15-418/15-618 Parallel Computer Architecture and Programming

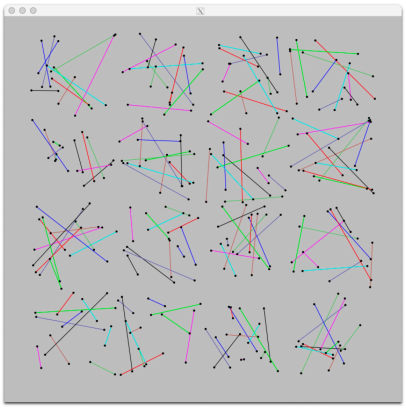
Homework 3, Parallel VLSI Wire Routing via OpenMP

Siqi Guo(AndrewID: siqiguo) January 21, 2025

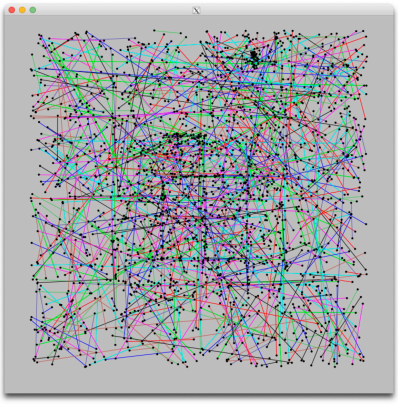
Use OpenMP to write parallel code using the shared address space model.

- Instrument the code to determine where the most time is spent in computation.
- Evaluate where optimizations are most valuable.
- Focus on avoiding sequential bottlenecks, memory contention, and workload imbalance.

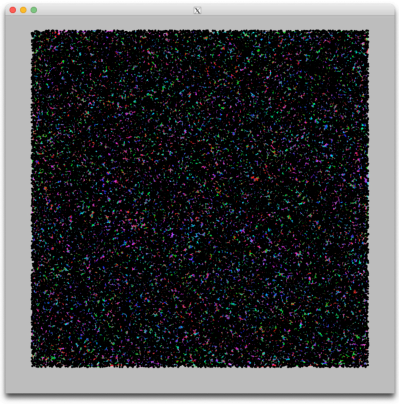
First, let's visualize the VLSI wire routing.



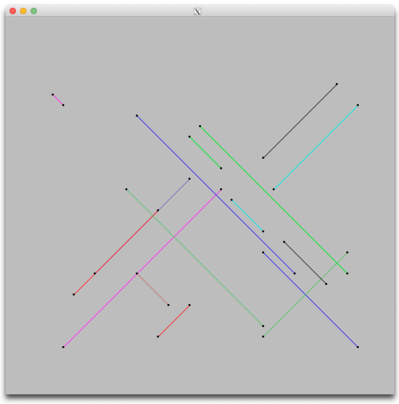
(a) `java WireGrapher ./inputs/timeinput/easy_4096.txt`



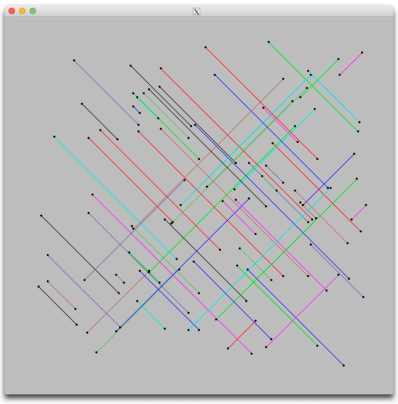
(b) `java WireGrapher ./inputs/timeinput/hard_4096.txt`



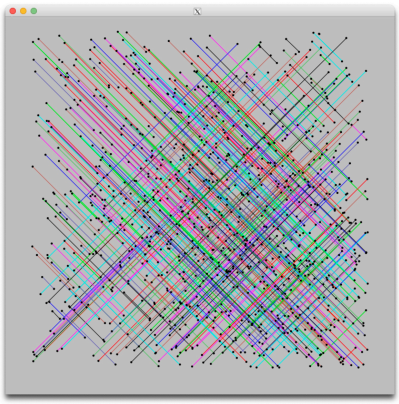
(c) `java WireGrapher ./inputs/timeinput/extreme_4096.txt`



(d) `java WireGrapher ./inputs/testinput/circuit_32x32_16.txt`



(e) `java WireGrapher ./inputs/testinput/circuit_256x256_64.txt`



(f) `java WireGrapher inputs/testinput/circuit_1024x1024_512.txt`

Note:
In this assignment, I ignore Simulated Annealing to simplify the problem. I did not benchmark the cache performance as `perf` not installed on my Andrew machine.
Besides, I do not have the access to the PSC machines, so I also skip that section.

1. Design and performance debugging for my Within-Wires approach (20 pts)

i. This problem requires us to minimize the overall total cost (the sum of the squares over all matrix elements), while speedup the programs, taking advantage of parallelism across multiple processor cores.

ii. The implementation of VLSI wire routing problem solver.

```
int main(int argc, char *argv[]) {
    // variable declarations...
    parse_arguments(argc, argv, input_filename, num_threads, SA_prob, SA_iters, parallel_mode,
        batch_size);

    Logger logger;
    WireRouter router(parallel_mode);

    /* Initialize any additional data structures needed in the algorithm */
    initialize_routing(input_filename, dim_x, dim_y, num_wires, wires, occupancy);
    logger.log_time_message("Initialization");

    // Don't use global variables. Use OpenMP to parallelize the algorithm.
    router.run(wires, occupancy, SA_iters, SA_prob, num_threads);
    logger.log_time_message("Computation");

    wr_checker Checker(wires, occupancy);
    Checker.validate();

    logger.print_occupancy_stats(occupancy);
    logger.persist_wires_occupancy(wires, num_wires, occupancy, dim_x, dim_y, num_threads,
        input_filename);
}
```

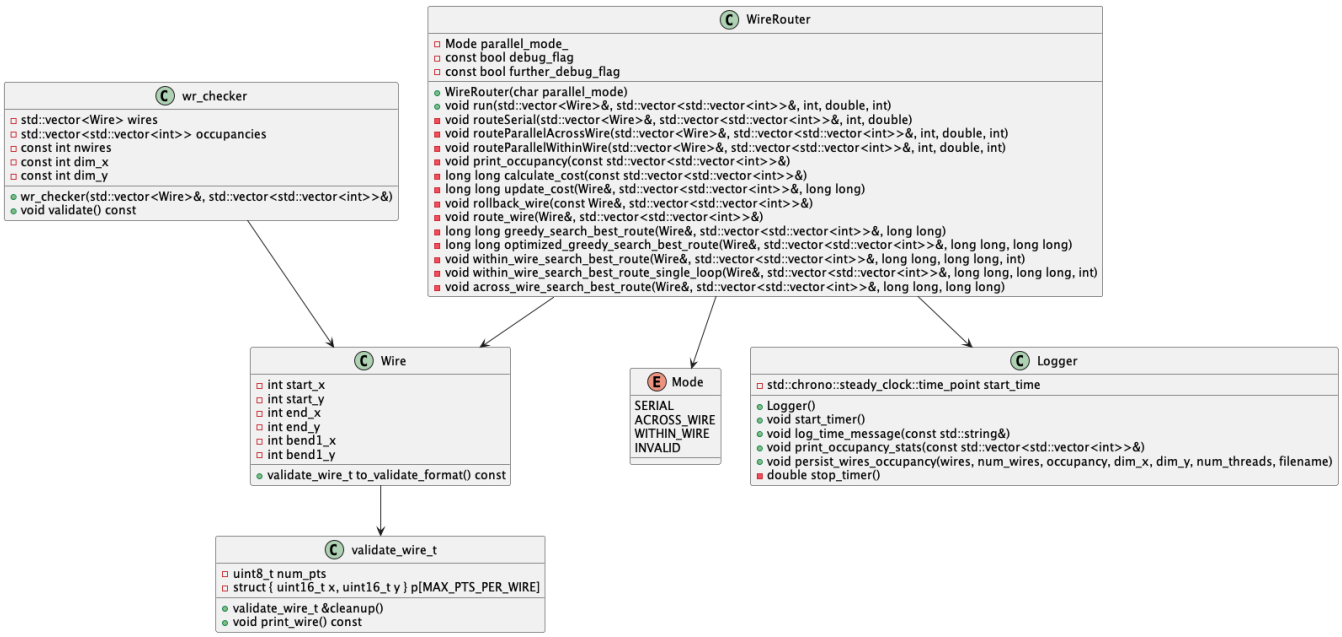


Figure 2: Structure of the Wire Routing Solver.

iii. I implement the serial version of the algorithm first, and then parallelize it using OpenMP.

```
function routeSerial(wires, occupancy, SA_iters, SA_prob):
    for iteration in range(0, SA_iters):
        for wire in wires:
            // Wire endpoints are on a straight line, skip bend calculations
            if wire.start_x == wire.end_x or wire.start_y == wire.end_y:
                continue
            end if
            rollback_wire(wire, occupancy) // Remove wire from occupancy grid
            new_wire = copy(wire)
            greedy_search_best_route(new_wire, occupancy, minimum_cost)
            route_wire(new_wire, occupancy) // Place the new wire on the grid
            wire = new_wire
        end for
    end for
```

At first, I did rollback and re-routed the wires, and re-calculated the cost from the occupancy matrix while doing the greedy search, which is inefficient. I realized that `rollback_wire()` and `route_wire()` are $O(\Delta x + \Delta y)$, but the calculate cost is $O(\Delta m \times \Delta n)$ (running in greedy searching along horizontal and vertical directions)., thus further optimizing the algorithm by alternating the cost calculation with online calculation with reading the occupancy matrix during the search.

What approaches did you take to parallelize the algorithm?

Then, after the serial version, I parallelize the algorithm with the within wire approach. Within wire approach divided the work of routing one wire into multiple threads, where each thread is responsible for routing one path ($\Delta x + \Delta y$ in total) for this wire.

Include your reasoning for your final implementation choices, including any graphs or tables that helped you make your decisions.

To increase the parallelism and make the threads as busy as possible, I use dynamic scheduling. Then, I either merge the greedy searching logic into one loop or use "omp for" to divide the work into several parts to balance the workload.
I tried omp sections as well, but it turns out to be inefficient when I only have two searching directions (horizontal and vertical).

Where is the synchronization in your solution? Did you do anything to limit the overhead of synchronization?

Use `#pragma omp critical` to split the critical sections and synchronize the threads.
Limit the private variables to reduce the overhead.

Why do you think your code is unable to achieve perfect speedup? (Is it workload imbalance? communication/synchronization? data movement?)

There is necessary communication and synchronization cost. For the private variables of each thread, they need to synchronize when they select the best wire.

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?

No (within 8 threads). The thread number is not the bottleneck and the overhead of more threads can be reduced by the shared memory and paid off by the speedup of the algorithm. Besides, I use dynamic scheduling to keep all threads busy and fully utilize the multi-thread parallelism.

2. Design and performance debugging for my Across-Wires approach (40 pts)

For the Across-Wires approach, I divided the work of all wires to different threads at the same time. Each thread will be assigned to route a wire, and each thread will do their own greedy search for the best path of this wire.

They have the shared occupancy matrix, and they need to synchronize when they try to update the matrix (`route_wire` and `rollback_wire`).

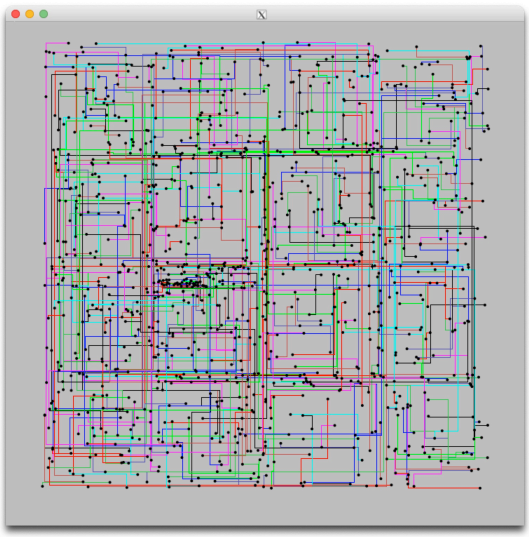
```
function routeParallelAcrossWire(wires, occupancy, SA_iters, SA_prob, num_threads):
    omp_set_num_threads(num_threads);
    for iteration in range(0, SA_iters):

    #pragma omp parallel for default(shared) schedule(dynamic)
        for wire in wires:
            // Wire endpoints are on a straight line, skip bend calculations
            if wire.start_x == wire.end_x or wire.start_y == wire.end_y:
                continue
            end if

            #pragma omp critical
            rollback_wire(wire, occupancy) // Remove wire from occupancy grid

            new_wire = copy(wire)
            across_wire_search_best_route(new_wire, occupancy, minimum_cost)

            #pragma omp critical
            route_wire(new_wire, occupancy) // Place the new wire on the grid

            wire = new_wire
        end for
    end for
```

To be honest, I think the synchronization here is not fully correct and safe, as read and write the matrix from different threads might mess up. However, it passed the correctness test, and here is an explanation from piazza Fall 2024 @215.

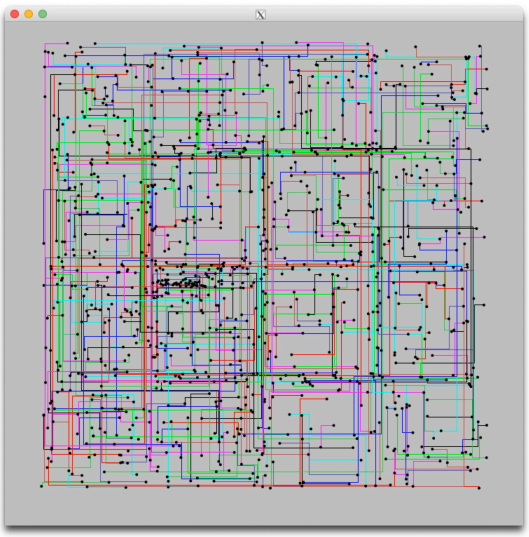
> "I had the same question during my implementation but after actually trying to use synchronization, I noticed that the cost without synchronization is actually lower than with synchronization, and faster too. I think the reason is that yes, reading when others are not finished updating will potentially mess up the actual cost of the path, but this cost is accounting for a partial wire. If the other workers completes that wire, then this routing might be better because it tries to avoid at least part of the other wire. Also, each iteration doesn't find the optimal path but just an approximation. It's likely that even if some partial wires mess up the cost calculation, the result is still going to be a good enough path. Hope this helps."

3. Routing output (2 pts)

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.



(a) Acorss-Wire `java WireGrapher ./outputs/timeinput/medium_4096_wires_8.txt`



(b) Within-Wire `java WireGrapher ./outputs/timeinput/medium_4096_wires_8.txt`

4. Experimental results from the GHC machines (20 pts)

i. For each of these experiments, please collect and present data for 1, 2, 4, 8, ~~and 16~~ threads for each experiment while running on the GHC cluster (ghcX, where X is between 26 and 86). Perform the analysis below for both of your parallelization strategies: **within-wires and across-wires**

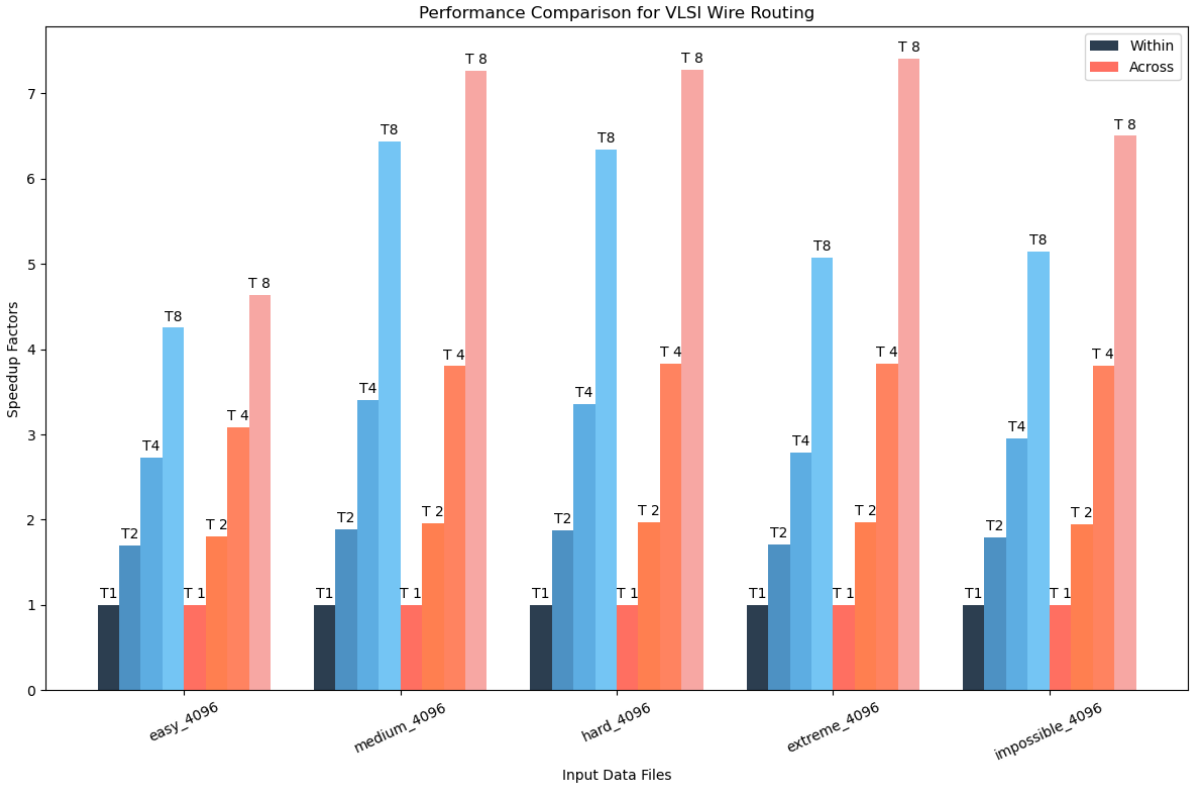
ii. Speedup graphs: Show a plot of the ~~Total Speedup and~~ Computation Speedup vs. Number of Processors (Nprocs).

iii. ~~Cache misses.~~ (skipped)

Table 1: Performance Comparison (Time in Seconds & Computation Speedup Factors) for VLSI Wire Routing

Data/TXT File	Method	Single Thread	2 Threads	4 Threads	8 Threads
easy_4096	within	0.7519 (1.00x)	0.4412 (1.7x)	0.2757 (2.73x)	0.1769 (4.25x)
	across	0.7465 (1.00x)	0.4153 (1.8x)	0.2424 (3.08x)	0.1610 (4.64x)
medium_4096	within	11.7119 (1.00x)	6.2040 (1.89x)	3.4459 (3.4x)	1.8175 (6.44x)
	across	11.5672 (1.00x)	5.9062 (1.96x)	3.0439 (3.8x)	1.5926 (7.26x)
hard_4096	within	14.8109 (1.00x)	7.9029 (1.87x)	4.4132 (3.36x)	2.335 (6.34x)
	across	14.7000 (1.00x)	7.4589 (1.97x)	3.8410 (3.83x)	2.0189 (7.28x)
extreme_4096	within	66.0754 (1.00x)	38.6065 (1.71x)	23.7005 (2.79x)	13.0250 (5.07x)
	across	65.5949 (1.00x)	33.2610 (1.97x)	17.1295 (3.83x)	8.8578 (7.41x)
impossible_4096	within	509.445 (1.00x)	285.091 (1.94x)	172.879 (3.45x)	99.168 (7.01x)
	across	514.908 (1.00x)	264.419 (1.95x)	135.083 (3.40x)	79.163 (6.76x)

iv. Computation Speedup Comparison Graph.



5. **Sensitivity studies on the GHC machines (8 pts)**

   i. For these experiments on the GHC machines, collect numbers for just your **across-wires** approach, using just 1 and 8 threads.

   ii. ~~Sensitivity to the probability of choosing a random route~~ (skipped)

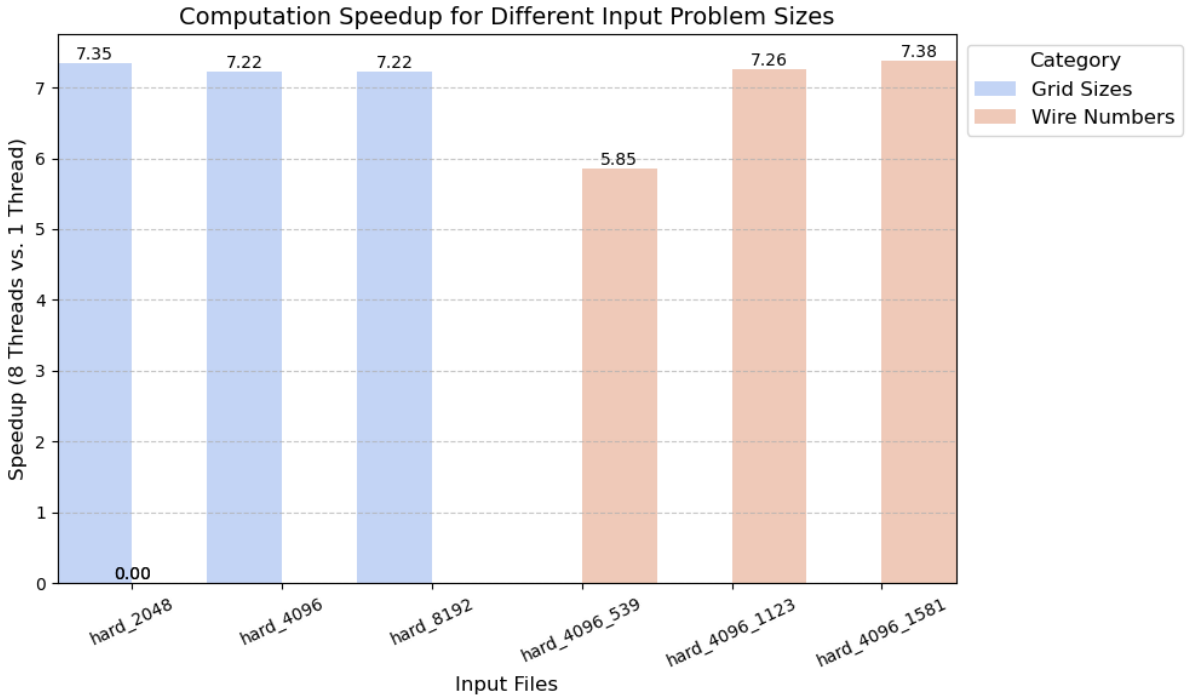
   iii. Sensitivity to the problem size: Show a plot of the Computation Speedup on 8 threads with respect to 1 thread where the input problem size is varied using the different input files in /code/inputs/problemsize directory. In particular, we are focusing on differences in grid sizes and numbers of wires.

Table 2: Sensitivity to the Problem Size based on Across-Wire 8-Threads Computation Speedup <I>

Grid Sizes	**hard_2048**	**hard_4096**	**hard_8192**
Speedup Factors	2.8548 (1.00x) 0.3883 (7.35x)	14.725 (1.00x) 2.0399 (7.22x)	114.982 (1.00x) 15.9210 (7.22x)

Table 3: Sensitivity to the Problem Size based on Across-Wire 8-Threads Computation Speedup <II>

Numbers of Wires	**hard_4096_539**	**hard_4096_1123**	**hard_4096_1581**
Speedup Factors	4.9027 (1.00x) 0.8387 (5.85x)	14.733 (1.00x) 2.0297 (7.26x)	31.790 (1.00x) 4.3061 (7.38x)



6. ~~**Experimental results from the PSC machines (10 pts)**~~