamics, at high temperatures the system can readily access a large number of possible states. In the context of placement, this means that the algorithm is likely and willing to accept a large number of possible component perturbations or configurations, regardless of their validity, when the temperature variable is large. As the thermal energy or temperature of the system decreases (the cooling process,) the number of states the system can readily access also decreases. Algorithmically this means that when the temperature variable is low, the algorithm must be less likely to accept possible component perturbations.

How can an algorithm realize the above description? Directly borrowing from statistical mechanics, the simulated annealing algorithm introduces the **Boltzmann factor** as a way to determine the probability of accepting or rejecting a system state (or component perturbation). Mathematically, the Boltzmann factor is simply:



Where P is the probability of accepting the state, T is the systems temperature, and E is the energy of the state. From simple observation, we see that when T is large, P approaches 1. When T is small, P approaches zero. Hence the Boltzmann factor satisfies the necessary behavior of accepting or rejecting new states as a system cools. E, the energy of the state, is algorithmically realized as the cost of the state. The delta indicates that the change in cost between the old state and the new state (or the old component placement and new component placement,) further determines the probability of accepting the state.

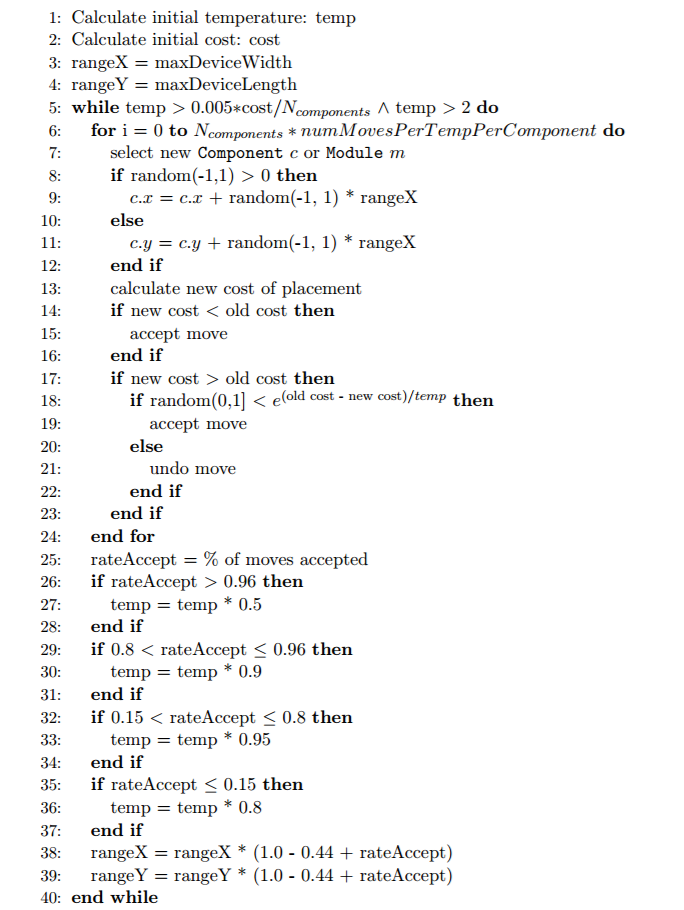
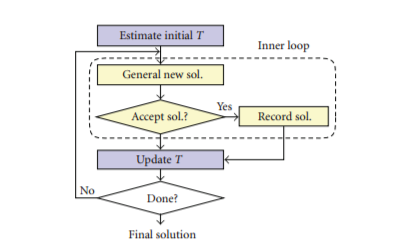
Finally, the algorithm has to realize the actual “states” that are being accessed as the system cools. The SA algorithm works by starting with a given undirected graph. This graph will, in our analysis, serve as the high level description of how we want our components to be placed and routed. The goal of the algorithm will be to determine the most optimal placement of components, given the input graph. To sample new states, the algorithms makes a random perturbation to the graph. The algorithm then calculates the cost difference between this new state and the previous state. Intuitively, if the cost of the new state is less than that of the new state, then a more optimal solution has been found, and this state is accepted. If the cost of the new state is greater than that of the prior state, then a less optimum solution is found; the probability of accepting or rejecting this newly sampled state is now defined by the Boltzmann factor; hence, accepting or rejecting is a function of what temperature the system is at, and the magnitude of the cost difference.

Next we discuss the technical details of the algorithm. There are two main components to focus on in this discussion: The base **SA algorithm**, and the **cost analysis function.**

* + 1. **Base Algorithm: Simulated Annealing**

We followed the implementation of SA for component placement as described by Huang 2016. The base simulated annealing algorithm used by Huang in the Fluigi software workflow is the Betz and Rose 1997 algorithm used for their VPR tool.

The pseudocode for Betz and Rose 1997 as implemented by Huang follows on the next page. This algorithm can also be understood through as block diagram representation, as seen next to the Betz and Rose pseudocode.



There are several components implemented in our algorithms that are unique to the Huang / Betz and Rose implementation of simulated annealing, we shall overview there details here. The first thing to note is that in our implementation, the initial temperature of the system is defined as being proportional to the cost of the input device. That is to say, to initiate the temperature we calculate the initial cost of the device (without having made any optimizations,) and scale this initial cost by a factor (20, in our implementation.) This is because we want the initial temperature to be substantially higher than any cost penalties than can be incurred when perturbations are made. I should note that in our experiments, we do not input a predefined device, rather we use a random device. Hence to calculate the initial temperature, we take the average cost over 20 randomly generated devices. This itself incurs a runtime penalty on the SA algorithm that is worse considering. The key idea here though, is that the initial temperature matters, and this should be derived from the initial input device cost.

The next thing to note is the threshold that we define our system to have converged. Where many SA algorithms will converge when temperature cools to beyond a set threshold, our implementation converges based on two conditionals: The simulated annealing process will continue ifthe temperature is above a threshold value of 2, *and* the cost of the current solution is above a threshold proportional to the number of components. That is to say that our algorithm will converge *either* if the temperature falls before 2, *or* if the cost of the device surpasses an optimum value. This is similar to the implementation of Betz and Rose VPR, but with the added condition of temperature over 2. In code,



The details of how we implement the random perturbations are fairly straight forward. We iterate through all components, and for each component we apply a random move. This movement of the component is not completely random: the maximum radius/distance that the component can be perturbed by is dynamic, and depends on the acceptance rate of perturbations at the prior temperature. Specifically, if there are a lot of rejections to new perturbations at a given temperature, then the maximum radius a component can be perturbed by decreases. This should intuitively make sense, because if most changed are being rejected then the system is nearing convergence and it would not make sense to move components by a large amount. In code this is realized as:



Notice the “1 - .44”. This is *not* an arbitrary value. Betz and Rose determined through experiment that SA performs the best when new perturbations are accepted at a rate of 44%, hence the radius by which a component can be perturbed is mediated to further ensure a 44% accept rate.

* + 1. **Auxiliary Functions: Cost Analysis**

A subtle but incredibly significant portion of this algorithm is the cost analysis. Indeed, with each perturbation made to the graph, the cost needs to be recalculated. (This assumption is an algorithmically costly one to make, and certainly it cannot be the best way- further discussion in the discussion section!) This section outlines the details of the cost analysis.

The cost function is critical because it is the multivariate function that the simulated annealing algorithm is attempting to minimize. Hence, it is critical that cost is calculated correctly. As defined by Huang, the total cost function is a sum of individual penalties:

*Net Cost = Channel Penalty + Overlap Penalty + Area Penalty*

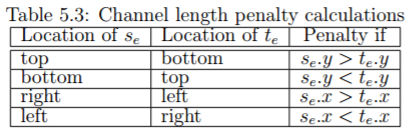
Where the channel penalty reffers to the cost incurred by placing channels (as defined by a cost per unit length). The overlap penalty reffers to the penalty reffered to overlapping components (intuitivly, this is a huge penalty,) and the area penalty is the penalty incured by the area of your grid (where a largher area means fabrication will require more materials.) Each of these individual penalties is addressed independantly.

The details of the cost function follow.

*Calculating Channel Penalty*: Channel penalty is determined by finding the manhattan distance betweeen all connected components in the device. As defined by Huang:



A manhattan distance (dx + dy) is used when calculatign the channel length, because microfluidic chip channels are not routed diaganolly. The general idea is that this function iterates over all components in the graph. For each component, the function looks at the up-to-four neighboring components. Knowing the positions of each component allows us to calculate the locations of each port on the component, and then the manhattan geometry is calcuated between the two conencted ports of these compoents in question. This function keeps a record of distances that were already calculated so as to not double count. Minimizing the channel length in component placment is only one part of the channel penalty. The other aspect to channel penalty is overlaps. Huang defined four possible overlap cases that we check for in our algorithm:



Quite simply, this is to say that you should never route from a top channel of a component to the bottom channel of another component when the first component is above the second, and the analogy for the other three cases. Now, the channel length pentalty is 2 per unit length. The overlap penalty is 10000. This makes sense: overlaps should be avoided at all cost.

*Calculating Overlap Penalty:* Overlap penalty in this sense reffers to component overlaps, rather than channel overlaps. The overlap penalty is defined by Huang quite sensibly as:



In otherwords, there is a penalty proportional to the area of component overlap. Overlap cost here is also 10000. The grid data structure, and the block size parameter (discussed in the next section,) defines the granularity of the overlap area calculation. To calcualte the overlap area, the program must iterate over two dimensions of the grid. For each grid location where there is a refference to multiple components, the algorithm tallies this as a overlap.

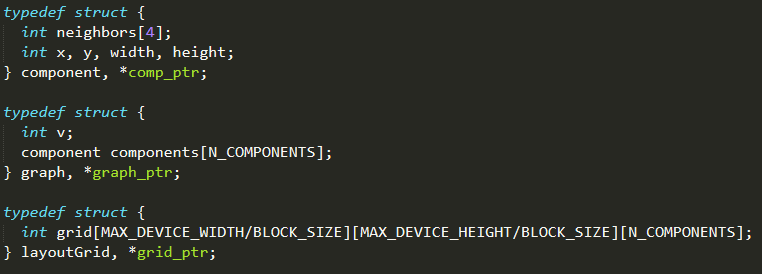
*Calculation Area Penalty:* Area penalty reffers to the total area needed (in the physical grid,) to successfully place all of the components. This pentalty is neccassary because minimizing substrate cost is a important consideration in fabrication protocols. Nevertheless, for the purposes of optimization and analysis of performance, we will be keeping the area of the grid constant. For this reason, we will not be doing area penalty calculations in our algorithm; this is consistent with Huang, who also forwent area cost in the original algorithm.

We will do a full complexity analysis later in the report, but for now it is important to consider the complexity of a single cost claculation. Remember, this cost calculation is redone for every singe pertubation made to the graph; there are N pertubations made per temperature, with the temperature loop complexity asympotic to logN; hence this cost calculation is done N\*logN times. Now, the cost calculation itself involved iterating over all components once (N), as well as over the entire grid area (A). Hence the complexity of the cost calculation is N\*N, and as will be formally analyzed later in the report, a total complexity of O(N^3\*log(N)).

* + 1. **Data Structures**

Our implementation of the SA algorithm was heavily reliant on data structures we defined. These data structures are the bare minimum needed to represent the placement of various components on a grid. It goes without saying that the full Fluigi software workflow included more complex data structures and details, but for the purposes of optimization and analysis we focused our implementation on the minimum case. More complexity introduces more noise in our analysis, so this is reasonable.

There are three data structures needed to realize a placement of components on a grid. The data structures we implemented follow:



The first data structure is the ***component.*** This, in the context of Fluigi, is a microfluidic component. The minimum information needed to define a component are its position (x, y), its size (width, height), and its neighbors. A microfluidic component can only have at most four ports, and hence four neighbors. In the neighbor array, the zero index denotes the top port of the component, the first index denotes the left port, the second index the right port, and the third index the bottom port

The second data structure is a ***graph.*** This graph simply contains information on the total number of vertices in the graph (or components,) and then a matrix of pointers to the components. To the sharp observer, you may notice that the neighbors defined in the component data structure are of type *int*. This is because the integers in the neighbor array refer to the index of said component in the graph data structure. Hence, when referencing a components neighbor, the neighbor must then be indexed in the graph data structure to access the pointer to the component.

The final data structure is the ***grid***. This grid defines the granularity of our solution space, with the block size parameter used to define how small the smallest grid pixel is. This data structure is simply a three-dimensional array. The first two dimensions are the actually spatial layout of the grid. The third dimension is used to store integers that refer to components that reside on that grid location. Because each grid slot can have multiple elements placed on it, we need the third dimension to specify multiple overlapping placements. As before, the components in the grid can be indexed through the graph data structure.

These three data structures are enough to holistically define the optimal placement of components. Notice again that we are concerned with the placement SA algorithm, not the routing. Hence these data structures omit some fields that are used in the routing process.

1. **Serial CPU Implementation**

Our serial implementation of the SA algorithm was a direct implementation of the algorithm described by Huang 2016, ultimately this is the Betz and Rose 1997 SA algorithm. This follows the pseudocode described in section 1. Please refer to the supplements section for the code implementation.

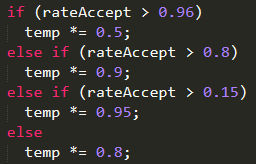
* 1. **Implementation and Analysis**

In this section, we assess the serial algorithm. First, it is important to discuss how this algorithm can be effectively analyzed, as it is a fairly complex routine.

The important idea to realize is that simulated annealing, being a heuristic algorithm, will not converge in a predefined number of steps. Just as in analyzing SOR, it is important to study the complexity of each individual iteration, as well as the effect optimization has toward the number of iterations to convergence. With simulated annealing, the outermost loop controls the slow cooling process, with each outer-loop iteration dropping the temperature variable be a certain amount (this amount the temperature drops by, as we will soon discuss, depends on other algorithm parameters). Within the temperature loop is a for-loop that controls the perturbations made to the graph. For each temperature, the algorithm then iterates across all components, generating a random perturbation to the graph regarding that component, then scoring the cost of this perturbation, and then assessing either to keep the change or discard it based on the Boltzmann factor as described previously. This process is done a predefined number of times per component. Hence the SA algorithm has a structure of three nested loops. In the following analysis, we summarize the complexity of the algorithm *per temperature*.

* We iterate over **N** components.
* For each component, we iterate **C** times, performing random perturbations and calculating cost.
* For each cost calculation:
  + Iterate over all **N** components, calculating the cost associated with distances between components. And after…
  + Iterate over the grid data-structure, calculating the cost associated with overlapping component placement. This ultimately involves **N^2** iteration at maximum over the grid, searching for the overlap of components.
  + Hence, cost calculations are **O(N^2)**
* Hence, ***this algorithm is O(N^3) per temperature***.

Now we are at a point where it is appropriate to talk about the temperature variable, and how we control the “slow cooling” process by decreasing the temperature. Traditionally, temperature in the SA algorithm is decreased by a constant scale factor, pertinent to the problem at hand. These scale factors often range from .01 to .95. In our SA algorithm, temperature is dropped dynamically based on the proportion of successful component perturbations relative to the total number of attempted perturbations. Specifically, we implement: (This is the exact implementation of temperature scheduling proposed by Betz and Rose in the VPR algorithm).



Hence what should be clear here is that the higher the rate of accepted perturbations relative to total attempted perturbations, the more substantial the temperature drop. A very low accept rate at a given temperature, and the temperature will drop less. The reasoning behind this paradigm is that if at a given temperature most perturbations are accepted, then the algorithm was successfully converging to a possible optimum, and the temperature needs to drop more so that the algorithm becomes more discriminatory toward new perturbations; hence the algorithm adjusts itself to begin making more optimal placements. If most perturbations are denied at a given temperature, then the temperature drops by less as the algorithm needs to continue sampling perturbations with the given acceptance rate. Betz and Rose cite that the most optimal accept rate for simulated annealing is 44%, hence this temperature scheduling is made to keep the accept rate close to that percent.

In either regard, the complexity of the above cooling can be understood by looking at worst-case scenarios. In the worst case, the temperature drops by .95 each iteration. The complexity of the other loop can then be expressed as O(log(n)). Hence, the total algorithmic complexity is:

***O( N^3 \* log(N) )***

Now, in measuring the arithmetic intensity, we are concerned with the memory reference pattern of the graph itself. The graph has up to N components that are referenced throughout the algorithm. Under this analysis, the arithmetic intensity is:

***N^3 \* log(N) / N = N\*log(N)***

Overall this arithmetic intensity is higher than that of matrix multiply, which is of order N.

**Complexity*: N^3 \* log(N)***

**Arithmetic Intensity*: N\*log(N)***

Finally, the memory reference pattern involved iterating over the graph of all components. Hence we iterate over an array of N component pointers. In this case, each component is a structure, and hence the memory reference pattern as we iterate through this array is not contiguous in memory. This is not the cleanest memory reference pattern, (it does not have much spatial locality,) and serves as a potential point of optimization. A structure of arrays, in which each array holds information on our components, would be contiguous in memory, and would be a simple optimization worth trying.

* 1. **Results**

Before we discuss performance results, we outline how we measured the performance of this algorithm. Overall, when measuring performance, we are concerned with three metrics: correctness (which is less-inherent in a heuristic algorithm,) runtime to convergence (or iterations to convergence), and runtime per temperature (or runtime per iteration). To measure these quantities, we use the following methodology:

1. Initiate a random device graph with **N** components of edge density **D** (between .2 and .8).
2. Run SA.
   1. Benchmark runtime per temperature / iteration.
   2. Benchmark iterations to convergence.
3. Validate solution (measure final cost).

This will be the general methodology we employ throughout out analysis.

In our serial experiments, we began with the baseline algorithm. This can be found in our supplement section. We then made a new serial version where we flattened one of the inner for loops. We would expect that this should have little to no effect on performance, as we validate with our results. The purpose of flattening the for-loop is such that we can exploit parallelism in our next implementation with multithreads.

The next obvious choice is to unroll the for loops with accumulators. The best place for-loop unrolling would be in the section of the algorithm that calculates the overlap cost. Here the algorithm will iterate over all N components; for each component, it iterates again over all N components, looking for overlaps. This is where we employ loop unrolling. Overall we found that this did not offer much of a benefit in performance.

Performance of un-optimized serial code:

CPE: ~47

Performance of flattened, un-optimized serial code:

CPE: ~46

Performance of 4x unrolled serial code:

CPE: ~46

Performance of 4x unrolled, flattened serial code:

CPE: ~46

As our results show, the general methods we learned for serial optimization have a small effect at best on the SA algorithms’ performance. This does make sense, because the complexity of our algorithm makes it hard to fully take advantage of Amdahl’s law; the maximum speedup we can achieve my parallelizing a small segment of the program is, well, pretty small. To get good performance, we need to make the common case fast, hence involving the optimization of the most intensive and common parts of the algorithm. Immediately, we think to the cost analysis.

1. **Multithreaded CPU Implementation**

This section outlines our approaches to optimizing SA through multithreading. Generally, a multithreaded approach to simulated annealing is difficult, with the biggest challenge being the decomposition and mapping stages of parallelizing the program. Naïvely observing the algorithm pseudocode, we can start to look for regions that can be parallelized. One thought is: how about we parallelize the loop that iterates over all components for a single temperature. Such parallelism would involve each thread being responsible for perturbing a single component and assessing the change in cost, and then making or rejecting the decision. Such an approach would necessitate that the process of perturbing and then accepting or reject a decision be a critical section of the program, otherwise correctness would be lost if all threads were modifying the device simultaneously. Another approach would be to determine if the initial device graph can be partitioned such that parallelism could be exploited in having threads operate on distinct sections of the graph. This would involve partitioning the device graph into multiple sections through min-cuts (hence segregating sections of the graph that are least interconnected or related,) and then exploiting parallelism on each partition.

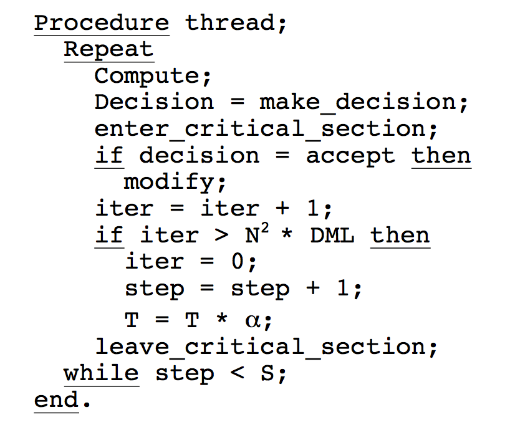
We discuss first our implementation of the naïve approach, and its results and performance. Then we discuss the method of graph partitioning to exploit parallelism.

* 1. **Naïve Approach: Simple multithreading**

Our naïve implementation follows an approach similar to that discussed by Jelenkovich and Plojiak (1998). This is an asynchronous approach, where each thread is responsible for making perturbations to the components.

* + 1. **Implementation and Analysis**

J&P described their algorithms pseudocode as:



Notice that each thread is responsible for perturbing a component, and each thread has the ability to accept this decision and modify the temperature after so many iterations. Already we can see that this parallelism inherently changes the nature of the algorithm, and the immediate first concern with this implementation is correctness. Each thread has the ability to modify the temperature, and each thread is able to modify the device. We focused our multithreaded optimizations on realizing an analogous implementation similar to J&P, but compatible with Huang’s SA methodology. Notable we want to retain as many of the same parameters and algorithmic considerations; for example, in the J&P example above, the temperature is dropped by a constant each time. The Fluigi SA does temperature scheduling based on acceptance rate, and hence we need to capture this temperature scheduling schema.

We began by implementing a naïve 4-proccess multithreading of the inner for-loop. Immediately we found that, while the speedup was huge, the correctness was sacrificed. There were multiple issues we needed to address, namely the scope of variables, the temperature scheduling, and the ability for individual threads to modify the device.

Beginning with variable scope, we had to make sure the general variables specific to a thread were kept private to individual threads.

In terms of temperature scheduling, the tricky aspect we realized was that temperature is scheduled based on acceptance rate of perturbations. With each thread responsible for its own perturbations, acceptances and rejections, we quickly realized that this multithreaded approach would simply not be able to realize the same temperature scheduling as the serial algorithm. So we decided to create our own model of the temperature schedule, to best mimic the temperature schedule first presented by Betz and Rose and discussed earlier.

One way of modeling is this is to consider that should every thread decide to apply a perturbation, the net effect on the temperature should be the same as it would have been in the discrete temperature levels approach. Since our simulated annealing implementation has hard definitions for the amounts to scale the temperature down by as a function of the ratio of approved to disapproved perturbations, one might model it as such:

We start with the assumption that each thread’s temperature change should be equivalent to scaling by a certain factor α. Given this, we consider that in the current implementation, every action being approved results in the temperature being scaled by 0.5. So in our new model, the result of every perturbation (the number of components *N* multiplied by *M*the number of perturbations per temperature per component) being applied should yield:

α*NM*= 0.5.

Hence we can say that an approximation of alpha would be:

α = log NM (0.5)

We employ this approximation of temperature scheduling in our multithreaded approach. Ideally, this would provide a cooling schedule most-similar to the serial implementation.

* + 1. **Results**

Generally, with our new temperature scheduling model and our multithreaded implementation, the big results is that we got SA to converge in far fewer iterations, but the actual CPE for an iteration slightly increased, to our surprise. This is not necessarily good or bad; looking at the cost function, it seemed the results were roughly correct for this optimization. Would this output generate a valid microfluidic for Fluigi? We are doubtful. The fact that the SA algorithm converged in far fewer steps indicated that temperature dropped fast, which indicated many moves were being accepted. In either case, iteration the work increased.

CPE: 60

What we see doing further analysis is that even though the CPE appears lower in the parallel implementation, the amount of work done per element per discrete system temperature is much higher: this is what results in the number of iterations of temperatures being much lower, and thus an overall much faster run time.

In general, the naïve approach hardly offers performance benefits without requiring very heavy handed modification of internal algorithmic parameters, and a sacrifice to correctness. But there are better methods. The biggest issue with the naïve approach steps from the nature of a combinatorial optimization problem itself. Such problems often involve components that are highly interdependence, and hence running an algorithm like SA on data that is interdependent makes it hard to avoid the overhead of communication. For this reason, in the next section we will discuss a graph partitioning method that leverages potential independencies within the graph structure to create concurrency.

* 1. **Clever Approach: Graph partitioning**

As we have seen, optimizing simulated annealing by simply exploiting the per-component parallelism is cumbersome at best, and provides only minimal performance increase. Indeed, to parallelize simulated annealing requires a bit more work in exploiting parallelism of the graph structure itself. This section of the report will discuss the work of Ababei (2009) in using a mincut partitioning method to split the graph into simpler, nearly-independent sub problems. The simulated annealing algorithm is then run in parallel on each partition of the original graph; the simpler sub-problems allow for less iterations to convergence, as well as let runtime per iteration since there are less components in a partition. Further, the partitioning allows each annealing to be concurrently. The main consideration from this approach this becomes communication, coordination and overhead, which is nontrivial. We did not end up implementing this method, as we found the necessary steps were too involved to successfully take under time considerations – that being said, we did thoroughly explore this option and its architectural considerations in regards to our SA algorithm, and this would be the next step in improving performance.

Ababei implemented an optimized parallel version of the Betz and Rose 1997 SA algorithm. This implementation used a mincut partitioning scheme and multithreading scheme to achieve a cited speedup of 2.5x.

To achieve this, the following algorithm was implemented. First, the algorithm performs a “multilevel 4-way partitioning of the top level device description using a hypergraph partitioning tool called hMetis (Karypis et al. 1999). This partitioning is similar to a mincut, as it minimizes the number of nets with edges that span different partitions, thus ensuring that there is a minimal number of dependencies between tasks. In the context of our SA algorithm, this would involve partitioning our initial device in a way such that each partition minimizes the number of ports that connect to a component outside of the partition. Ababei uses a two-level 4 way partitioning scheme, such that the end result is a 16-way partition. Because these partitions are minimally dependent, Ababei is able to implement a multithreaded SA on each partition. These 16 sub-partitions are added to a queue, where 4 worker threads concurrently process the subroutines until they complete the entire queue. Finally, the main thread does a top-down refinement of the whole device under a low-temperature SA run. In pseudocode.

Main Thread

* Level 1 hMetis 4-way mincut partition of graph into 4 sub-problems.
* Level 2 hMetis 4-way mincut partition of sub-problems into a total of 16 sub-problems.
* Add all sub-problems to queue (shared between all worker threads).

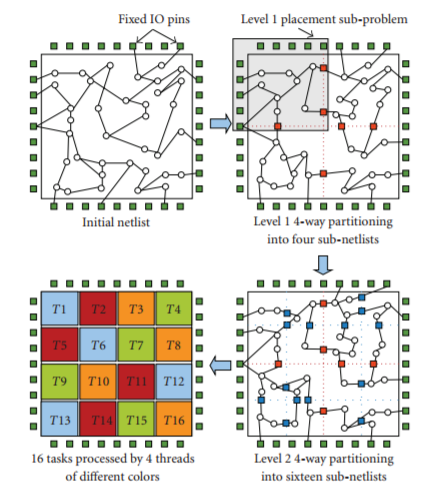
Worker Threads [4-way concurrency]

* Pop next sub-problem from queue.
* Run SA on sub-problem.
* Process until queue is empty.

Main Thread

* Apply a fast top-down SA at low temperatures to refine the final device.

Graphically, the partitioning can be understood as:



Overall this method effective partitions the device into independent segments that SA can run on concurrently. This method would be the obvious next step to take with Fluigi’s SA algorithm.

* 1. **An aside: a brief overview of other parallelization methods in the context of place and route.**

In the domain of FPGA and IC design automation, placement and routing is a substantial and processing intensive step, especially as designs become more complex in their component densities and functional descriptions. Given industry and academic incentive, much work has already been done in optimizing this part of the design automation process. Here we very brieflydiscuss common optimization paradigms for place and route with simulated annealing.

Parallelization techniques for SA can broadly be defined in three categories, as discussed by Ababei. These are

1. Techniques using one copy of the main placement problem.
2. Techniques multiple copies of the main placement problem.
3. Techniques using placement sub problems of the original problem.

The graph partitioning method described earlier is a great example of method 3. Method 1 is representative of our naïve approach, where there exists one copy of the device that gets perturbed and optimized by concurrent worker threads. As we saw, our naïve implementation was not-so-effective, but there are more effective implementations of method 1. Finally, method 2 is an avenue we did not explore. This involved having each concurrent process apply SA to their own unique copy of the device, and then having the main thread run a top down method that combines the results from each process. While performance is an important metric to consider when comparing these methods, it is also important to remember that correctness is critical, and with a heuristic algorithm such a result cannot be guaranteed.

1. **GPU Considerations**

We did not implement the SA algorithm on the GPU, but we did parse the literature to learn more about how the SA algorithm could be mapped to a GPU. In this section, we will discuss at a high level how the simulated annealing algorithm could be implemented on a GPU under architecture considerations. We will also discuss general algorithmic considerations and changes that would need to be made for the GPU approach to work.

One possible GPU scheme is as follows. It would be practical to begin by having each thread block in the GPU grid have the same identical copy of the initial graph or device. Each thread is then responsible for selecting a component in the device; once the thread is assigned a component, it will remain “responsible” for modifications to the component thereafter. Each thread then performs the N iterations per component per temperature - in each iteration, the thread modifies his representative component and calculates the cost of this modification. Each individual thread will also have the ability to modify / accept the change appropriate criteria are met. At the end of N iteration, per component, the threads would need to synchronize, the temperature would need to med dropped, and the process would continue.

Naïve indeed, that approach most likely would not work. Again, the problem being that to implement the GPU algorithm means to fundamentally change the serial algorithm, or the multithreaded algorithm. This means that, even though we might implement a program that runs fast, it is very likely that the program would fail to return a valid solution. For example, one group discussed the process of mapping SA from a serial CPU implementation to a CUDA GPU [Wei et al. 2015]. This group generally found that the biggest issue in mapping the algorithm from a CPU to a GPU is that application specific parameters that worked very well for a CPU will not work well on the GPU. The canonical example from this lab report is the temperature scheduling; this process is fine-tuned with the aim of keeping the acceptance rate of perturbations at roughly 44%, as this was what Betz and Rose found to be optimal. It is likely that none of these findings translate to the GPU, and hence a GPU implementation of the VPR SA algorithm would look much different. Overall, we were not able to do much more than explore the possibilities of GPU use. If time permits in the future, this would be a certain area to implement!

1. **Closing Discussion**

This section is reserved to simply discuss optimization methods, ideas and suggestions that we never got around to implementing. This part of the report will be in a “list” format, with each paragraph dedicated to some thoughts and considerations.

The burning question in our head when we started optimizing Fluigi’s place SA algorithm was, “Why recalculate the entire cost on each perturbation!?” As we discussed, the cost analysis is heavy – O(N^2), and this is done in the innermost loop! The obvious thing to do would be to, instead of recalculating the cost on each iteration, derive an algorithm that simply calculates the delta in the cost. Given that it is known what component is being perturbed, and we know how we perturb it, there should be a clever way to calculate the change in cost from a perturbation without undergoing a full cost analysis. This would make a substantial difference in performance, even simple in the serial case.

Upon further research, we found that indeed, there is a better way. While we did not get around to implementing this method, it would seem like an ideal optimization for both the serial and parallel algorithms. One way to forego redoing the cost calculation on each iteration is to keep a “delta matrix” of N^2, where N is the number of components. Each *i,j* element in the delta matrix corresponds to the cost of swapping those two elements. I should note that in our SA algorithm, we did not “swap” components, rather we moved them by an amount. Hence the same idea of a delta matrix between two swapped components would not work, but there could be a better way to keep a predetermined set of deltas for single perturbations on hand.

In general, the big challenges in optimizing SA was in balancing correctness with new implementations, as well as determining parallel optimizations that allow concurrency on independent subsets of the device. These two tend to do hand in hand. Most methods for parallelizing a serial SA implementation will inherently involve changing the structure of the serial code fundamentally. This can have a significant impact if one is not careful, especially given the randomness components in the algorithm. The beauty of methods like mincut partitioning are in that they segregate the large problem into independent sub problems that SA can be run on concurrently. But even this implementation cannot guarantee the independence of concurrent processes.

Overall, we were able to achieve minor performance increases in SA. Indeed, while we were not too happy with the speedup we achieved, studying simulated annealing revealed a lot of insight into the process of optimizing programs within a more complex ecosystem of software. One fairly important thing we learned through this process is that many optimizations are application specific; hence, if you have a non-general application, the optimizations too, will need to be specific to that application. Even in the context of simulated annealing, many parallelization techniques that would work for one implementation of the algorithm would falter in our version. This overall reveals yet another aspect to performance considerations we did not discuss as much in class; the performance considerations of an application under predefined constrains.

1. **Supplements**

All supplementary content is provided in the .zip submission.

1. **References**

S. Kirkpatrik, C. Gelatt, M. Vecchi. “Optimization by Simulated Annealing,” Science, Vol 220, Issue 4598. doi: 10.1126/science.220.4598.671. May 1983.

C. Ababei, “Speeding Up FPGA Placement via Partitioning and Multithreading,” International Journal of Reconfigurable Computing, vol. 2009, Article ID 514754, 9 pages, 2009. doi:10.1155/2009/514754

G. Karypis, R. Aggarwal, V. Kumar, S. Shekhar, “Multilevel Hypergraph Partitioning: Applications in the VLSI Domain,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol 7, No 1. University of Minnesota. March 1999.

H. Huang, “Fluigi: An end-to-end Software Workflow for Microfluidic Design,” PhD Thesis, Boston University, 2015.

L. Jelenkovic and J. Poljak, “Multithreaded Simulated Annealing,” 20th International Conference on Information Technology. University of Zagreb, Zagreb Croatia. January 1998.

V. Betz and J. Rose, “VPR: a new packing, placement and routing tool for FPGA research,” in Proceedings of the 7th International Workshop on Field Programmable Logic and Applications (FPL '97), pp. 213–222, London, UK, September 1997.