Due: Wednesday, October 10th, 2018, 11:59 PM

Final Deadline: Saturday, October 13th, 11:59 PM

Maximum Late Days: 3

As you may have realized with Assignment 2, the earlier you start this the better.

Overview

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

Environment Setup

You will collect your numbers on the Latedays cluster. Please read the README file to learn about how to submit a job to latedays. The code will be run on Xeon Phi, and we'll be using the offload mode in the assignment.

More information about running on Latedays can be found in the following link:

http://15418.courses.cs.cmu.edu/spring2016/article/6

You will ssh with the following:

```
ssh ANDREW_ID@latedays.andrew.cmu.edu
```

You'll notice that when you log in, you won't be in AFS. This is just fine, **untar the starter code in the directory you're in right when you log in or a subdirectory that you create yourself.**

Everything you need to know about how to run your code, how to use OpenMP, and what is provided in the starter code is in the README! Download the starter code from Autolab.

You will need to add the following lines to your ~/.bashrc file:

```
module load gcc-4.9.2
export PATH="/opt/intel/bin:${PATH}";
source /opt/intel/bin/compilervars.sh intel64
```

Policy and Logistics

You will work in groups of two people in solving the problems for this assignment. Turn in a single writeup per group, indicating all group members. If any revisions or changes are made to the assignment, you'll hear from an email and a post on Piazza.

To get started, download assignment3-handout.tar from Autolab to a directory accessible only to your team. In the following, ASSTDIR refers to the directory that the handout is unpacked with:

```
> tar xzf assignment3-handout.tar.gz
```

Please make sure you read this document and ASSTDIR/README before you start working on the assignment. Seriously.

Introduction: VLSI Wire Routing

VLSI ("Very-Large Scale Integration") refers to the process for designing and creating integrated circuits (ICs) on computer chips. A common step in VLSI design is routing **wires** to connect components on the surface of the chip. When routing these wires, an important goal is to **minimize the maximum number of wires that overlap the same position on the surface of the chip (which is a 2D grid).** This metric is important 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.

For this assignment, we will consider **a set of relatively short wire paths** between two endpoints. It is clear that there are a number of wires that share the shortest distance between two endpoints, which is just the "Manhattan Distance" between them (i.e. the sum of the absolute values of the differences in both dimensions: |x1-x2| + |y1-y2|, where (x1,y1) and (x2,y2) are the coordinates of the endpoints). For each input data set, a parameter delta is specified, and for each pair of endpoints, we will consider all wire paths whose lengths are no longer than **shortest path distance + delta** (delta is guaranteed to be an even number and will be less than 10). For simplicity, we will consider such paths with at most two "bends". This will become clearer with some examples. With that in mind, consider the following 10x10 grid of wires:

With the wire routing shown above, there are three coordinates where wires overlap: (2,6), (3,5), and (3, 6). Hence the cost at each of these coordinates is 2 (i.e. the number of wires occupying that same location). Overall, the maximum cost that we find anywhere in the 10x10 grid above (given the illustrated routing of wires) is 2, which means that the VLSI designer would need to pay for a process that supported 2 layers of metal.

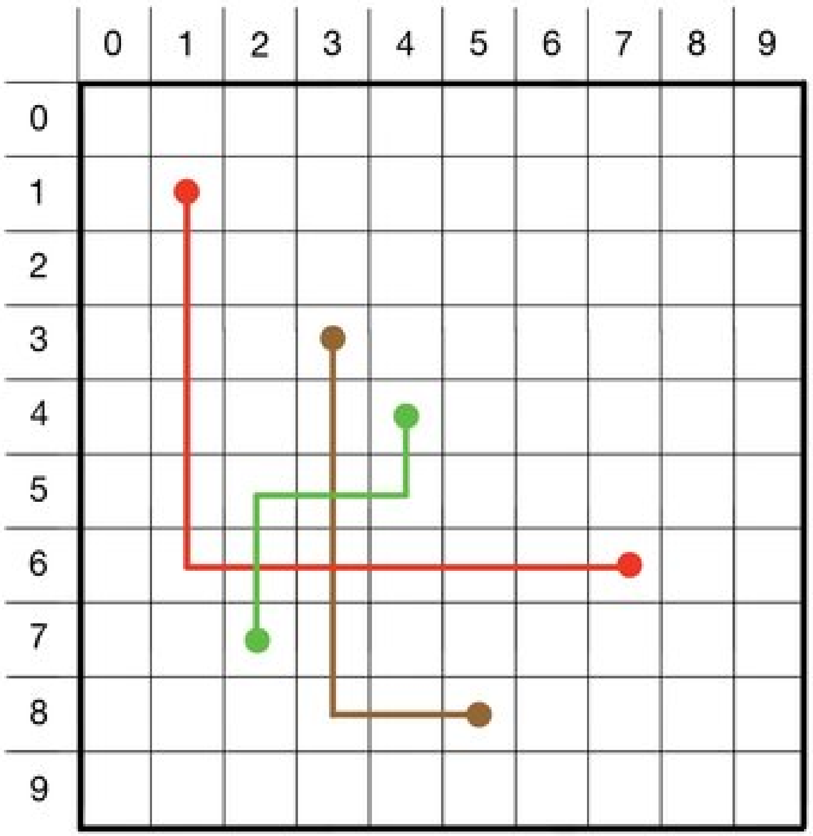


Figure 1: Bad Wireroute

Is there a better way to route the wires? If we still consider only wire routes that minimize the wire path distance, there is indeed a better choice of wire routes. The figure below shows an example of wire routes that achieve a maximum overlap of just 1 (i.e. no coordinate is occupied by more than 1 wire):

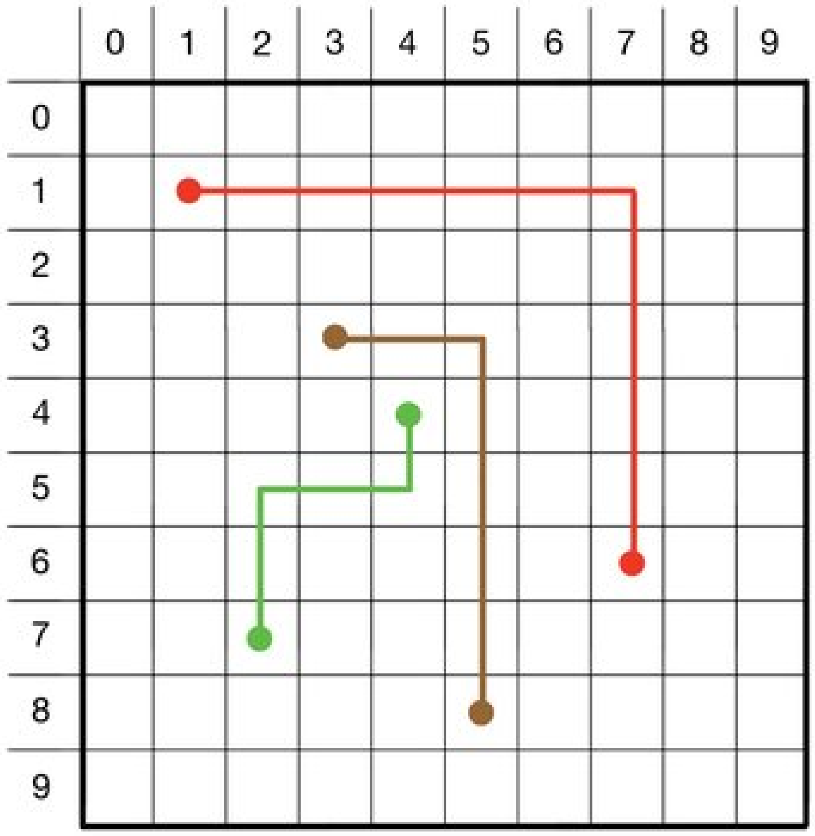


Figure 2: Good Wireroute

In general, given arbitray sets of wire endpoints, it is not always possible to achieve a maximum overlap of just 1, but our goal is to minimize the maximum overlap to the extent possible. In particular, the solution would be quite different with various delta settings. For example, given the following input, the best solution for delta = 0 and delta = 2 would be **very different**.

In this assignment, you will be designing and implementing a parallel algorithm that attempts to minimize the maximum overlap for a set of wire routes between endpoints specified by an input file, under different delta settings. You will be

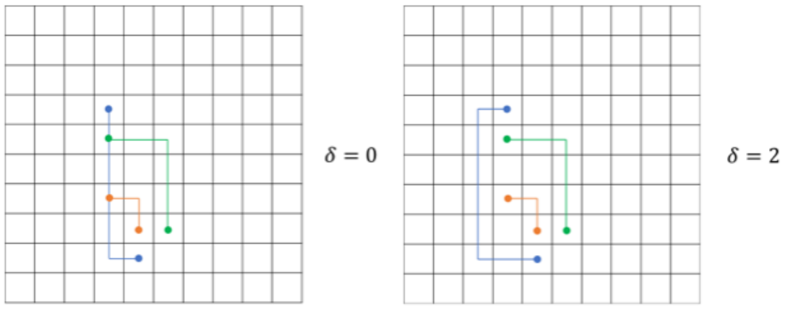


Figure 3: With Delta Wireroute

using OpenMP to implement your algorithm, using the Shared Address Space parallel programming model.

Specification

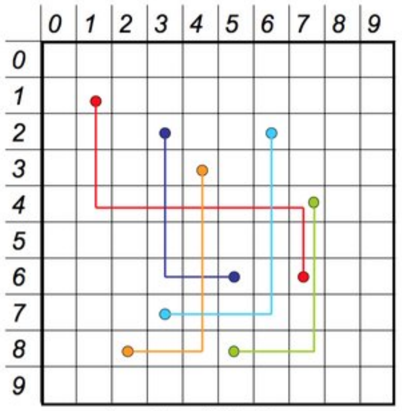
Your **first priority** in choosing a route is to select one that **minimizes the maximum cost array value** along the route. Given a set of routes with equivalent maximum cost array values, your **second priority** is to **minimize the sum of all values in the cost array where the value is greater than 1**. Here is pseudo-code for calculating this aggregate cost value:

```
aggregate_cost = 0;
for x = 0 to N-1
   for y = 0 to N-1
      if (cost_array[x][y] > 1)
           aggregate_cost += cost_array[x][y];
```

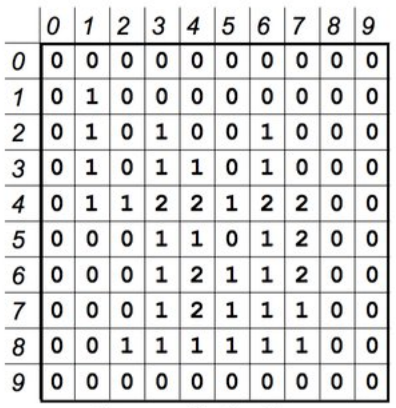
Beyond these two priorities, you may break ties arbitrarily.

An important data structure for your algorithm is a 2D **cost array**, as illustrated below. Each entry in the cost array is simply the number of wires (including end points) that are routed through that position in space. For the example below, the maximum overlap, or **maximum cost array value**, is 2.

As mentioned earlier, you may assume each wire has at most two "bends". For example, in the image below, there are wires that are straight lines, wires



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 4: Wireroute

with only one and two "bend(s)". Restricting the number of bends in the wire drastically reduces the search space.

The Algorithm

The focus of this assignment is on the parallelization of this application rather than developing the algorithm itself. Because of this, we will cover how we want you to write this algorithm before parallelizing it.

Consider the wire placement algorithm for a particular wire whose two endpoints are not on a straight line. The "inner loop" code might follow this basic outline. "Minimum path" refers to the minimum path using our metric for optimization; remember to break ties using the rules outlined previously.

1. Calculate the cost of the current path, if not known. This is the current minimum path.
2. Consider all paths which first travel horizontally. If any costs less than the current minimum path, that is the new minimum path.
3. Consider all paths which first travel vertically. If any costs 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 achieved significant improvements, and it might use simulated annealing to avoid being

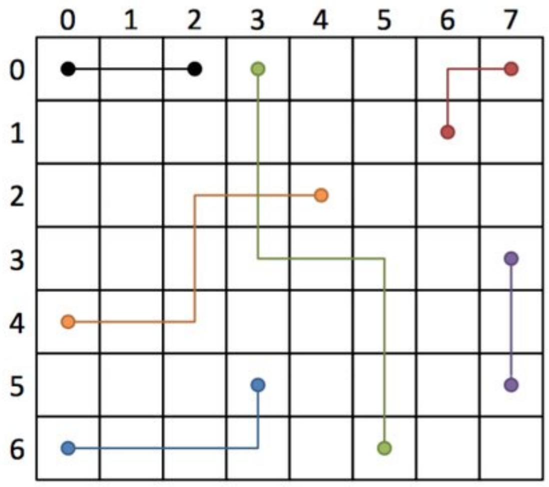


Figure 5: Bends

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 quality of solution is no longer improving, you will simply iterate for a fixed number of iterations (N_iters) after the initial wire placement. The value of N 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 will 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, you will pick a new route uniformly at random from the set of all possible routes. Otherwise, you will use the improved route your algorithm suggests. This simply adds a step to 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 costs less than the current minimum path, that is the new minimum path.
    3. Consider all paths which first travel vertically. If any costs 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 dx + dy possible routes.

Remember, **the goal of this assignment is not to create a new algorithm!** Your goal is to use this algorithm and see how fast you can make it.

Implementation Details

Executable Format

You will write an executable program that should accept the following parameters as command line arguments:

./wireroute -f FILENAME -n NUM_THREADS [-p P] [-i N_ITERS]

where both P and N_iters are optional arguments.

With the provided Makefile, your executable will be called wireroute. Your code should be able to parse out the -f, -n, and optional -p and -i arguments.

Your starter code will handle quite a bit of this already. It will handle parsing in these arguments and provide them to you to write your code.

Input File Format

When your executable takes in a filename as an argument, the file will be formatted as follows

```
X_dimension Y_dimension   # dimensions of the 2D grid
delta                    # delta, an even number less than 10
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)
```

Your executable should be able to parse this input file to fetch the grid dimensions, number of wires, and the start- and end-points for each wire. Your script can assume that the filename given to your script indeed refers to a file that follows this format according to this spec.

Output File Format

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 ...
c21 c22 ...
...
```

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 following format:

```
maxX maxY
delta
#_of_wires
w1x1 w1y1 w1x2 w1y2 w1x3 w1y3 w1x4 w1y4
w2x1 w2y1 w2x2 w2y2 w2x3 w2y3 w2x4 w2y4
...
```

Please note that **these coordinates consist of the start point, bend points (if any), and end points in that order**. That's why the example above shows 4 coordinates per wire (2 bends per wire), but there can be 2-4 coordinates per wire based on the number of bends a wire has.

The computation for writing these out can be done sequentially (on one thread) after the parallel work 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 would also like you to report the maximum value found anywhere in the cost array. We have provided a visualization tool, located in ASSTDIR/code/grapher/WireGrapher.java, to help you with this part of the assignment You can compile and run WireGrapher using the following commands:

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

More specifics on what we want in your writeup are in the **Performance Analysis** section.

The starter code also contains a validate script that will check that your costs file and outputs file match up correctly. Details on how to run this are in the README. **When you submit your results to Autolab, they must validate with this script.** More on this in the **Hand In** section. ## Measuring Performance ##

Execution time: To evaluate the performance of the parallel program, measure the following times using gettimeofday()

1. Initialization Time: the time required to do all the sundry initialization, read the command line arguments, and create the separate processes. Start

timing when the program starts, and end just before the main computation starts.

2. Computation Time: this is strictly the time to compute the result. (It does not include the time necessary to print them out.) Start timing when the main computation starts (after all the processes have been created), and finish when all of the results have been calculated.

Note that: Total Time = Initialization Time + Computation Time. Speedup is calculated as T1/Tp, where T1 is the time for one processor, and Tp is the time for P processors. Computation Speedup uses only Computation Time, and Total Speedup uses the Total Time.

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

```
$ perf stat $PROGRAM
```

You can use the following command to measure the cache misses:

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

"-e" is the option to specify the events we want to report, you can see the list of events using:

```
$ perf list
```

When you're running on the Xeon Phi, the job output will have the cache-miss count at the bottom after your print statements for you already. This will make more sense after reading the README in the starter code.

Here is a tutorial in perf if you're curious: https://perf.wiki.kernel.org/index.php/Tutorial

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 taken 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) for the different input circuits. You can generate this using the WireGrapher.java program explained in the **Implementation Details** section.
3. A plot of the Total Speedup and Computation Speedup vs. Number of Processors (Nprocs). Use Nprocs = 1, 4, 16, 64, 128 and 240. Please submit your program to latedays cluster for the experiment, see ASSTDIR/README for more information about how to submit to latedays.
4. A plot of the total number of cache misses for the entire program vs. Number of Processors (Nprocs).
5. A plot of the arithmetic mean of per-thread cache misses(from perf stat -e cache-misses $PROGRAM) vs. Number of Processors (Nprocs).
6. Discuss the results that you expected for all the plots in Question 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 240 threads with respect to 1 thread where the value of P (i.e. the probability of forcing a wire to be rerouted ala simulated annealing) is varied between 0.01, 0.1, and 0.5. (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 240 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. Plese report the result of input in ASSTDIR/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.

Grading

We will be grading based on your report on times in the **Performance Analysis** section and the performance will be graded using the inputs in **ASST-DIR/code/inputs/timeinput**, running with 64, 128, and 240 threads. Details on reference solution performance will be released a week before the assignment is due.

60% of your grade will be based on your performance and timings while 40% of your grade will be based on your writeup. We **really** care about your thought process on parallelizing the algorithm and your analysis on the result. **Do NOT make the mistake of focusing solely on the parallelization and ignoring the writeup.** Even if you reach the benchmark, you won't end up with a good score if you can't explain why your final solution is good and how you got there.

If you're struggling to reach the benchmark, it's in your best interest to spend time creating a quality writeup.

Yes, we will be checking for style. We'll be relatively lenient, but you will be docked if your style is particularly bad. If you write all of your code in the main function, you will lose points.

We will also check that your job outputs print out max cost and aggregate cost values of the resulting cost arrays. We will check that the values aren't unreasonably high, you want this as low as possible.

Hand In

This is important, read this carefully

Electronic submission through Autolab: Your submission should be a .tar file (named as **andrewid1_andrewid2.tar.gz**, that has a single directory (named as **andrewid1_andrewid2**) consisting of the following files and directories:

- Your entire **code** directory. **Please run make clean before tarring and submitting**. If we copy the code directory you submit to our machine, it should still compile and run without a hiccup.
- In the **file_outputs_submit** directory within the **code** directory, put in your resulting output and costs files for the benchmark tests in the **Grading** section. There should be 18 files submitted here, 2 per benchmark/thread count combination. **These must pass the validation script.**
- In the **job_outputs_submit** directory within the **code** directory, put in your resulting job outputs for the benchmark tests. There should be 9 files submitted here, 1 per benchmark/thread count combination. The starter code by default prints computation and total times. **You also need to**

add print statements that give us your max cost and aggregate cost of your resulting cost array.
- **Read the previous bullet again**, I suspect a lot of people are going to miss this detail.
- **writeup.pdf**, your writeup in .pdf format. Your writeup should include the items listed in the **Performance Analysis** section.

To create this tar file, run this command:

```
> tar czf andrewid1_andrewid2.tar.gz andrewid1_andrewid2
```

Pieces of Advice

- This assignment can be quite time consuming, so try to start as early as you can. Even if you can learn how to run on the Xeon Phi, that's still one less thing to worry about later. Try to get the sequential version of the algorithm working correctly with the validate scripts as soon as you possibly can.
- The README files in the starter code are just as important as this writeup. Make sure you read those really carefully.
- **Modular code is VERY important, because you'll be implementing this same algorithm with message-passing in Assignment 4. You will be thankful a few weeks from now if your code is reusable with plenty of helper functions.** Do NOT do all of your code in the main function.
- Try thinking about which part of the algorithm you'd want to parallelize. List your options. What are the tradeoffs with each choice? Which version can improve the most from parallelizing across threads?
- If you know how to use Git, use it. It can be very helpful when experimenting with different parallelization methods. Once you finish the sequential version, make a commit and try different parallelization methods in different branches that stem from that sequential commit.