Parallel Computing Assignment 3

1. Design and performance debugging journey for your Within-Wires approach

- In the Within-Wires approach, we parallelize the path exploration. For any two points, each thread will explore the path both vertically and horizontally. There are dx + dy paths for any two points. Consequently, dx + dy threads are concurrently exploring these paths. This is our first and ultimate design of the within-wires strategy because this approach to implement within-wires approach is very intuitive and it was the first thought that came to our mind. The only minor improvement we introduced is early processing random wires. In the first, we chose whether to use the optimal route or a random route at the end of a wire's processing. This method wasted resources in getting the optimal route because if the random route was chosen, the optimal route is discarded. After realizing, we prioritized random checks immediately after processing a wire.

- To optimize the process for each wire, we have endeavored to maximize parallelization in this final implementation. We believe this is the only way to implement the within wires approach. Each wire has delta x + delta y potential routes, we parallelize all routes, removing the old route and examine delta cost after one route is placed. The initial best route is set to the current route. if we find a better route, update the best route as the current one until we have checked all routes. Then adding the best route to occupancy.

- Within-wires approach doesn't require much synchronization because each time we only process one wire. The only synchronization is applied when updating the optimal wire. If one thread finds a potential better route, it will enter a critical region enclosed by omp critical. When it enters the critical region, it compares its local cost with the global one, and updates the global cost and best wire when necessary. We have one trick to reduce synchronization to reduce overhead. Each thread keeps a local potential best cost to avoid unnecessary global best cost access.

```
#pragma omp parallel for default(none) shared(wire, delta_cost, delta_x, delta_y,
occupancy) firstprivate(private_wire, private_delta_cost)
            for (int i = 1; i <= delta_x + delta_y; i++) {
                cost_t _delta_cost = set_bend<true, false>(i, &occupancy, private_wire);
                if (_delta_cost >= private_delta_cost) {
                    continue;
                }
#pragma critical
                if (_delta_cost < delta_cost) {
                    {
                        delta_cost = _delta_cost;
                        wire.bend1_x = private_wire.bend1_x;
                        wire.bend1_y = private_wire.bend1_y;
```

```
                }
            }
            private_delta_cost = delta_cost;
        }
```

- Our speedup is close to the benchmark, close to linear but not. The most important reason is that workload is not perfectly balanced, especially for those wires with close start and end. Supposing a wire has very far start and end, workload should be distributed equally between all processors. But this is not the case, if the distance between start and end is close, there are only very limited potential routes and we cannot utilize all processors to calculate. Say, if the start and end are only two units away, like the start in (0, 0), the end in (1,1). There are only two routes, and we can assign them to two processors. If the program is run with 8 processors, 6 of 8 processors are idle, waiting for those two processors to finish.
  Furthermore, the time consumed for synchronizing also adds to the overall computation time

- As the number of processes increases, the speed improvement becomes less and less significant. Testing with GHC, the speedup is very linear when increasing the number of threads from 1 to 8. However, 128 processes have almost the same runtime as 64 processes.
  This phenomenon is partially caused by increasing synchronization and scheduling overhead. The more threads there are, the greater the negative impact of load imbalance. All threads must wait for the slowest thread which accounts for this phenomenon.

2. Design and performance debugging journey for your Across Wire approach

- We contemplated two designs for the cross-wire approach.
  The first design involves processing batches concurrently, with each wire within a batch being processed sequentially.
  The second design is to process each batch sequentially while concurrently processing each wire within the batch.
  After careful consideration, we chose to take the second design which processes batches sequentially and to handle each wire within those batches concurrently. In this way, we can also optimize workload balanced in the wires array.

- Processing batches concurrently could necessitate increased synchronization during occupancy matrix updates. Besides this design doesn't fit with the existing codebase and therefore challenging to implement.
  Processing wires concurrently, on the other hand, has many similarities with the within-wires approach, it is more intuitive and easy to understand. It fits the old code base, and can reuse many old functions.
  Therefore, we decided to implement the process wires concurrently inside a batch. We also use dynamic parallel loop scheduling to improve the workload balance. The speed up for dynamic scheduling is large.

- We concurrently process wires inside a batch. Each thread is in charge of one wire at a time, so there is no synchronization in accessing the wires vector. However, all threads have to read/update the occupancy vector. Reading race is not a big issue, some sort of staleness is inevitable. If we want to get rid of all race conditions in reading, it sacrifices too much performance and will be slower than one thread processing. We employ locking mechanisms when updating the occupancy matrix. This lock only works in across-wires approach, within-wires doesn't have race in updating occupancy.

```
template<bool CalculateDeltaCost, bool UpdateOccupancy, bool UseLock>
cost_t update_point(const int x, const int y, std::vector<std::vector<int>> &occupancy, const int delta) {
    /* Update the occupancy count of a point (x, y) */
    cost_t delta_cost = 0;
    if constexpr (CalculateDeltaCost) {
        delta_cost = (occupancy[y][x] + delta) * (occupancy[y][x] + delta) - occupancy[y][x] * occupancy[y][x];
    }
    if constexpr (UpdateOccupancy) {
        if constexpr (UseLock) {
            omp_set_lock(&lockTable[y][x]);
            occupancy[y][x] += delta;
            omp_unset_lock(&lockTable[y][x]);
        } else {
            occupancy[y][x] += delta;
        }
    }
    return delta_cost;
}
```

- In GHC, the performance improvement with respect to the number of processors is very close to linear. The small dropoff is caused by workload imbalance, scheduling and communication cost.
  When running on PSC, that is, when the number of threads exceeds 16, the dropoff in performance gains becomes more significant, and the runtime even increases when increasing the number of threads from 64 to 128.
  The obvious reason is that with the increase of threads, communication between different threads takes more and more time. And scheduling between different tasks becomes much more complex when there're a few threads. The cost of communication and scheduling doesn't grow linearly and will take a large chunk of performance.
  Another reason is with the number of processors increasing, if one thread has a large task and it needs more time to finish, the number of processors that are waiting for it increases. Tail latency in this case slows down the whole system.
  One potential reason is the architecture of the CPU in the GHC machine, which is an i7-9700 Processor with eight cores. Beyond eight threads, workload imbalance can occur, leading to each thread executing a minimal amount of work and incurring significant synchronization overhead to manage the increased number of threads, which in turn causes a performance drop-off.
- We took several tricks to improve the speedup. Comparing 8 threads to 1 thread, at first the speedup is only 2.xx times faster and the result cost was far from optimal if the batch is large, after optimization, we reached over 7 times speed and maintained a total cost when the batch size is large.
  - Writing optimal/random wire back to occupancy immediately. At first, we wrote back result wires back to the occupancy after the task had finished and wrote all wires back sequentially. This resulted in a more stale occupancy vector and sequentially writing was slow. To avoid data race when writing to occupancy, we introduced lock in updating the occupancy. We created a large lock table, each coordinate had one individual lock. Before updating one coordinate, the thread had to get the lock first. The small granularity of lock ensures minimal synchronization and no data race.
  - Sorting wires in (delta x + delta y)'s descending order. (delta x + delta y) is all possibilities for a wire. Sorting wires first ensures that each batch contains wires of similar complexity. And descending order ensures larger tasks are processed firstly to avoid too many processors idly waiting for one processor executing a large task.
  - Prioritizing random check to avoid redundant optimal path finding.
  - Utilizing dynamic loop scheduling in parallel for-loops. It balanced the workload, introducing communication overhead though.

```
./wireroute -f inputs/timeinput/medium_4096.txt -n 8 -m A -b 50
Number of threads: 8
Simulated annealing probability parameter: 0.1
Simulated annealing iterations: 5
Input file: inputs/timeinput/medium_4096.txt
Parallel mode: A
Batch size: 50
Initialization time (sec): 0.0557728140
```

```
Computation time (sec): 1.4365504000
Max occupancy: 3
Total cost: 607625
```

With Dynamic Scheduling

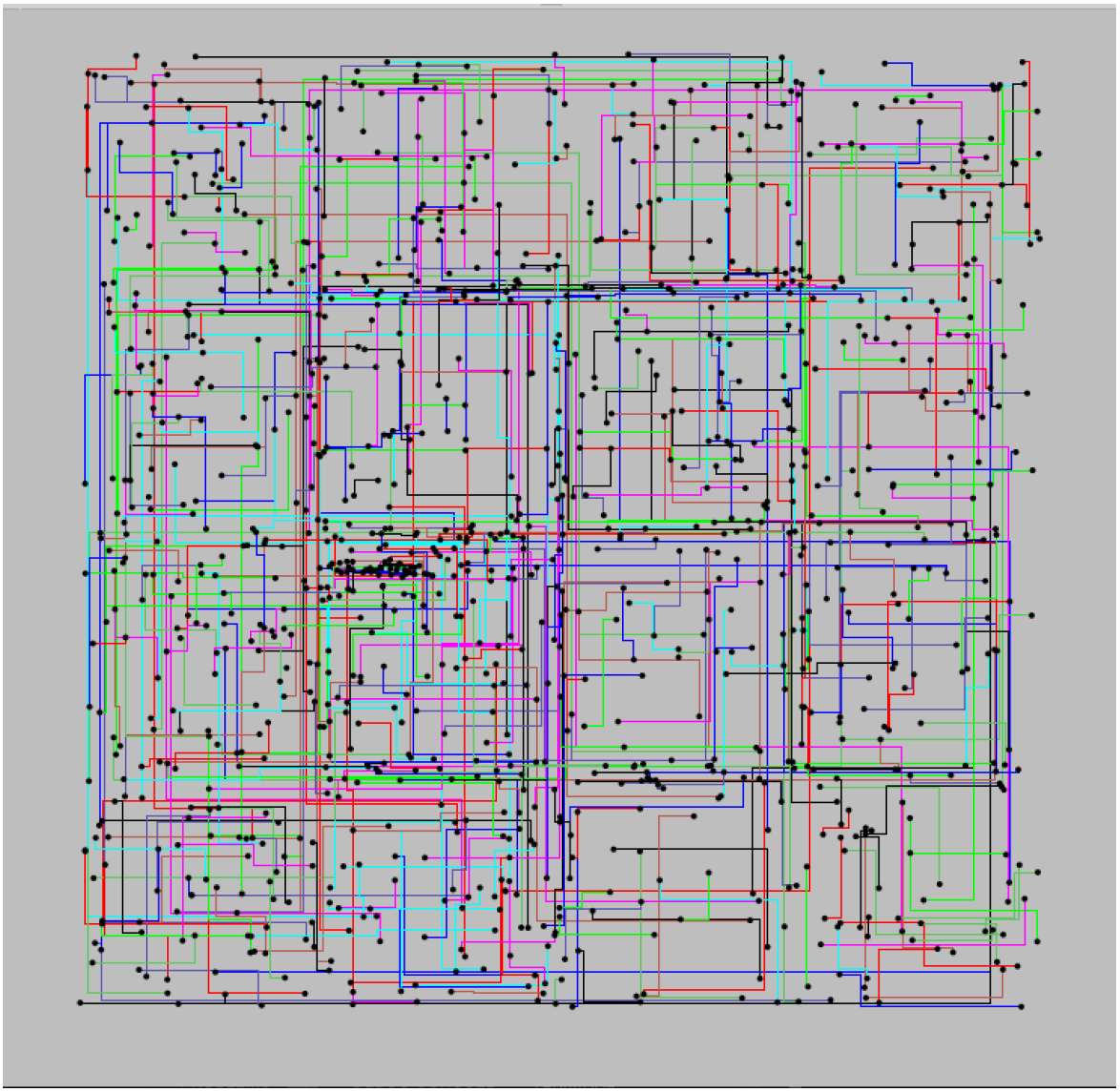
```
Number of threads: 8
Simulated annealing probability parameter: 0.1
Simulated annealing iterations: 5
Input file: inputs/timeinput/medium_4096.txt
Parallel mode: A
Batch size: 50
Initialization time (sec): 0.0999121160
Computation time (sec): 2.7129008540
Max occupancy: 3
Total cost: 609989
```

Without Dynamic Scheduling

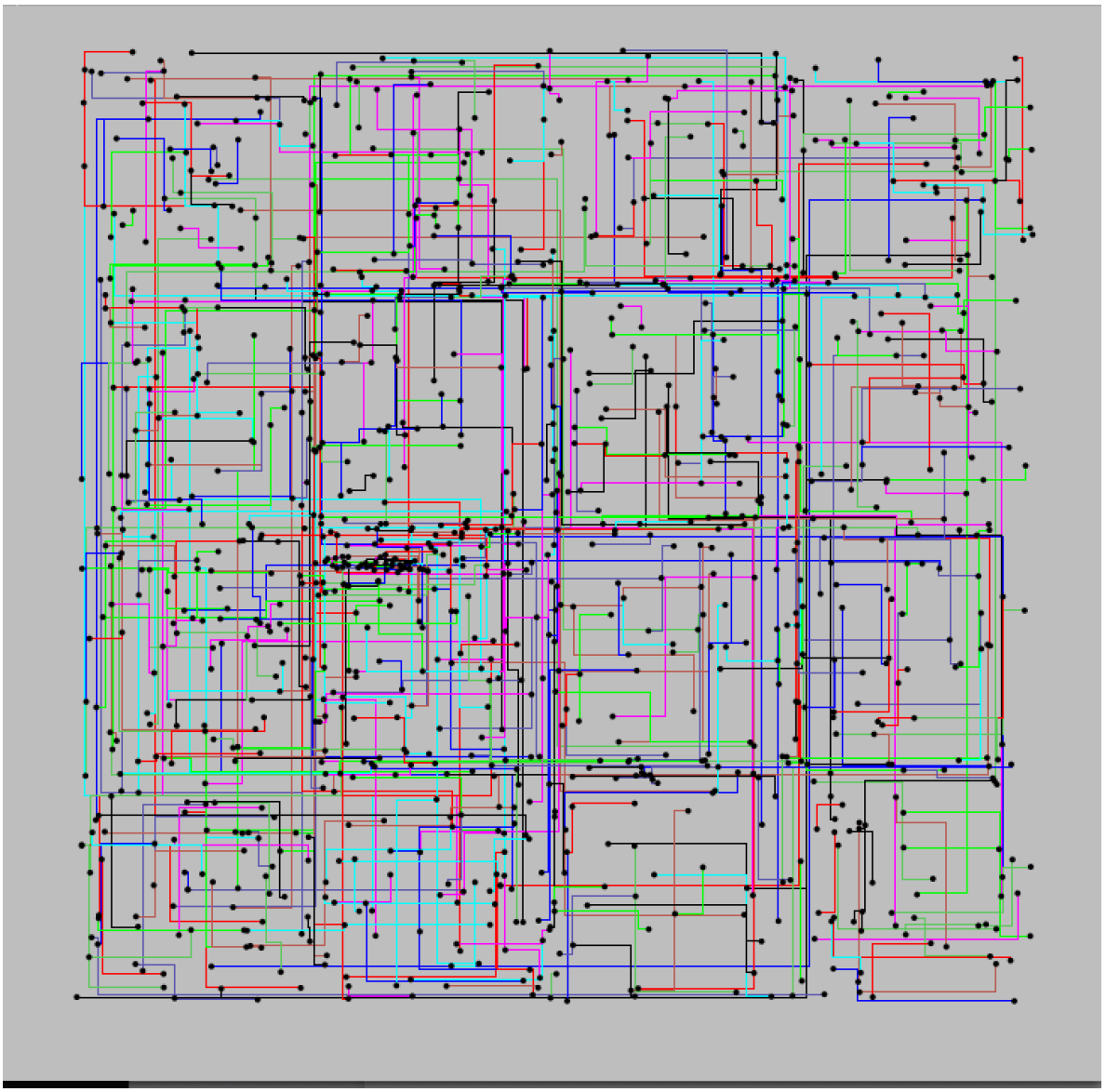
3.Routing output

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 screenshots of the output from the WireGrapher.java program that we provide.)

With Wire output

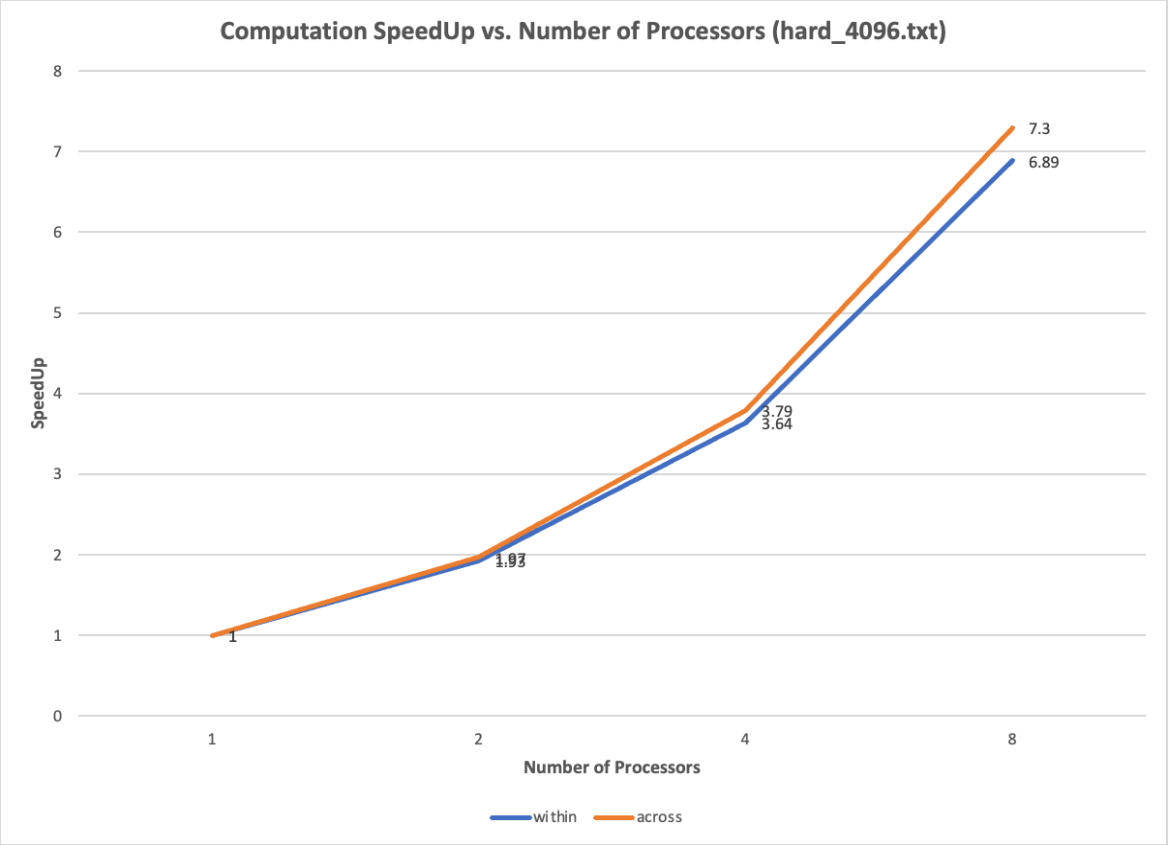


Across Wire output



4. Experimental results from the GHC machines:

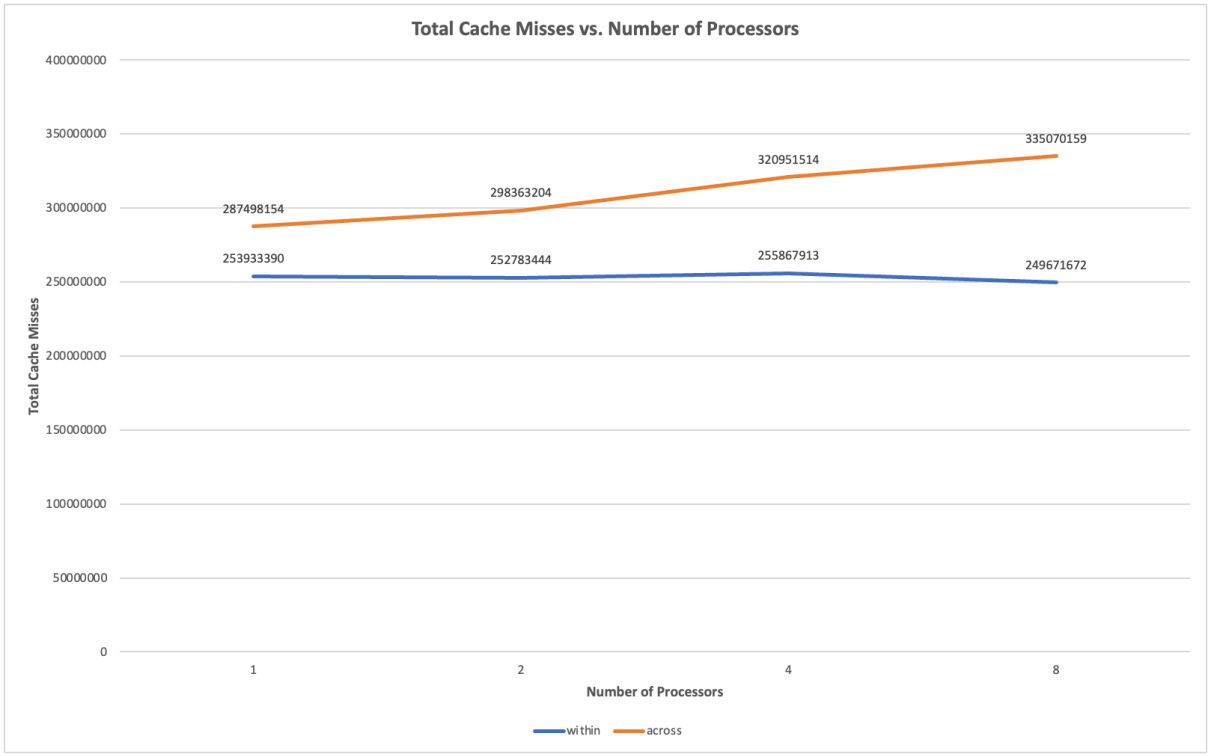
(a) Speedup graphs: Show a plot of the Total Speedup and Computation Speedup vs. Number of Processors (Nprocs). Discuss these results, including any non-ideal behaviors.



We tested hard_4096.txt, the speedup over the number of processors is very close to the benchmark. The overall trend meets our expectations for both plots. They show a nearly linear speedup with the increase of the number of processors.
Across-wires strategy shows a consistent advantage of speedup over within wires, but the margin is pretty small. One potential explanation to the small difference is that across-wires strategy has better load balance compared with the within-wires strategy. Within-wires strategy should achieve good workload balance if all wires' start and end are far enough, such that all potential routes can be distributed nearly equally to processors. But the assumption is not true, there're always some wires with close start and end points such that they can not be processed efficiently with within-wires strategy. By comparison, across-wires strategy doesn't require this assumption, it allocates a batch of wires at a time. If one processor finishes a wire's task, it can grab another task from the batch. Since dynamically scheduling is applied in our implementation, if the batch size is big enough, load can achieve greater balance.

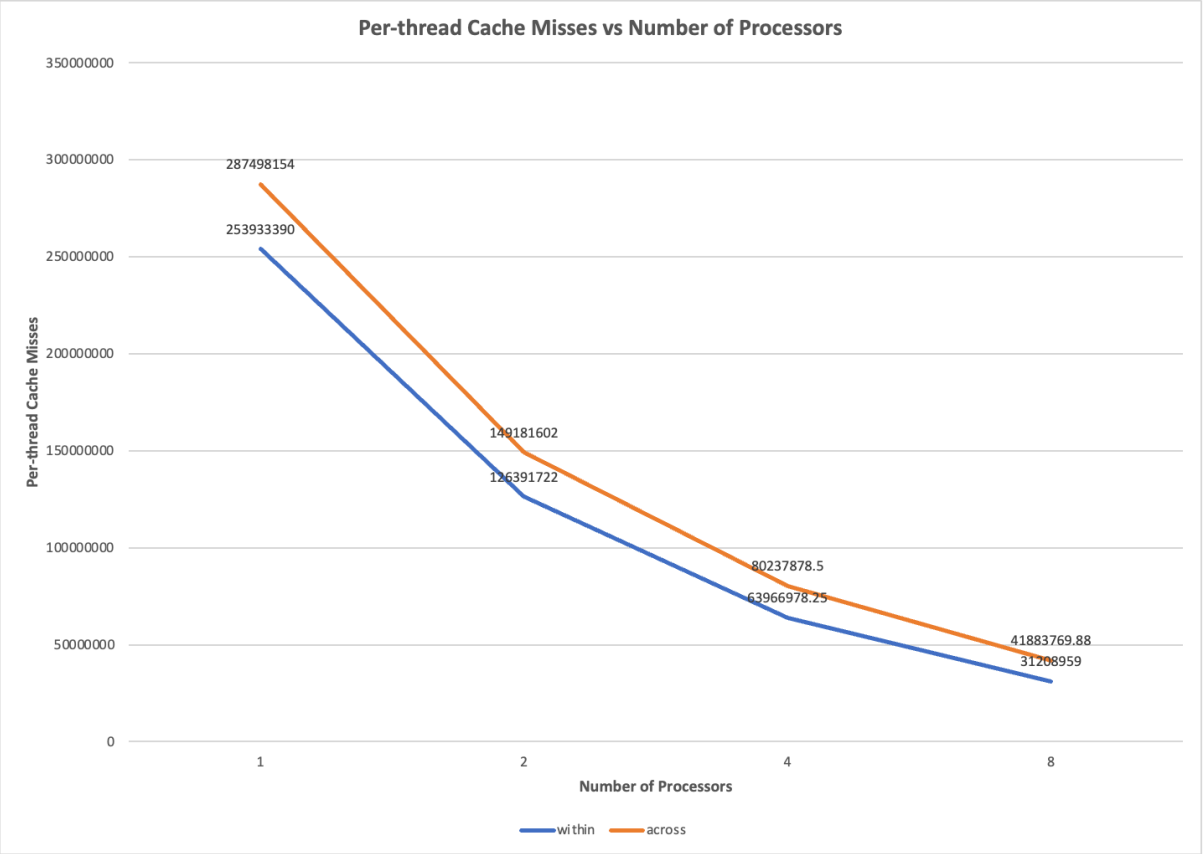
(b) Cache misses:

i. Total cache misses: Show a plot of the total number of cache misses for the entire program vs. Number of Processors (Nprocs).



---

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 (Nprocs).



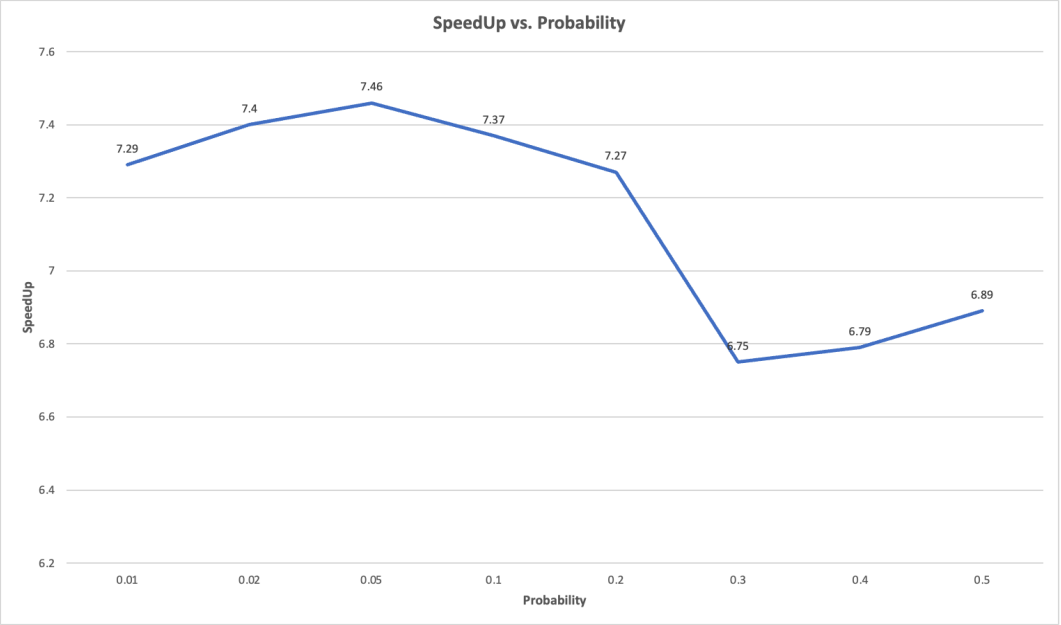
iii. Discussion: Discuss the trends that you see in these cache miss plots, including any surprises, and how these numbers relate to your speedup graphs.
Regarding the first plot, in the "across wires" strategy, the total number of cache misses slightly increases with the addition of more processors, whereas in the "within-wires" strategy, it remains almost unchanged. The "within-wires" approach consistently demonstrates an advantage over "across wires," with fewer cache misses regardless of the number of processors involved. In the "within-wires" method, a single wire is assigned to multiple processors, making the situation essentially the same no matter how many processors are used. On the other hand, the "across wires" strategy significantly differs as each processor independently processes a wire. As the number of processes increases, the span between each processing instance becomes larger, reducing the likelihood of utilizing the previously cached data.

The second plot shows that the per-thread cache misses significantly decrease with the increase in the number of processors, both for across-wires and within-wires strategies. For within-wires, this is easy to understand: regardless of the number of threads, the total cache misses are roughly the same, hence the number of misses per thread decreases. For across-wires, the reduction is under the premise that the overall increase in cache misses is slower than the increase in the number of processors. The first graph indicates that the rate of increase in total cache misses is actually not fast, much slower than the increase in the number of processors. The primary reason for this is that as the number of threads increases, the workload per thread significantly decreases, leading to a reduction in

per-thread cache misses. However, the situation where within-wires results in fewer cache misses compared to across-wires remains unchanged.

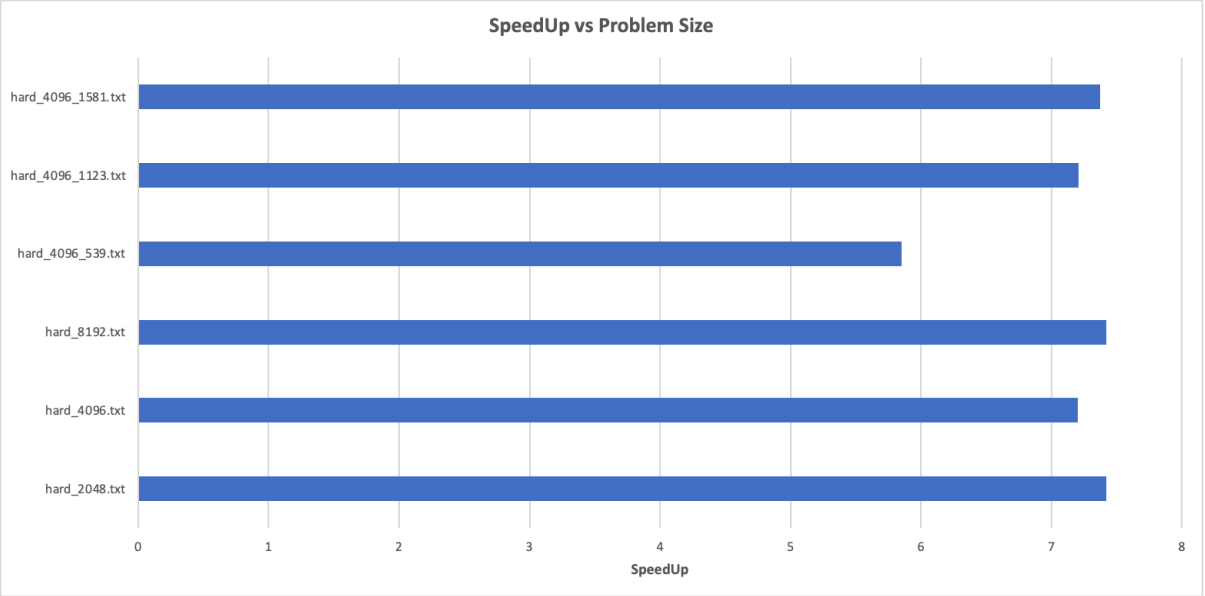
5. Sensitivity studies on the GHC machines:

(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.



The speedup does not scale linearly with the probability, the speedup initially improves and peaks at 0.05, then deteriorates and reaches the lowest point at 0.3, then it improves again. This looks really bizarre at first sight, I cannot find a universal explanation for this plot.
I expected that as the probability increases, the speedup would decrease due to increased load imbalance. When the probability is hit, the program will have an early return and result in a very short task, therefore the task may not be distributed very equally if there are many processors.
I still believe that fluctuations in speedup with increasing randomness might indicate that the workload balance between threads is not even. My previous expectation might explain why the speedup decreased between 0.05 and 0.3. Before 0.05, more random early returns happened, such that there would be some very small tasks and they could fill in between larger tasks and make the overall distribution more balanced. After 0.3, there were so many random early returns happening, and early returns were very uniform tasks that took almost the same time. This would make the overall workload distribution more balanced than before.

(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.



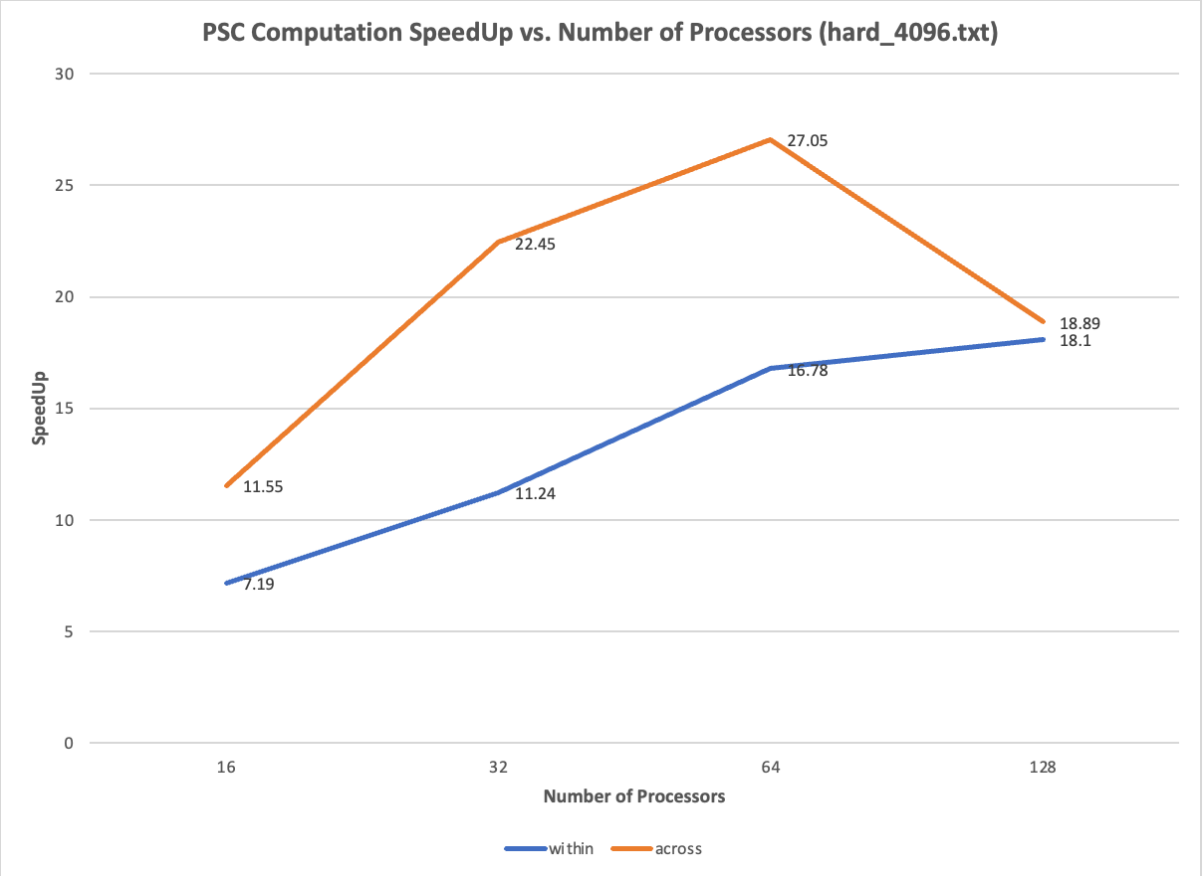
For the number of wires, the more wires, the higher the speedup. The speedup is substantially lower when there are only 539 wires. Speedup in 1581 wires is slightly higher than 1123 wires' case. This difference is easy to explain, for all cases I used the batch size of 256. The first reason is the 539 sample size does not fit 256 well. 539 = 256 * 2 + 27, choosing a different size like 135 will produce a better speedup. In this case, the speedup, as I tested, is 6.42. The second reason is that, when the number of wires is limited, distribution of wires can be more unequal. Some processors have to spend more time processing and the other processors have to wait. When the number of wires reach 1123, it is large enough such that there is not a large room to improve.

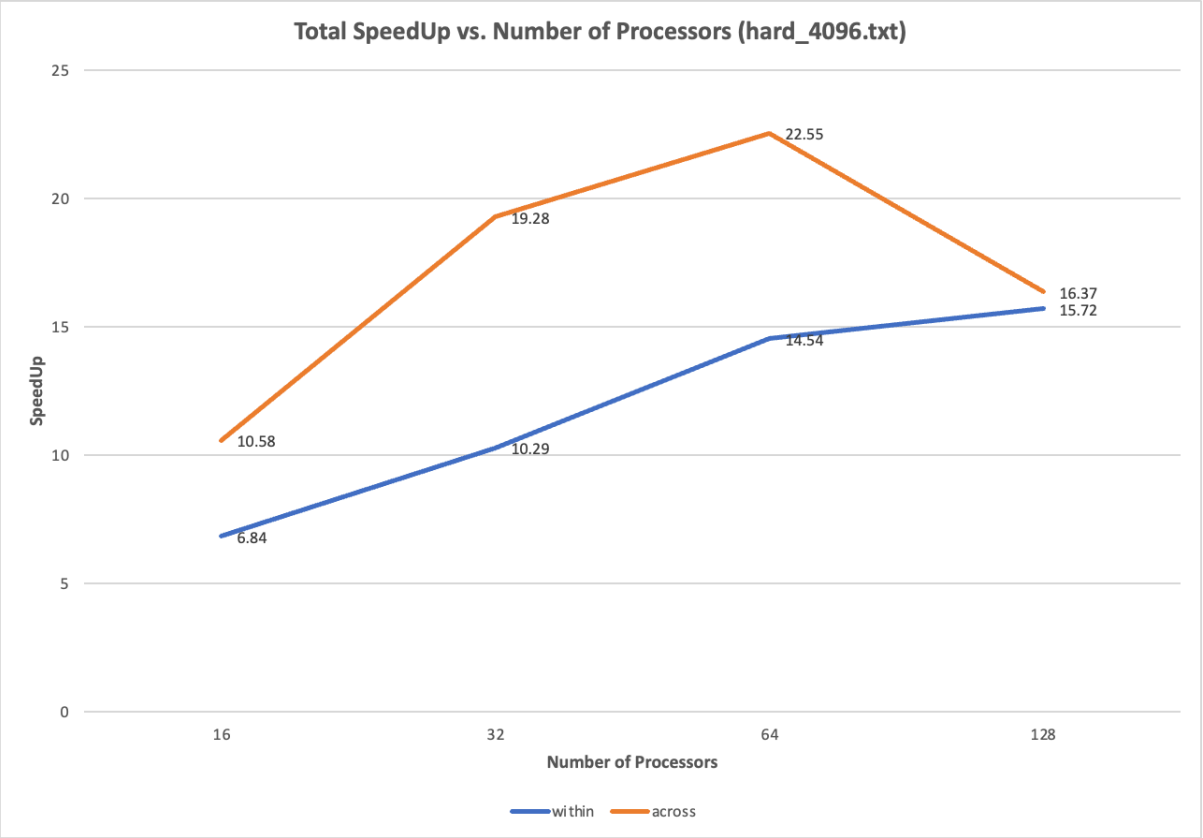
Regarding grid size, the impact on speedup appears marginal yet follows a distinct pattern. The highest speedup is achieved with both 8,192x8,192 grid and 2,048x2,048 grid, while the lowest corresponds to a 4,096x4,096 grid. This trend is perplexing, and while I cannot definitively explain it, I speculate that larger grid sizes allow for more extensive computation per wire, effectively reducing the proportion of processors' idle time. Conversely, smaller grids, such as 2,048x2,048, narrow the computation time variance among wires, potentially diminishing speedup benefits. The intermediate grid size of 4,096x4,096 yields the least favorable speedup, suggesting a complex interaction between grid size and computational efficiency that warrants further investigation.

---

6. Experimental results from the PSC machines:

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





The PSC is equipped with dual AMD EPYC 7742 processors, offering 2.25-3.40 GHz and 128 cores per node. In an ideal scenario, the speedup would scale linearly from 16 to 128 processes. However, our data shows that the most substantial speed improvement caps at 64 processes. Beyond this point, the speedup gains are negligible for the within-wire approach and even decline for the across-wire approach. This performance bottleneck is attributed to the synchronization and communication costs between processes. While the within-wire approach requires synchronization for comparison operations, the across-wire approach incurs more synchronization overhead due to frequent updates to the occupancy matrix. Consequently, the greater synchronization demands of the across-wire approach lead to more significant performance degradation compared to the within-wire approach.

(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.

The general trend observed is that speedup increases with a certain range of process counts, reaching a plateau or even declining beyond a specific threshold. This range is influenced by the CPU's architecture, specifically the number of cores. The performance improvement for the across-wire approach consistently outpaces the within-wire approach, albeit with a higher synchronization cost. For the across-wire approach, the performance gains are more pronounced as the number of processes increases. However, this approach reaches a performance downturn more rapidly due to its substantial synchronization overhead. While the synchronization cost is less pronounced on the GHC with its limited core count, it becomes a significant factor impacting performance at higher process counts.