15-418/618, Spring 2025
Assignment 3
Parallel VLSI Wire Routing via OpenMP

Assigned:	Wed., Feb. 12
Due:	Wed., Feb. 26, 11:59 pm
Last day to handin:	Sat., Mar. 1

Overview

In this assignment and the next assignment, your mission will be to write parallel versions of the same application using two different parallel programming models: in this assignment, you will be using *OpenMP* to write parallel code using the *shared address space* model, and in Assignment 4, you will be using *MPI* to write a *message passing* version of the same application.

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, given how the realities of parallel machines affect performance. You will need to instrument your code to determine where the most time is spent in computation and evaluate where optimizations are most valuable. You will also need to focus on avoiding sequential bottlenecks, memory contention, and workload imbalance. We strongly recommend that you go back and review the material in lectures 6 and 7 (on various performance optimization techniques), since they are likely to be useful for these assignments. Good luck!

Before you begin, please take the time to review the course policy on academic integrity at:

`http://www.cs.cmu.edu/~418/academicintegrity.html`

Download the Assignment 3 starter code from the course Github using:

```
git clone https://github.com/cmu15418s25/asst3.git
```

OpenMP Resources

The OpenMP standard has been widely adopted by a number of compilers (including GCC), on a variety of platforms. While OpenMP programs can be written in several programming languages (including C, C++, and Fortran), in this assignment, you will be working in C++.

```
X_dimension Y_dimension  // dimensions of the 2D grid
number_of_wires          // total number of wires (N)
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 N-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.

Like many standards, it started with a small core of simple and powerful concepts but has grown over the years to contain many quirks and features. You only need to use a small subset of its capabilities. A good starting point is our short introduction to OpenMP. OpenMP.org has a list of several useful OpenMP tutorials. For example, this OpenMP tutorial from Lawrence Livermore National Laboratory (llnl.org) is a useful reference. We will also have a recitation that discusses how to use OpenMP. You can also review the OpenMP parallel pragma directives documentation created by IBM.

For any C++ questions (like what does the *virtual* keyword mean), the C++ Super-FAQ is a great resource that explains things in a way that's detailed yet easy to understand (unlike a lot of C++ resources), and was co-written by Bjarne Stroustrup, the creator of C++!

Programming Task: Parallel VLSI Wire Routing

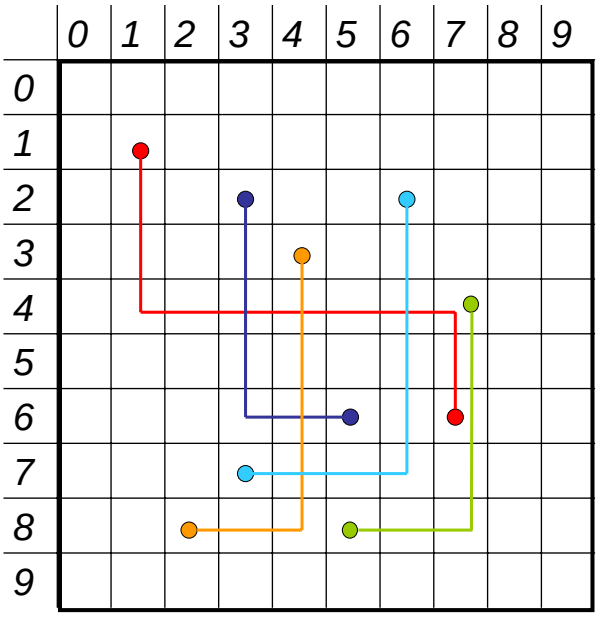
The programming task for this assignment is inspired by VLSI wire routing. You will be given input files of the format shown in Figure 1 that specify the dimensions of 2D grid, along with the end points for a collection of wires. Your mission is to route the path of each of these wires using Manhattan-style routes (i.e., only 90 degree bends in the routes) that fall within the bounding box specified by these two end points. To simplify the search space, your routes can contain no more than two bends (i.e. no more than three segments). Your goal is not to minimize wire length (since all valid routes will have the same length), but rather to minimize the number of wires that *overlap* at the same positions in this 2D grid, since this cost grows as N^2 (where N is the number of overlapping wires). Note that the end points count as occupying a given point in space. 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.

```
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: Example input file.

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 5 wires to be routed in a 10x10 grid. (The first wire is between points $(1, 1)$ and $(7, 6)$, the second is between



	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

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

points $(3, 2)$ and $(5, 6)$, etc.) Figure 3 illustrates a hypothetical shapshot of the state of this algorithm for the input file shown in Figure 2. On the lefthand side of Figure 3, 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 *occupancy matrix*, shown on the righthand side of Figure 3. Each entry in the *occupancy matrix* 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 *occupancy matrix* is two.

Further Algorithmic Details

Routing Constraints.

There are two constraints on how wires are routed: (i) the route must be contained **within the bounding box** defined by the two end points; and (ii) there can be **no more than two bends** in the route. Figure 4 illustrates examples of potential wire routes, including straight lines (when the two end points lie along either the same row or column), wires with only one bend, and wires with two bends.

By restricting the number of bends in the wire (to no more than two), we drastically reduce the search space. Consider a wire with endpoints at $(0, 0)$ and (x, y), with $x > 0, y > 0$. In this example, the total number of possible routes is exactly $x + y$. To see this, consider the following route, which first travels along the

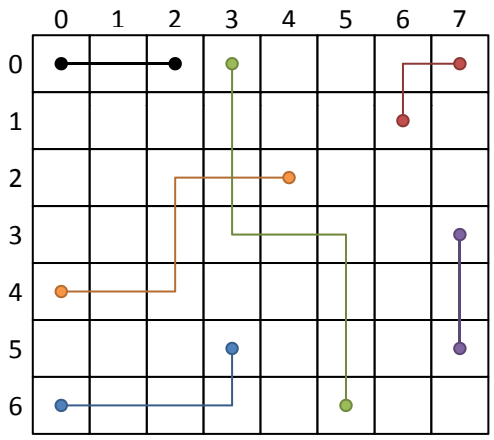


Figure 4: Examples of possible wire shapes. Wires can lie along vertical or horizontal lines, have one bend in their route, or have at most two bends in their routes.

x-axis: $(0,0) \rightarrow (1,0) \rightarrow (1,y) \rightarrow (x,y)$. There are exactly two "bends", which can be representated 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 (in which case there is only one legal route).

Metric For Optimization

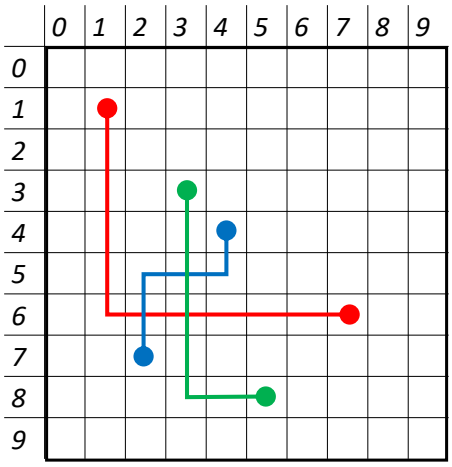
As we are choosing wire routes, our goal is to minimize the VLSI cost due to wires overlapping. At a particular position in the occupancy matrix, the cost is the *square* of the number wires occupying that location. The overall cost is the sum of these individual costs over all matrix elements. In other words, our optimization goal is to find wire routings that minimize this sum-of-the-squares equation:

$$OverallCost = \sum_{i=0}^{R-1} \sum_{j=0}^{C-1} OccupancyMatrix[i][j]^2 \tag{1}$$

where R and C are the numbers of rows and columns in the matrix, respectively.

For example, Figure 5 shows an example of two different options for routing three different wires. Figure 5a shows a sub-optimal routing where three different locations contain two overlapping wires. (The associated occupancy matrix is shown in Figure 5b.) The overall cost metric for this sub-optimal routing is 32, since three locations have a cost of 4 (i.e. 2^2).

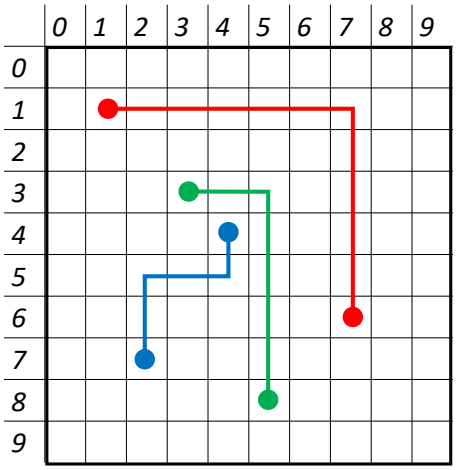
Figure 5c shows a better way to route the wires in this example that avoids having overlapping wires. The cost metric of this improved routing is 26, which is the minimum cost for these particular endpoints since



(a) Sub-optimal wire routing.

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

(b) Occupancy matrix for the sub-optimal wire route on the left (overall cost = 32).



(c) Improved wire routing.

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

(d) Occupancy matrix for the improved wire route on the left (overall cost = 26).

Figure 5: Illustration of how improved wire routing can reduce the cost metric.

no entry in the occupancy matrix (shown in Figure 5d) is greater than one.

A Single Iteration

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 path using our metric for optimization; you may break ties arbitrarily.

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 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 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_{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 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[1] randomly from a uniform distribution of the set of all possible routes. Otherwise, you will use the improved route your algorithm suggests.

  This simply adds a step to the beginning of your algorithm:

  1. With probability P, choose a route randomly from the set of $\Delta x + \Delta y$ possible routes (using a uniform distribution). Otherwise (i.e. with probability $1 - P$), continue on to the remaining steps to choose the route.
  2. Calculate cost of current path, if not known. This is the current minimum path.
  3. Consider all paths which first travel horizontally. If any costs less than the current minimum path, that is the new minimum path.
  4. Consider all paths which first travel vertically. If any costs less than the current minimum path, that is the new minimum path.

Two Sources of Parallelism: Within and Across Wires

There are two opportunities for exploiting parallelism in this algorithm, and you are to explore both of them.

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

Within Wires: In the process of finding the best routing option for a particular wire (given its endpoints), there are a number of potential routes to be considered. As discussed earlier, unless the two endpoints lie on the same column or row in the grid, there should be $\Delta x + \Delta y$ possible routes to explore. If the endpoints are non-trivially far away from each other, this can potentially provide a signficant source of parallelism. With this first strategy, you will exploit parallelism by exploring these routes (for a given wire) in parallel.

Across Wires: A second source of parallelism is the fact that there are many wires to be routed on each time step, and the routing decisions for separate wires can be made in parallel. With this latter approach, the "tasks" (using our terminology from lecture 6) are *wires* (or entire batches of wires, as we will discuss later), in contrast with the within-wires approach above (where the tasks are individual *routes* for given wires).

You are to write a single program that uses a runtime switch to set the parallelism mode: "`-m W`" for *within* wires, and "`-m A`" for *across* wires.

Comparing the two approaches, the *within-wires* strategy should be much simpler to implement. The *across-wires* strategy will require much more thinking and optimization on your part (perhaps four times more), so you will want to give yourself plenty of time to iterate on its design.

More Details on the Across-Wires Strategy

In contrast with *within-wires* approach (where the computation associated with exploring different routes is fairly uniform and only involves reading the shared occupancy matrix), the *across-wires* approach presents more signficant challenges: the amount of computation can vary dramatically across different wires (depending on how far apart their endpoints are), and it involves not only reading but also *updating* the shared occupancy matrix (when a new route is chosen). Hence you will want to think carefully about issues such as load balancing, locality, synchronization, and how to minimize overheads related to communication, contention, and extra work.

Synchronization. For the across-wires approach, you will need to be careful about how you update the occupancy matrix when wire routes are changed so that it does not become corrupted. While we are not concerned with reads of the occupancy matrix during intra-wire route exploration happening concurrently with wire route updates (i.e. it is not necessary to use a readers-writer lock), we are concerned with concurrent wire route updates that overlap in the occupancy matrix. You will want to think carefully about your synchronization strategy (and also perform experiments to make sure that it does not become a signficant bottleneck).

Batch Size Parameter for the Across-Wires Strategy.

For your implementation of *across-wires* parallelism, you should structure your code so that the parallel tasks are *batches* of wires, where the batch size is given as a runtime parameter (using "`-b batch_size`"). (Your code should default to a batch size of one if no runtime parameter is specified.) When a processor grabs a batch of wires, it will process each wire *within* that batch sequentially. (The parallelism arises because each separate processor is concurrently processing its own batch.) The purpose of the batch size

```
// Outer simulated annealing time steps
for (timestep = 0; timestep < N_iters; timestep++) {
   while (there is still work to do in this timestep) {
      grab a batch of B wires to be routed;
      for (i = 0; i < B; i++) {
         determine the new route for wire i (within the batch);
         // Note: remember the route, but do not update the
         // occupancy matrix yet
      }
      // Note: the loop below does not start until after the loop
      // above finishes
      for (i = 0; i < B; i++) {
         update the occupancy matrix for wire i (within the batch);
      }
   }
}
```

Figure 6: Pseudo-code for the outer control structure of the Across-Wires approach with batching (where B is the batch size).

parameter is to introduce some flexibility regarding how frequently updates to the occupancy matrix occur by performing wire route updates in batches as follows.

When a processor grabs work using the *across-wires* strategy, it grabs a batch of B wires at a time. It then explores the routes for those wires (sequentially within the batch) and chooses the best routes for all B wires, but it does not yet update the occupancy matrix. *After* the new routes have been chosen for all B wires, the processor then updates the occupancy matrix (to reflect the new wire routes) for all B wires as a group. (Note that when $B = 1$, the code will simply apply the changes for a given wire to the occupancy matrix immediately after it determines its new route.) In summary, a pseudo-code version the outer control flow in shown in Figure 6.

There are tradeoffs with increasing the batch size. On the one hand, it may potentially help performance, since updates to the occupancy matrix will occur less frequently. On the other hand, by postponing the updates for all of the wires within a batch (until after their routes have been chosen), there may be a negative impact on the overall quality of the routing result (i.e. the overall cost may increase). Hence we would like for you to experiment with this batch size parameter, to see how these tradeoffs affect both quality and performance.

General Advice

Ignore SIMD: Although OpenMP contains primitives for exploiting SIMD parallelism, the focus of this assignment is taking advantage of parallelism across multiple *processor cores*. Hence we recommend that you save time by ignoring SIMD parallelism in this assignment: there are plenty of other interesting things that will take up your time, and which are more relevant to optimizing speedup across multiple cores.

Start the Across-Wires Approach Early: There is far more subtlety (and far more opportunities for optimization) with the across-wires approach (compared with the within-wires approach), so it would be wise to start it early. As mentioned in the Piazza post entitled "Performance Debugging", when we do grading on this assignment, we are just as interested in the journey as the destination: we want to hear about your thought processes as you did performance debugging, including the various strategies that you tried, what you learned from measuring these earlier iterations, how that shed light on your next approach, etc. So be sure to give yourself enough time to explore these issues in some depth.

Task Assignment for the Across-Wires Approach: During lectures, we discussed a number of different strategies for doing task assignment in parallel programs, including static, semi-static, and dynamic assignment with different granularities. Remember that the goal of task assignment is to achieve good load balancing and good locality while being careful about run-time overheads. The across-wires approach should give you some interesting opportunities to think about these issues. The OpenMP loop pragmas support a number of different scheduling policies (http://jakascorner.com/blog/2016/06/omp-for-scheduling.html). However, keep in mind that these are not your only options: similar to what we saw in the Barnes-Hut galaxy simulation, you can write arbitrary code to partition work across processors (e.g., after loading the input file but before the main computation begins).

Synchronization for the Across-Wires Approach: As mentioned earlier, you will want to think carefully about your synchronization strategy for the across-wires approach. Keep in mind that as a programmer, you can adjust the granularity of locking on data structures. At the same time, be careful not to accidentally create deadlock situations. Finally, when a processor is updating an element of the occupancy matrix (inside an appropriate form of critical section), it is much safer to perform increment/decrement operations rather than simply overwriting it with a value (since other processors may have updated it recently).

Staleness of the Occupancy Matrix: For the across-wires approach, when the batching parameter (B in Figure 6) is set to be greater than one, it introduces some amount of staleness in the occupancy matrix, since later wires within a batch do not see the updates from earlier wires within the same batch (as they are exploring potential routes). Taken to the extreme, if we allow the occupancy matrix to become arbitrarily stale, it may be very good for performance, but bad for the quality of the result. In this assignment, you may not introduce any additional staleness of the occupancy matrix beyond what we have specified (i.e. with the batching parameter, plus the fact that readers-writer locks are unnecessary).

Output from your program.

The output from your program should include (i) the contents of the *occupancy matrix* and a (ii) representation of the actual wire routes for each wire, which should be stored under the following filenames, respectively:

- `occupancy_$inputFileName_$numThreads.txt`
- `routes_$inputFileName_$numThreads.txt`

The content format for the occupancy matrix output file should be a space-delimited matrix of numbers:

```
<maxX> <maxY>
<OM[0][0]> <OM[0][1]> <OM[0][2]> ...
<OM[1][0]> <OM[1][1]> <OM[1][2]> ...
<OM[2][0]> <OM[2][1]> <OM[2][2]> ...
...
```

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 work completes, and it should not be counted against your parallel speedup.

In your writeup, please present the routes and the occupancy matrix 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 occupancy matrix, as well as the overall cost (the sum of the squares over all matrix elements).

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:

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

Command Line Flags

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

- `-f <filename>`: Specifies the name of the input file that specifies the wire routing problem (using the format in Figure 1).
- `-n <numThreads>`: Specifies the number of threads. Note that another way to specify the number of threads is by setting the `OMP_NUM_THREADS` environment variable (which can be queried from within a parallel region by using `omp_get_num_threads()`). (Further details can be found in the OpenMP.org documentation on setting the number of threads.) If you find it useful to explicitly specify the number of threads in the command line, use this flag.
- `-p <probability>`: Floating-point value between 0.0 and 1.0 that is used by the simulated annealing algorithm as the probability (P) of choosing a random path (as opposed to the least expensive path). The default value should be 0.1.

`-i <numIters>`: Integer value (> 1) that is used by the simulated annealing algorithm as the number of optimization iterations (N_{iters}). The default value should be 5.

`-m <parallelMode>`: Specifies whether the parallelization strategy is either *within* wires ("`-m W`") or *across* wires ("`-m A`").

`-b <batchSize>`: Specifies the batch size for the *across-wires* strategy. The default value should be 1.

Measuring Performance

While it may be helpful to compile using the `-g` flag when you are debugging your code, **please be sure to use the `-O` flag to generate any programs that you will be timing!** There can be significant differences in performance between `-g` and `-O`, and we are only interested in speeding up optimized code.

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 $\frac{T_1}{T_p}$, where T_1 is the time for one processor, and T_p 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
```

Also you can look at this tutorial for `perf`.

Logistics

Input files: 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`.

Using the GHC and PSC machines. For debugging, testing, and the bulk of your performance evaluation, you will be using the GHC machines. Host names for these machines are `ghc`X`.ghc.andrew.cmu.edu`, where X is between 26 and 86. You will also be running a limited set of experiments on the PSC Bridges-2 Regular Memory (RM) machines. Here is a tutorial on accessing the PSC machines. You may also find it helpful to refer to the Bridges-2 User Guide.

IMPORTANT: We have a limited allocation of time on the PSC machines, and that time is especially important for Assignment 4 and the class projects. Therefore please follow these rules regarding using the PSC machines:

- Do not perform any debugging (or signficant editing) on the PSC machines. We are charged the same for debugging time as we are for collecting performance data, and this time is precious. You should debug on GHC, and only transfer code to the PSC machines when it is stable.
- Collect a full set of performance results on the GHC machines before you start running experiments on the PSC machines. Once you are happy with your performance debugging on GHC, then start doing your PSC experiments.
- Submit no more than 10 sets of experiments per day on the PSC machines. There is a signficant lag between when machine time is used up and when the instructors are notified about it. In previous years, we have had problems where a few groups performed large numbers of experiments, using up the bulk of the time allocation for the class (resulting in their PSC accounts being suspended). We are only asking you to perform a limited set of experiments on the PSC machines for Assignment 3, so please be careful to stay within your 10-experiments-per-day budget. Thanks!

Performance and Cost Targets

When we evaluate the performance of your code, we care about three metrics: (i) scalablity (i.e. how much speedup you achieve as you scale up to more processors), (ii) absolute performance, and (iii) the overall cost metric for your wire routing result.

We do not provide a single target metric because there are ways to artificially boost some of these three metrics by penalizing the others. For example, if the processors did not communicate with each other at all during wiring, they might achieve excellent speedup at the expense of extremely poor wiring cost. As other examples, it is sometimes possible to achieve excellent speedup by artificially slowing down single-thread performance, or to hit an absolute performance target on 8 threads by focusing only on boosting single-thread performance on non-scalable code. We definitely do not want you to play games like this.

Here is the way to think about what we are looking for: assuming that your code has reasonably good single-thread absolute performance and produces a wiring result with reasonably good cost, we are especially interested in the *scalability* of your code (i.e. how well it speeds up as you increase the number of processors). You should be able to get reasonably good speedups using both the *within-wires* and *across-wires* approaches (although performance may vary more in the latter case).

To give you a more concrete sense of what we are looking for, we will be posting more specific guidance in this performance targets file by 5pm on Friday, February 14th. (We will also post on Piazza when it is ready.) When we assign your overall score on the assignment, 30% of the score will be based upon the performance of your code, and 70% will be based upon the quality of your discussion and analysis in your writeup. Our performance guidance will include two target numbers: one to achieve 50% of the performance points, and one to achieve 80% of the performance points. To achieve credit beyond 80% on the performance score, we will be looking at the distribution of the performance of the top performers in the class. (Note that once you achieve the 80% target, the course staff will not be providing advice on how to optimize your code further: that is up to your group to explore on your own.)

Your Mission

In this assignment, you will be implementing, evaluating, and iteratively improving both the *within-wires* and *across-wires* strategies for leveraging parallelism across mutiple processors using the OpenMP programming model. We recommend that you implement these two strategies through separate procedures in your `wireroute.cpp` source file that are invoked based upon the parallelization mode command line flag (i.e. ("`-m W`") or ("`-m A`").

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.

In your write-up, please include the following items:

1. **[20 points] Design and performance debugging journey for your Within-Wires approach:** Provide a detailed discussion of the thought process that went into designing your program, and how it evolved over time based upon your experiments. (See the Piazza post entitled "Performance Debugging"). Specifically, try to address the following questions:

    - What approaches did you take to parallelize the algorithm (including not only your final design, but any other designs along the way)?
    - Include your reasoning for your final implementation choices, including any graphs or tables that helped you make your decisions.
    - 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. **[40 points] Design and performance debugging journey for your Across-Wires approach:** Now repeat this same analysis (answering the same questions from part 1 above) for your *across-wires* approach.

3. **[2 points] Routing output:** Show (graphically) the routing outputs for both parallel versions of your program for the `medium_4096.txt` input circuit, running on 8 processors on the GHC cluster.. (It is okay to include screen shots of the output from the `WireGrapher.java` program that we provide.)

4. **[20 points] Experimental results from the GHC machines:** For each of these experiments, please collect and present data for 1, 2, 4, and 8 threads for each experiment while running on the GHC cluster (`ghc`X, where X is between 26 and 86). Perform the analysis below for **both** of your parallelization strategies: *within-wires* and *across-wires*.

   (a) **Speedup graphs:** Show a plot of the *Total Speedup* and *Computation Speedup* vs. *Number of Processors* (N_{procs}). Discuss these results, including any non-ideal behaviors.

   (b) **Cache misses:**

      i. **Total cache misses:** Show a plot of the total number of *cache misses* for the entire program vs. *Number of Processors* (N_{procs}).

      ii. **Per-thread cache misses:** Show plot of the arithmetic mean of *per-thread* cache misses (from perf stat -e cache-misses $PROGRAM) vs. *Number of Processors* (N_{procs}).

      iii. **Discussion:** Discuss the trends that you see in these cache miss plots, including any surprises, and how these numbers relate to your speedup graphs.

5. **[8 points] Sensitivity studies on the GHC machines:** For these experiments on the GHC machines, collect numbers for just your *across-wires* approach, using just 1 and 8 threads.

   (a) **Sensitivity to the probability of choosing a random route:** Show a plot of the *Computation Speedup* on 8 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 2 threads.) Discuss the impact of varying P on performance, explaining any effects that you see.

   (b) **Sensitivity to the problem size:** Show a plot of the *Computation Speedup* on 8 threads with respect to 1 thread where the input problem size is varied using the different input files in `/code/inputs/problemsize` directory. In particular, we are focusing on differences in grid sizes and numbers of wires. Please discuss the impact of varying the problem size on performance, explaining any effects that you see.

6. **[10 points] Experimental results from the PSC machines:** For *both* of your parallelization strategies (i.e. *within-wires* and *across-wires*), measure the performance of your code on the PSC Bridges-2 Regular Memory machines.

   (a) **Speedup graphs:** Show a plot of the *Total Speedup* and *Computation Speedup* vs. *Number of Processors* (N_{procs}). Discuss these results, including any non-ideal behaviors.

   (b) **Comparison with GHC results:** For both of your parallel strategies, discuss how these results compare with the speedup curves that you measured on the GHC machines.

1 Hand-in Instructions

You will submit your code via Autolab and your report via Gradescope. For the code, you will be submitting all C++ header and source files in the `src` folder.

1. Submitting your code:
   (a) If you are working with a partner, form a group on Autolab. Do this before submitting your assignment. One submission per group is sufficient.
   (b) Make sure all of your code is compilable and runnable. We should be able to simply run make. Please remove excessive print statements, if they were added.
   (c) Run the command "make handin.tar." This will run "make clean" and then create an archive of any C++ source code in /src. If you find it absolutely necessary, you may modify the Makefile to include other files you have added.
   (d) Submit the file handin.tar to Autolab.
2. Submitting your writeup:
   (a) Please upload your report as file report.pdf 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.