15-418/618 Spring 2022
Assignment 3
Parallel VLSI Wire Routing via OpenMP

Assigned: Tuesday, February 15th
Due: Friday, March 4th, 11:59PM
Last day to handin: Monday, March 7th, 11:59PM

This assignment aims to introduce you to parallel programming using OpenMP and illustrate how the realities of parallel machines affect performance. Although the sequential version of the task you are asked to parallelize is relatively straightforward, there are a number of subtle issues involved in achieving high performance with your parallel code.

1 Policy and Logistics

You will work in groups of two for the problems in this assignment. Turn in a single writeup per group, indicating all group members. Any clarifications and revisions to the assignment will be posted on the class "Assignments" web page, and also announced via our class Piazza web site.

You may clone the Assignment 3 starter code from the course Github:

```
git clone https://github.com/cmu15418s22/Assignment-3.git
```

2 Programming Task: Parallel VLSI Wire Routing

The programming task for this assignment is inspired by VLSI wire routing. You will be given input files of the format shown in Figure 1 that specify the dimensions of a 2D grid, along with the end points for a collection of wires. Your mission is to route the path of each of these wires using Manhattan-style routes (i.e., only 90 degree bends in the routes) that fall within the bounding box specified by these two end points.

```
X_dimension Y_dimension  # dimensions of the 2D grid
number_of_wires          # total number of wires, each of which is described below
X1 Y1 X2 Y2              # coordinates of the endpoints for wire 0
X1 Y1 X2 Y2              # coordinates of the endpoints for wire 1
X1 Y1 X2 Y2              # coordinates of the endpoints for wire 2
...
X1 Y1 X2 Y2              # coordinates of the endpoints for wire (number_of_wires-1)
```

Figure 1: Format of the input file that describes a particular problem. Note that the comments to the right will not actually be in the file: they are just there to describe the contents of the file.

To simplify the problem, you may assume that each wire has **at most two** "bends", but the wires should always make forward progress from one end point to the other. Your goal is not to minimize wire length (all valid routes will have the same length), but rather to **minimize the maximum number of wires that overlap the same position in this 2D grid.**

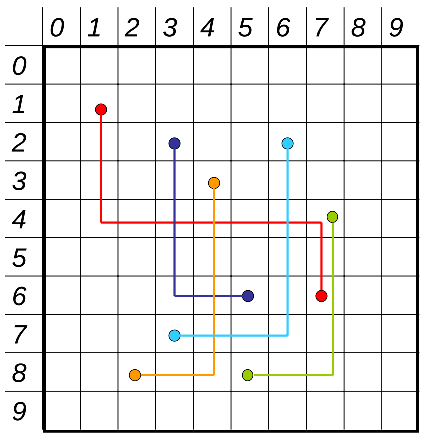
This metric is important in VLSI because it corresponds to the number of layers of metal that are needed in the VLSI process to route the chip. (The more layers of metal, the more expensive the process.) Even if only a single point in the 2D grid requires an additional layer of metal, you still incur the cost of the more expensive VLSI process.

Your algorithm will begin by visiting each wire once to place it in an initial route using a simple greedy heuristic. After the initial routes have been created, you will then make several iterative passes over the wires, possibly improving the route of each wire as you visit it. To illustrate how your algorithm will work, consider the small example input file in Figure 2, which specifies five wires to be routed in a 10x10 grid. (The first wire is between points $(1, 1)$ and $(7, 6)$, the second is between points $(3, 2)$ and $(5, 6)$, etc.)

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

Figure 2: Illustration of an input file.

Figure 3 illustrates a hypothetical snapshot of the state of this algorithm for the input file shown in Figure 2. On the left-hand side of this figure, we see the current routes of the wires in the 2D grid. (Note that the wires at coordinates (7,4), (7,5), and (7,6) are occupying the same space, even though they were not drawn directly on top of each other for the purpose of illustration.) An important data structure for your algorithm is a 2D *cost array*, as illustrated on the right-hand side of Figure 3. Each entry in the *cost array* is simply the number of wires (including end points) that are routed through that position in space. For this particular routing example, we can get away with two layers of metal, since the maximum value in the cost array is two.



Snapshot of Wire Routes

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0
2	0	1	0	1	0	0	1	0	0	0
3	0	1	0	1	1	0	1	0	0	0
4	0	1	1	2	2	1	2	2	0	0
5	0	0	0	1	1	0	1	2	0	0
6	0	0	0	1	2	1	1	2	0	0
7	0	0	0	1	2	1	1	1	0	0
8	0	0	1	1	1	1	1	1	0	0
9	0	0	0	0	0	0	0	0	0	0

Corresponding Cost Array

Figure 3: Example of a potential wire routing (shown on the left) and the corresponding cost array (shown on the right), for the input file given in Figure 2.

3 Further Algorithmic Details

Because VLSI wiring is computationally hard, we will approach an approximate solution using a simulated annealing algorithm.

Metric For Optimization:

Your first priority in choosing a route is to select one that minimizes the maximum cost array value along the route. Given a set of routes with equivalent maximum cost array values, your next priority is to minimize the sum of the cost array values along every wire path (including end points). Beyond that, you can break ties arbitrarily.

For example, if we were to revisit the red wire in Figure 3 with end points at coordinates $(1, 1)$ and $(7, 6)$, a better route for this wire would consist of two segments: one segment running from $(1, 1)$ to $(7, 1)$, and another running from $(7, 1)$ to $(7, 6)$ (with a 90 degree bend at position $(7, 1)$). While this improved route still has a maximum cost array value of 2, it has a lower total cost over its entire path (15 versus 18).

A Single Iteration:

After initially placing the wires using a simple greedy heuristic, you should make several iterative passes over the wires, possibly improving each wire route as you visit it. To simplify the problem, you may assume each wire has at most two "bends". For examples, see Figure 4, which contains wires that are on straight lines, wires with only one "bend" and wires with up to two "bends":

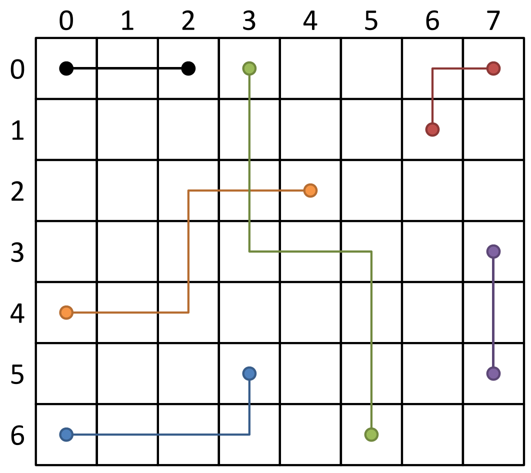


Figure 4: Examples of possible wire shapes. Wires can lie along vertical or horizontal lines, have one bend in their route, or have at most two bends in their routes.

Restricting the number of bends in the wire drastically reduces the search space. Consider a wire with endpoints at $(0, 0)$ and (x, y), with $x > 0, y > 0$. The total number of possible routes is exactly $x + y$. To see this, consider the following route, which first travels along the x-axis: $(0, 0) \rightarrow (1, 0) \rightarrow (1, y) \rightarrow (x, y)$. There are exactly two "bends", represented by intermediate points along the route. Another possible route is $(0, 0) \rightarrow (2, 0) \rightarrow (2, y) \rightarrow (x, y)$ and in general $(0, 0) \rightarrow (a, 0) \rightarrow (a, y) \rightarrow (x, y)$

where $0 < a < x$. In the special case of routing $(0,0) \rightarrow (x,0) \rightarrow (x,y)$ there is actually only one "bend" or intermediate point along the route.

Similarly, if we want to first travel along the y-axis: $(0,0) \rightarrow (0,b) \rightarrow (x,b) \rightarrow (x,y)$ holds, as long as $0 < b < y$, as well as the route $(0,0) \rightarrow (0,y) \rightarrow (x,y)$. You can convince yourself that these are all the routes possible between $(0,0)$ and (x,y).

In general, the total number of routes will be $\Delta x + \Delta y$, assuming the endpoints do not lie on a straight line (then, there is only one possible route according to our scenario).

Consider the wire placement algorithm for a particular wire whose two endpoints are not on a straight line. The "inner loop" code might (but does not need to) follow this basic outline. "Minimum path" refers to the minimum cost path according to our metric for optimization.

1. Calculate cost of the current path, if not known. This is the current minimum path.
2. Consider all paths which first travel horizontally. If any cost less than the current minimum path, that is the new minimum path.
3. Consider all paths which first travel vertically. If any cost less than the current minimum path, that is the new minimum path.

It is acceptable for the inner loop of your implementation to differ from this outline; this is merely one way to approach the problem.

Simplified Simulated Annealing: A real version of this application would iterate until it no longer achieves significant improvements, and it might use simulated annealing to avoid being trapped in local minima. Since our focus in this assignment is on understanding and improving parallel performance rather than generating a high-quality CAD tool, we will simplify things a bit.

- Rather than iterating until the solution quality no longer improves, you may simply iterate for a fixed number of iterations N_{iters} after the initial wire placement. The value of N_{iters} should be an input parameter to your program. By default, please set N_{iters} to 5.
- Rather than performing a true simulated annealing algorithm, you may perform a crude approximation of simulated annealing as follows. You will visit each wire to see whether its route can be improved. With some probability P, where P is an input parameter, you will pick a new route[1] uniformly at random from the set of all possible routes. Otherwise, you will use the improved route your algorithm suggests. Choosing a random route with some probability each iteration can help your algorithm escape poor local minima from which it cannot make any further optimization progress. By default, please set P to 0.1.

  In other words, this simply adds a step to the beginning of your algorithm:

  1. Calculate cost of current path, if not known. This is the current minimum path.
  2. Consider all paths which first travel horizontally. If any cost less than the current minimum path, that is the new minimum path.
  3. Consider all paths which first travel vertically. If any cost less than the current minimum path, that is the new minimum path.
  4. With probability $1 - P$, choose the current minimum path. Otherwise, choose a path uniformly at random from the space of $\Delta x + \Delta y$ possible routes.

[1] It is acceptable to skip this step when the minimum route between two points lies on a straight line.

Output from your program: The output from your program should include the contents of the cost array (named `costs_$inputFileName_$numThreads.txt`) and a representation of the actual wire routes for each wire (named `output_$inputFileName_$numThreads.txt`). The content format for the cost array output file should be a space-delimited matrix of numbers:

```
<maxX> <maxY>
<c11> <c12> ... <c1n>
<c21> <c22> ... <c2n>
...
<cm1> <cm2> ... <cmn>
```

where `<maxX>` and `maxY` are the x and y dimensions of the grid. The content for the wire routes output file should be in the format used in Figure 1:

```
<maxX> <maxY>
<# of wires>
<w1x1> <w1y1> <w1x2> <w1y2> <w1x3> <w1y3> ...
<w2x1> <w2y1> <w2x2> <w2y2> <w1x3> <w1y3> ...
...
```

The computation for writing these out can be done sequentially (on one thread) after the parallel algorithm completes, and it should not be counted against your parallel speedup. In your writeup, please present the routes and the cost array in a graphical format (not as dumps of text or numbers). We have provided a visualization tool, located in `code/WireGrapher.java`, to help you with this part of the assignment. You can compile and run WireGrapher using the following commands (make sure to SSH with the `-Y` flag):

```
$ javac WireGrapher.java
$ java WireGrapher [input]
```

4 Parameter Names

Your executable should accept the following parameters as command line arguments:

```
<executable> -f <filename> -n <num_threads> [-p <P>] [-i <N_iters>]
```

where both `<P>` and `<N_iters>` are optional arguments.

5 Logistics

You should parallelize these applications using OpenMP to create a shared-memory application. Further details on how to use the OpenMP can be found in `tutorials/openmp.pdf`, along with several example C programs in `examples`. You can also take a look at the following documents: The OpenMP 3.0 specification and An OpenMP cheat sheet .

We will make a number of different input files available to you (using the format shown in Figure 1) in the `code/inputs` directory. We will provide inputs at a variety of different sizes. Please start with the test files in the `code/inputs/testinput` first in order to debug your program. Once it appears to be working, try the files in `code/inputs/timeinput`. You will need to report your performance on the `code/inputs/timeinput` files as part of your assignment submission.

We provide a script called `code/validate.py` to validate the consistency of output wire routes and the cost array. You can run `python2 validate.py -h` for instructions on how to use the script.

6 The Parallel Machines

You should use the machines in the Gates cluster for your initial development and testing. Host names for these machines are `ghcX.ghc.andrew.cmu.edu`, where X is between 47 and 86.

However, you will collect your final performance numbers on the PSC Bridges-2 RM (Regular Memory). For more information on using these machines, please see the `tutorials/machines.pdf` document. You may also find it helpful to refer to the Bridges-2 User Guide. **Please do not implement your solution on these machines! We are given a limited amount of computing resources to test on the PSC and we will need them for assignment 4. All development should be done on the gates machines.** Also, every time you finish your testing on PSC machines, don't forget to terminate your interactive session so that the allocated resources can be released in a timely manner.

Warning: You are welcome to test your program on PSC once it is working on GHC machines, but any blatant abuse of PSC resources (e.g., to brute force a solution) will result in a score of zero on the assignment.

7 Measuring Performance

Execution time: To evaluate the performance of the parallel program, measure the following times:

1. *Initialization Time:* The time required to parse the command line arguments, read the input file, initialize your data structure, and perform the initial wire placement. Start timing when the program starts, and end just before the main computation starts.

2. *Computation Time:* This is strictly the time to perform the N_iters simulated annealing algorithm. Start timing when the main computation starts and finish when all of the results and metrics have been calculated. You do **NOT** need to include writing to files in Computation Time.

Note that: *Total Time* = *Initialization Time* + *Computation Time*. *Speedup* is calculated as $\frac{T_1}{T_p}$, where T_1 is the time for one thread, and T_p is the time for P threads. *Computation Speedup* uses only Computation Time, and *Total Speedup* uses the Total Time. The starter code includes some basic timing code that you may use.

Cache misses: In this assignment, we will also be using hardware counters to report certain performance metrics at the program level. In particular, we will measure the number of cache misses of your parallel programs using `perf`, a performance analysis tool for Linux. You can use the following command to measure the total cache misses across all threads:

```
$ perf stat -e cache-misses $PROGRAM
```

8 Performance Analysis

The goal of this assignment is for you to think carefully about how real-world effects in the machine are limiting your speedup, and how you can improve your program to get better performance. If your performance is disappointing, then it is likely that you can restructure your code to make things better. **We are especially interested in hearing about the thought process that went into designing your program, and how it evolved over time based on your experiments**.

Your report should include the following items:

1. A detailed discussion of the design and rationale behind your approach to parallelizing the algorithm. Specifically try to address the following questions:
   - What approaches have you took to parallelize the algorithm?
   - Where is the synchronization in your solution? Did you do anything to limit the overhead of synchronization?
   - Why do you think your code is unable to achieve perfect speedup? (Is it workload imbalance? communication/synchronization? data movement?)
   - At high thread counts, do you observe a drop-off in performance? If so, (and you may not) why do you think this might be the case?
2. The output of your program (shown graphically using the `WireGrapher` tool) for the input circuits in `code/inputs/timeinput`.
3. A plot of the *Total Speedup* and *Computation Speedup* vs. *Number of Threads* ($N_{threads}$) for the input circuits in `code/inputs/timeinput`, where $N_{threads}$ = 1, 4, 16, 64, and 128. Please measure your speedup on PSC Bridges-2.
4. A plot of the total number of *cache misses* for the entire program vs. *Number of Threads* ($N_{threads}$) for the input circuits in `code/inputs/timeinput`, where $N_{threads}$ = 1, 4, and 16. You can measure this using `perf stat -e cache-misses $PROGRAM` on the Gates cluster machines (unfortunately, this measurement is not available on PSC Bridges-2).
5. A plot of the arithmetic mean of *per-thread* cache misses vs. *Number of Threads* ($N_{threads}$).
6. Discuss the results that you expected for all the plots in Questions 2-5 and explain the reasons for any non-ideal behavior that you observe.
7. A plot of the *Total Speedup* and *Computation Speedup* on 128 threads with respect to 1 thread where the value of P (i.e. the probability of forcing a wire to be rerouted through simulated annealing) is varied between 0.01, 0.1, and 0.5, for the input circuits in `code/inputs/timeinput`. (If running with 1 thread is too slow, you are free to change the baseline to 4 threads or even 16 threads)
8. Discuss the impact of varying P on performance, explaining any effects that you see.
9. A plot of the *Total Speedup* and *Computation Speedup* on 128 threads where the input problem size is varied. There are different ways to vary the problem size, for example, grid size, number of wires, the average length of wires or even the layout of the wires. Here, please explore the different grid size and number of wires. Please report the results for the input circuits in `code/inputs/problemsize`. (Again, if running with 1 thread is too slow, you are free to change the baseline to 4 threads or even 16 threads)

10. Discuss the impact of problem size (both grid size and number of wires) on performance.

9 Grading

You will be graded on the report items in Section 8. While we do ask you to include your performance on the inputs provided in `code/inputs/timeinput`, **we won't put too much emphasis on performance. We care a lot more about your thought process in parallelizing the algorithm and your analysis of the results**. That being said, here are the results for the reference solution executed on PSC non-shared RM with $P = 0.1$ and $N_{iters} = 5$ to serve as a performance target (Again, you do not need to hit these benchmarks - there isn't a specific computation time or target speedup that we would like to see. We want to understand your thought process and analysis!):

easy_4096	medium_4096	hard_4096
1.66	15.98	25.63

Figure 5: Reference computation time (in seconds) of sequential program on PSC non-shared RM

Threads	easy_4096	medium_4096	hard_4096
4	2.51	2.34	2.35
16	8.12	7.53	7.91
64	15.33	21.14	20.34
128	1.77	10.02	17.23

Figure 6: Reference computation speedups on PSC non-shared RM

10 Hints

- For the initial wire placement, you can use a greedy approach by considering the cost along each wire path, or you can arbitrarily place the wire in any legal position.
- There are two potential dimensions of parallelism in this assignment. One is parallelism across different wires. The other is parallelism across different candidate paths of one wire.
- Collecting results on PSC machine with a high thread count might help you identify bottlenecks related to synchronizations.
- Performance monitoring tools such as perf are always your friend. For example, if you want to profile which line of your code generates most cache misses, you can record cache miss event by running `perf record -e cache-misses $PROGRAM`. Then you can run `perf report`, which provides you with a command line interface to analyze your assembly code in detail. You might want to switch the compiler optimization flag from `-O3` to `-O0` before you conduct the above profiling.

11 Hand In

If you are working with a partner, please form a group on Autolab before submitting your assignment. One submission per group is sufficient.

Your submission should be a `handin.tar` file with a single directory titled `andrewid1_andrewid2` consisting of your source code for `wireroute.cpp` and `wireroute.h` as well as your outputs for each of the files in `code/inputs/timeinput` with `-n 128`. Please follow the directory structure below.

```
andrewid1_andrewid2
├── code
│   ├── wireroute.cpp
│   ├── wireroute.h
│   └── Extra helper files, if applicable.
└── outputs
    ├── output_easy_4096_128.txt
    ├── output_hard_4096_128.txt
    └── output_medium_4096_128.txt
```

Also remember to do a good job on *commenting your code*, which is very important.

Please upload your writeup in `.pdf` format to Gradescope (one submission per team) and select the appropriate pages for each part of the assignment. After submitting, you will be able to add your teammate using the add group members button on the top right of your submission. Your report should include the items listed in Section 8.