15-618 Assignment 3

1

My code and optimizations follow the rule:

all the wires must rely on some other wires (just like assignment 2).

Version 1: using `omp parallel for`

- Since the wires rely on each other, I cannot route the wires in parallel. Then for each wire, I tried to explore all the possible paths in parallel.
- There has to be a synchronization (critical area) before the end of the for-loop to update the best route for this wire. And there is another synchronization at the end of the for-loop. They take a lot of time.
- I also ovserved the process monitor using `htop`. Some cores are busy and some are not, which means load-imbalance.

Version 2: load-balance improvement: `schedule(guided)`

- The load-balance problem is because of some wires have more possibility than others. The wires are routed in parallel and some finishes earlier than others. To solve this problem, I added `schedule(guided)` after `omp parallel for`.
- When using 64 CPU cores, CPU usage is from about 57% to 63%.
- This improvement helped increasing the performance by 20%.

Version 3: limit the synchronization overhead in `omp critical` : user-defined `reduction`

- All the possible routes threads can run toghether but only one can update the `best_route` variable. I tried to make this update run in parallel by using `omp parallel for reduction`
- To make this happen, I implemented a user-defined reduction function using `omp declare reduction`. According to the OpenMP document, it can do the reduction in parallel.
- This improvement helped increasing the performance by 15%.

Version 4: try nested `omp for`

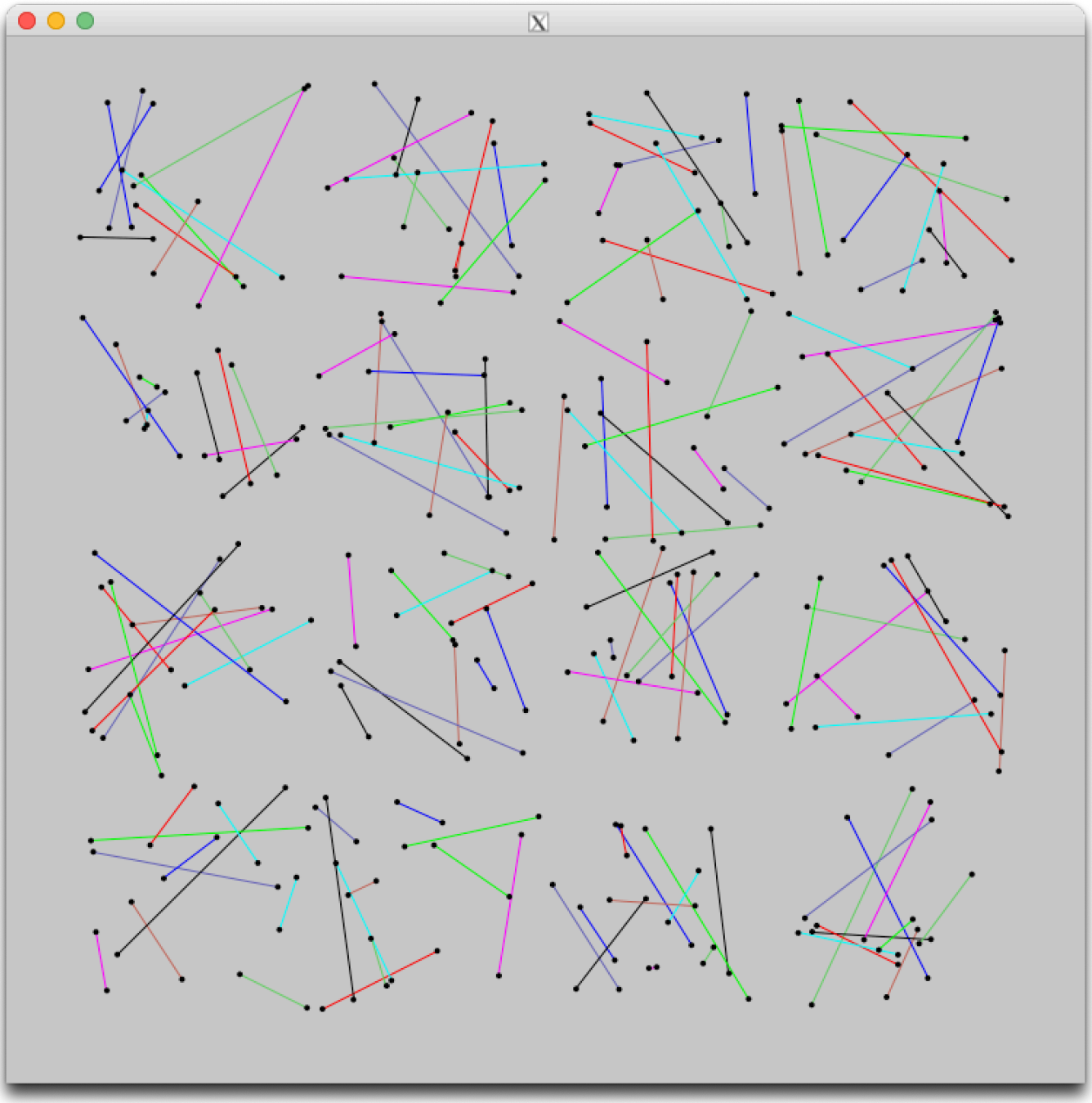
- Syntax error: too close for-loop
- Two for-loops need to be flattened to one for-loop. But as the wires are not independent, this cannot be done.

Version 5: try `omp sections` to calculate three routes tothegher

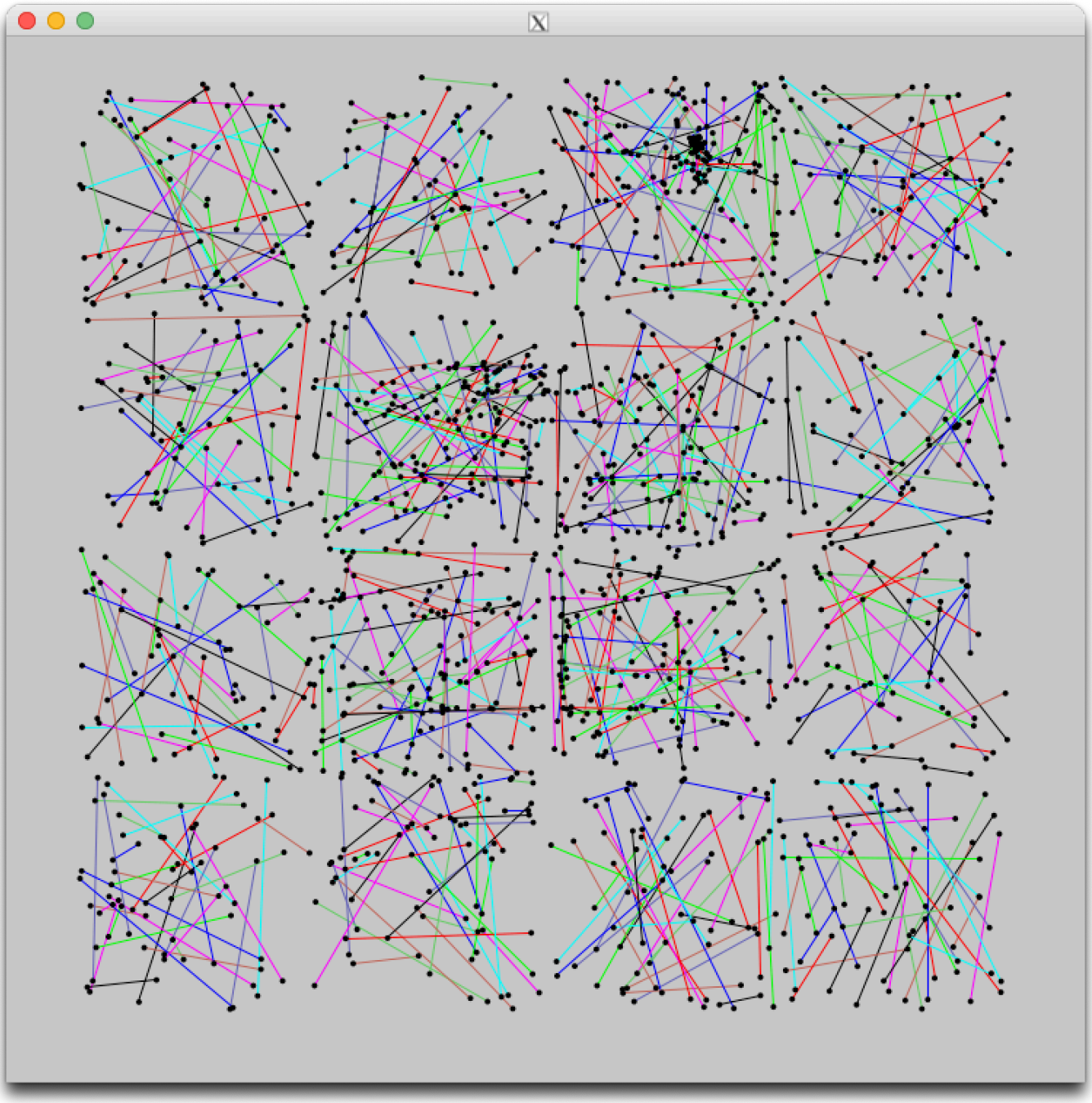
- Tried to calculate three lines in one path toghether. But only one thread is working when using `sections`. Maybe it is because the path calculation is already in one thread.
- Performance go down.

Version 6: limit the synchronization overhead: grouping the wires

- I tried to visualize the input data.
  - `inputs/timeinput/easy_4096.txt` can be divided into 4 cells x 4 cells. (1024x1024 cell)



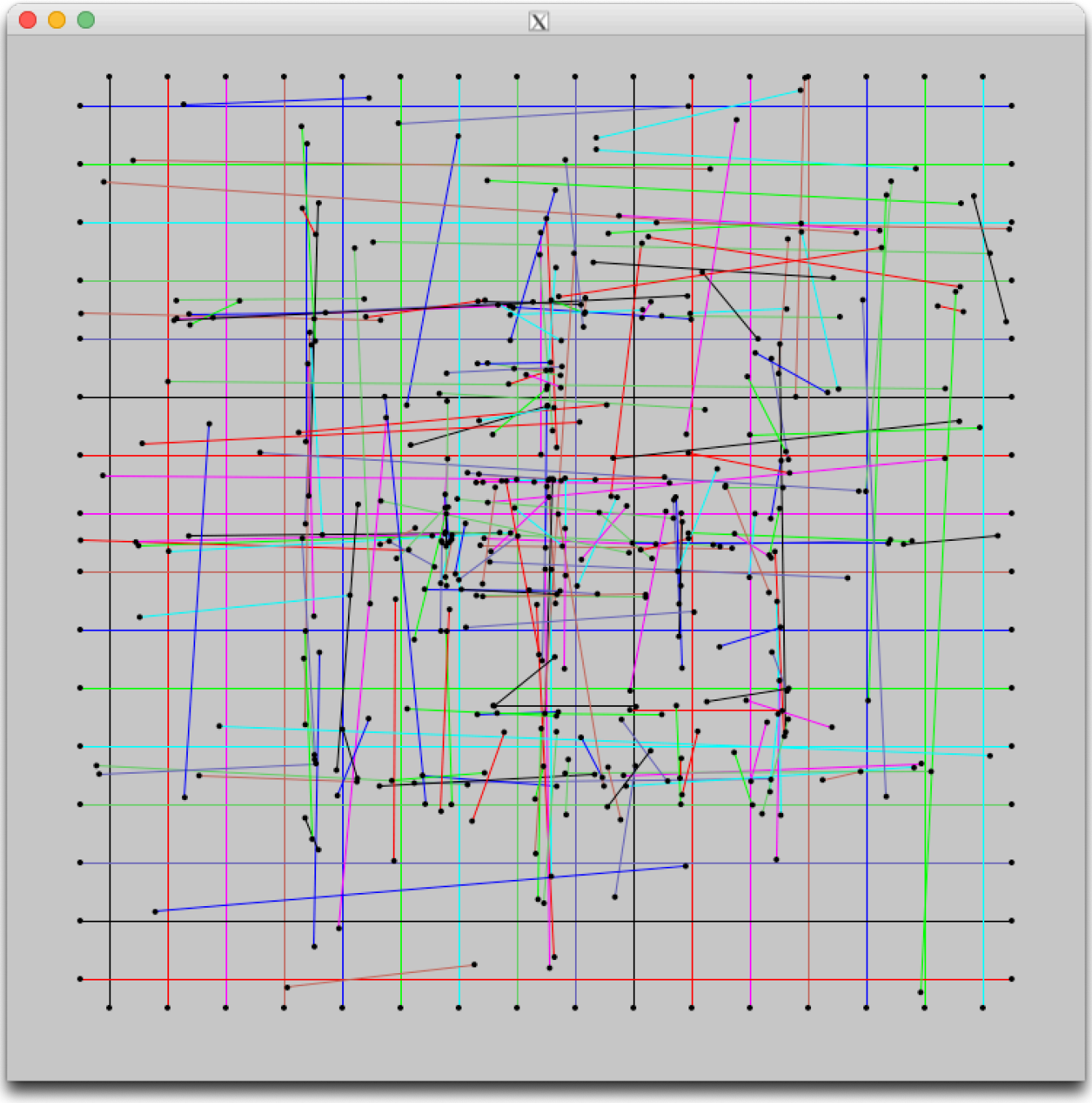
- I wrote another script to analyse the input data. 68% of the wires in `medium_4096` and `hard_4096` are in one of the cells.



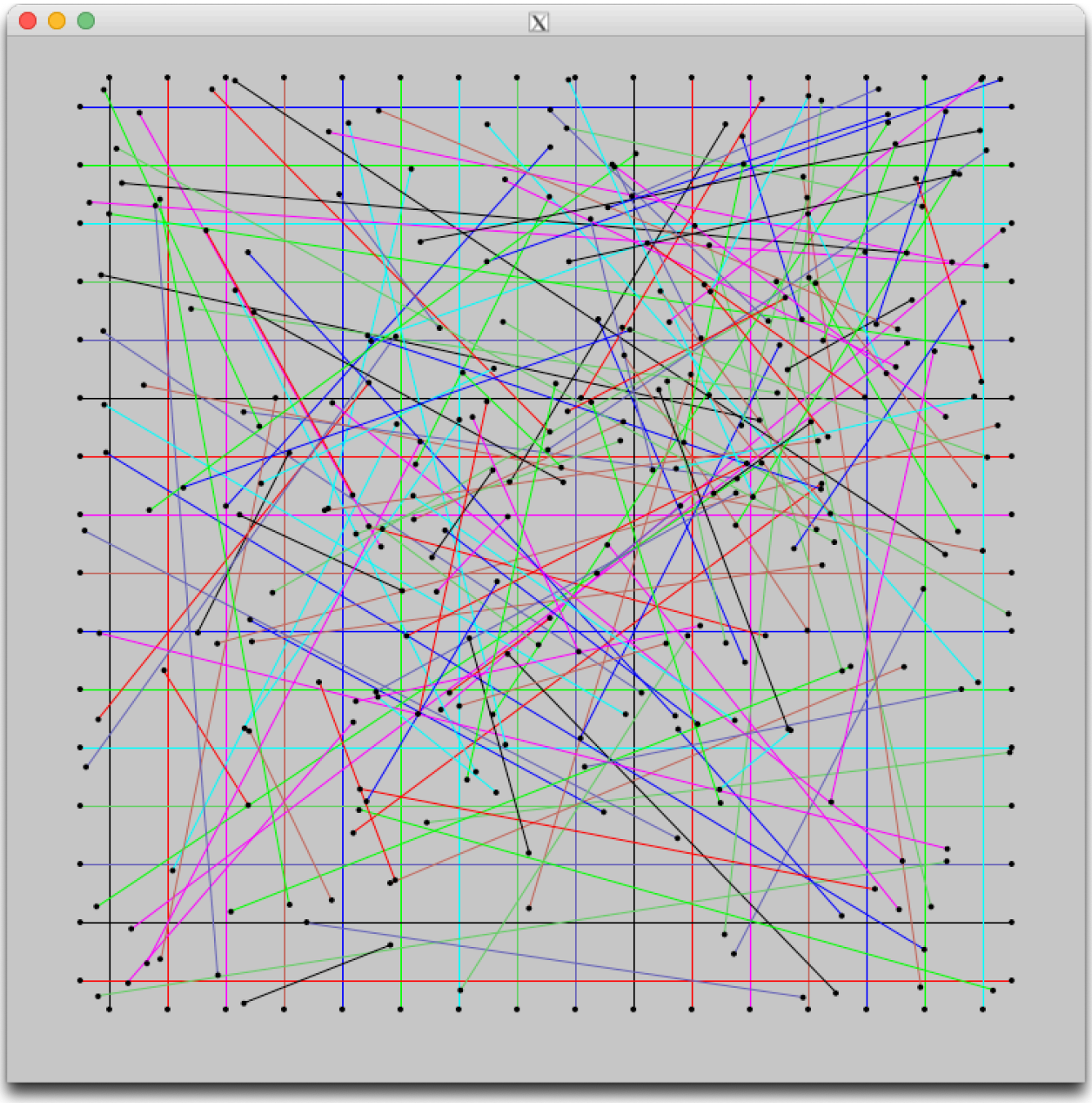
- So I split the input data into several parts. Solve the individual cells before other data.
- I tried to use one thread for each cell. No more synchronization in each thread this time.
- This method can solve **68% data** for medium_4096 and hard_4096 **within 1s**.
- This method can solve the easy_4096 data using 1,4,16,64 threads **in 2.0s, 0.56s, 0.21s, 0.22s** (reached the reference solution on PSC)
- The huge performance increasement comes from increasing `inst per cycles` **from 0.6 to 1.2**. I think IPC increasement is because that no more barrier after the for-loop.

Version 7: try to group more data

- Wires distribution in medium_4096 and hard_4096 are similar.
- I analyzed the wires that are not in one cell. As we can see, about 15% of all wires are in one 256x4096 cell.



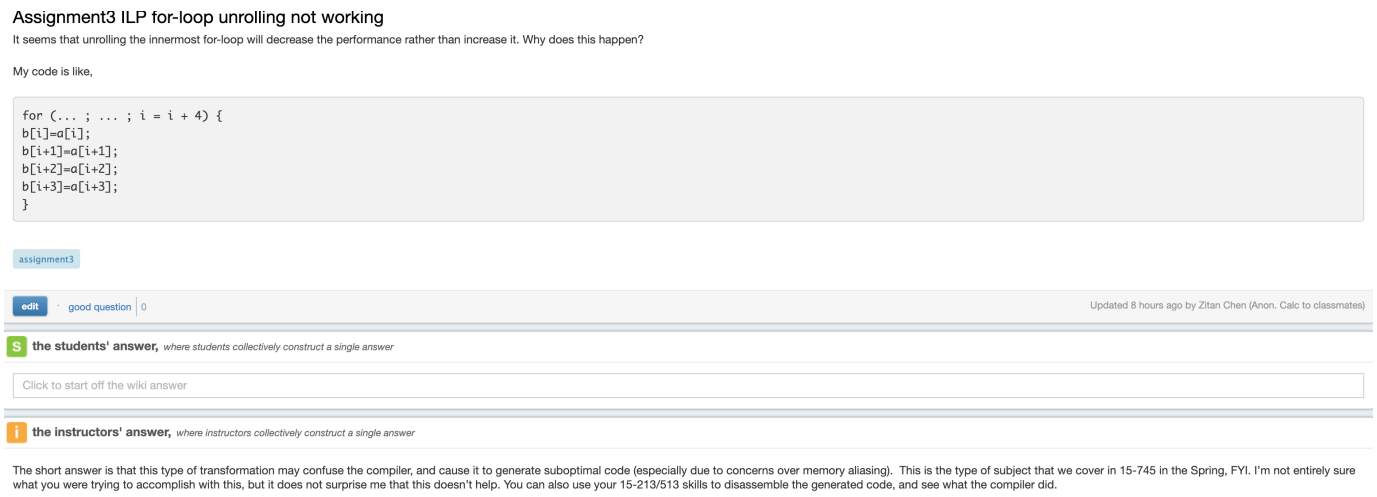
    - I tried to solve this wires seperately using 16 threads. The performance remain the same. Most of the wires are in the same several 256x4096 cells, which leads to load-balance problem.
- Other 17% wires cannot be grouped. I also tried to visualize them.



- I also tried additional 8 cells x 8 cells for 64 threads, 2 cells x 2 cells for 4 threads, 2 cells x 4 cells for 8 threads. No performance increase.

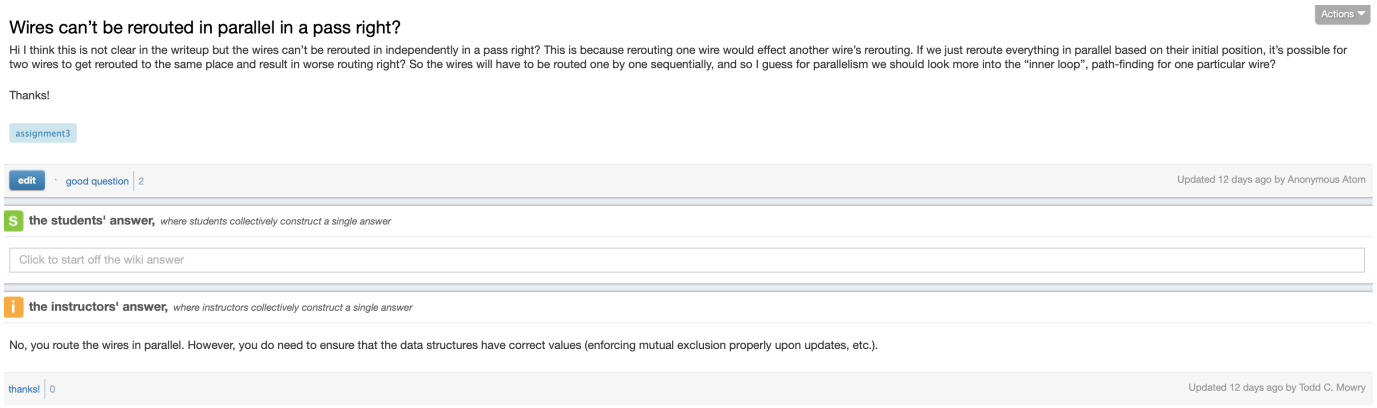
Version 8: try ILP: unrolling the for-loop

- `perf` told me `instruction per cycle` is about 0.4 to 0.8 for the non-in-cell routes. I think I can try to unroll some for-loop.
- I tried unroll step 2,3,4. But the performance will decrease badly(about 2x time).
- `perf record` said 74% of the time are used to walk one line after another line(my hotspot function). So it deserves some optimization.
- However, the compiler may be confused.



Version 9: route the wires in parallel (not finished)

- Since I just knew that all the wires can be routed toghether without much influence on correctness, I don't have enought time to refactor my code.
- All my codes are built on the idea that wires rely on each other. Refactoring all the codes need more time.
- Prof Todd said "No, you route the wires in parallel." Then I grouped most of the wires to make their routing work in parallel. But I did not realize that, in fact, we **can** route **all** the wires in parallel.



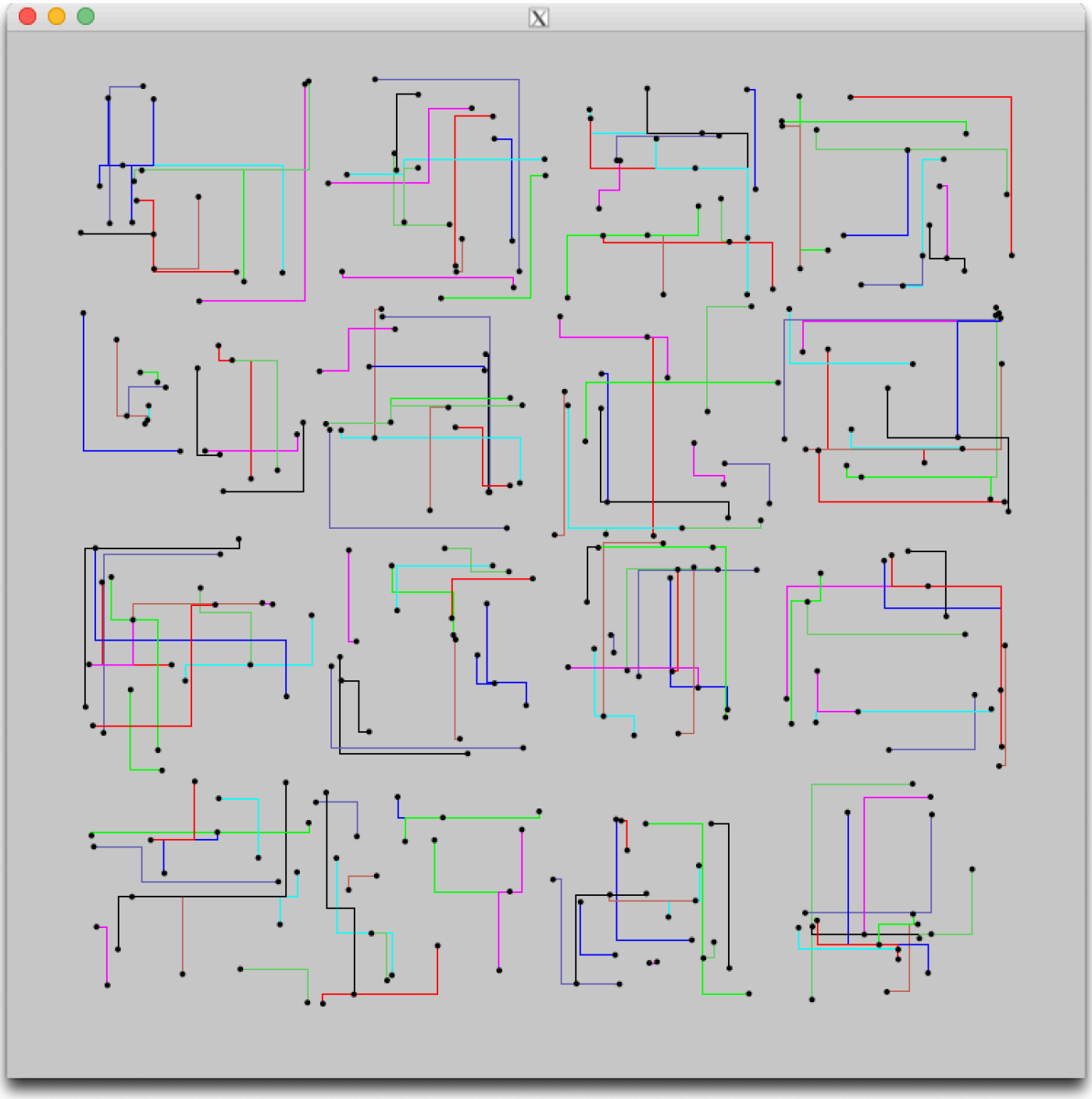
Other optimizations and experiences

- Data movement:
  - `memcpy` **large arrays is expensive.** Copying the `costs` matrix each time costs **0.1s**.
  - Then I tried to uses references of arrays at all time. However, I found that small and frequently-used arrays should be copied to lcoal variables in a function. This will slightly increase the performance. I think this increases spatial locality.
  - I also tried to optimize the hotspot function by changing local variables from `int i` to `register int i`. No performance increase. I think the complier have place these variables in registers already. Explicit declaration of using `register` won't help in these cases.
- **Synchronization**
  - **Synchronization is the main reason that affects my speedup. We need to avoid synchronization as much as possible.**

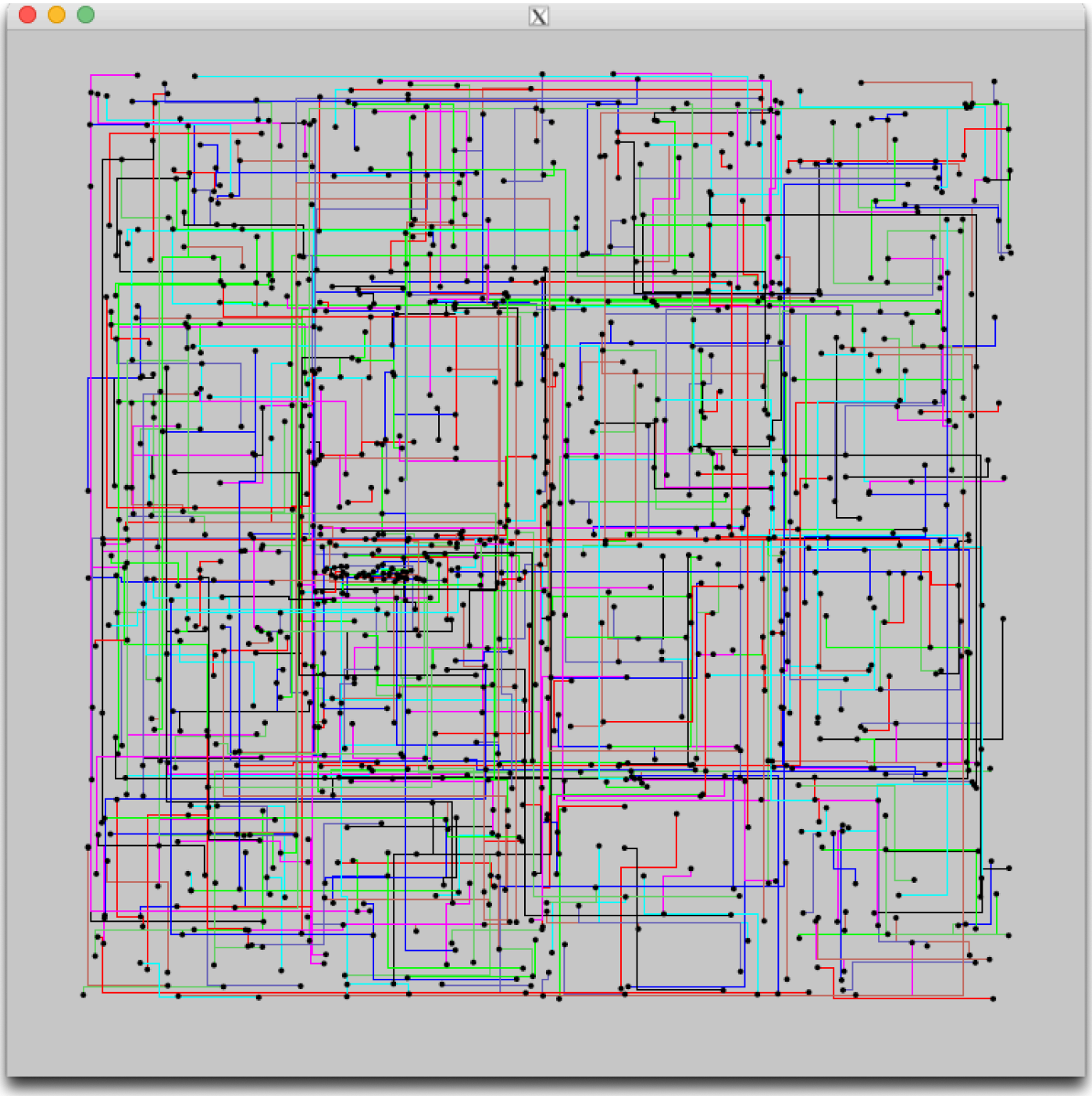
    - My code is unable to achieve perfect speedup because of synchronization. As I believed that the wires rely on each other, this problem cannot be solved. However, I tried to **remove the synchronization barriers(ignore the correctness of results). There will be a >3x speedup.**
- **High thread counts drop-off**
    - Yes. My code is slightly slower when using 128 threads than 64 threads.
    - I used `perf record` to find out the reason. The main reason is the overhead from `OpenMP`. `OpenMp` needs some time to spawn lots of threads. Synchronization is another main reason.
- Workload-imbalance
    - Solved by using `omp for schedule(guided)`
    - This will greatly improve load-balancing by asking idle threads to help busy threads.

2

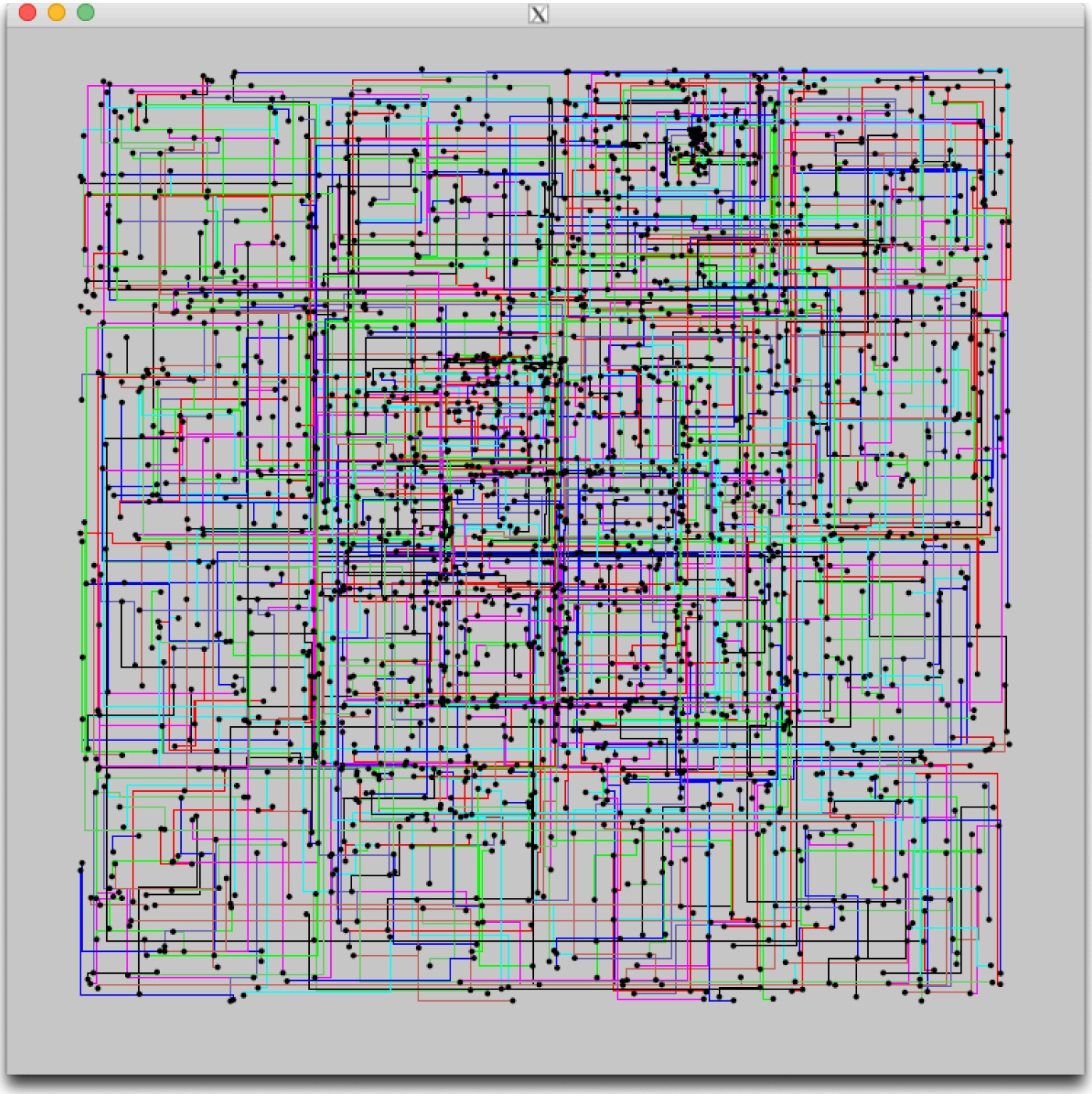
`code/inputs/timeinput/easy_4096.txt`



`code/inputs/timeinput/medium_4096.txt`



`code/inputs/timeinput/hard_4096.txt`

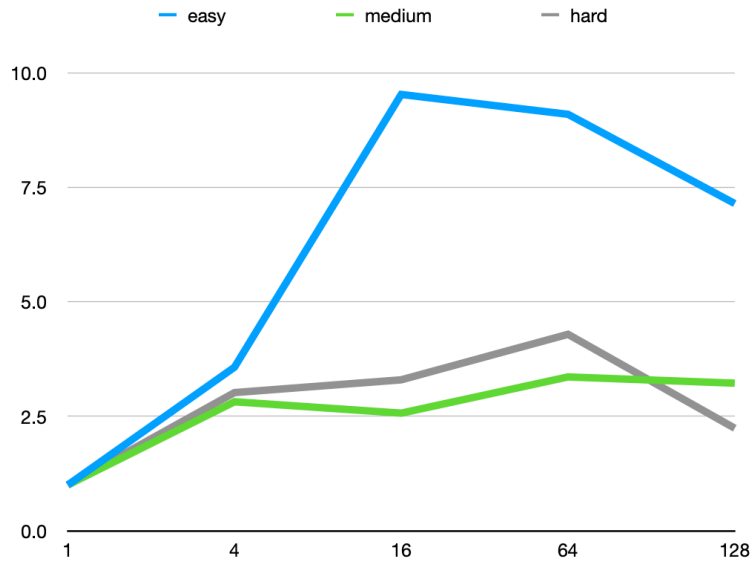


3

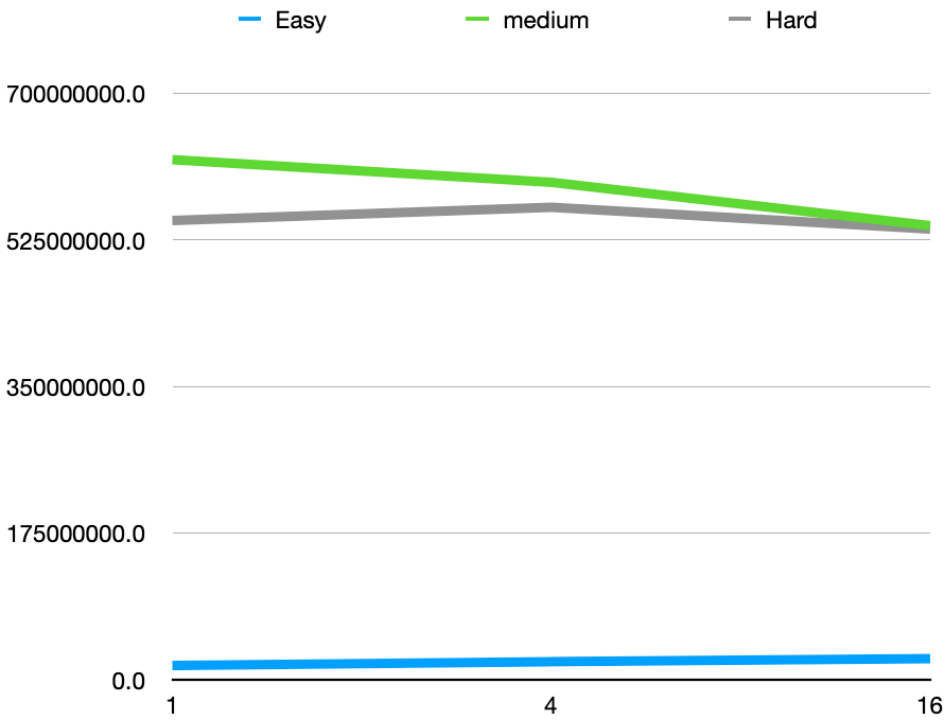
SpeedUp(Computation)	1	4	16	64	128
easy	1.0	3.6	9.5	9.1	7.1
medium	1.0	2.8	2.6	3.4	3.2
hard	1.0	3.0	3.3	4.3	2.2

	1	4	16	64	

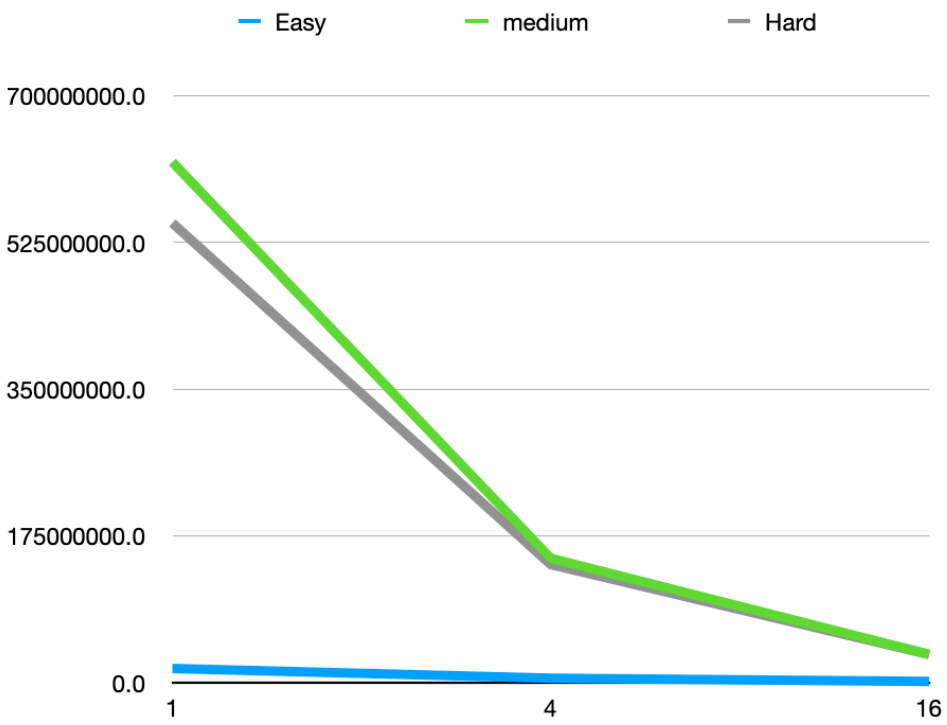
SpeedUp(Total)					128.0
easy	1.0	3.5	9.1	8.7	6.9
medium	1.0	2.8	2.6	3.4	3.2
hard	1.0	3.0	3.3	4.3	2.2



4



5



6

- To be honest, my own code's performance is in my expectation. It is not ideal because the wires has to wait for each other. They have severe synchronization overhead.
- Cache-miss is stable for the entire program and will go down when have more threads. More threads and CPU cores means we have more cache. But the cache miss does not go down very much. I think it means there so many memory is being used and cache sizes is small compared to the memory being used.
- Cache-miss per thread go down is as expected. Cache-miss for entire program is stable so for each thread will go down.