15-618 Parallel Computer Architecture & Programming

Project 3: Report **Vibha Arramreddy (varramre), Vedant Bhasin (vedantb)**

Within Wires

Initially, we implemented a serialized approach that was described in the handout, and summarized here in Algorithm 1. This approach consists of two for loops back to back, one that considers all routes that travel horizontally first and one that considers all routes that travel vertically first. We abstract away all wire placing/ tracing/ modifying functionality to a helper function which works based on the wire struct that contains endpoints and the first bend coordinates.

The serial implementation easily lends itself to within wires parallelization as each iteration of each of the loops are independent. The two main design considerations then are data sharing and scheduling.

Algorithm 1 Wire Placement Algorithm with Simulated Annealing Approximation

```
Input: Initial wire placement, Niters, probability $P$
for each wire do
    for $i \leftarrow 1$ to Niters do
        Calculate cost of current path (if unknown)  Set current path as the minimum path
        Consider paths that travel horizontally first  if any horizontal path has a lower cost than the current minimum then
            Update the minimum path
        Consider paths that travel vertically first  if any vertical path has a lower cost than the current minimum then
            Update the minimum path
        With probability $1 - P$, choose the current minimum path  Otherwise, pick a random path from the possible routes
```

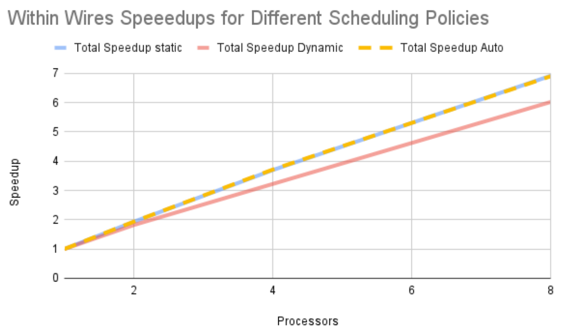
Data Sharing

The two key data structures being shared are the Wire struct and the occupancy matrix. For the Wire struct we use the *firstprivate* clause. The main reason behind this is that the struct is relatively lightweight (24 Bytes assuming 4 Byte ints), and it is being modified by each thread for the respective calculations. Having a thread local copy per processor ensures quick access, better cache performance, and reduced synchronization compared with a shared approach.

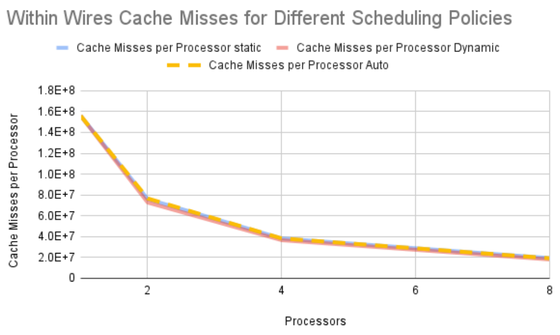
For the occupancy matrix we make the design decision to have it be shared across threads, this is because the occupancy matrix can grow quite large, so having thread local copies would be memory intensive. Crucially, our cost calculations function never modify the occupancy matrix ensuring quick accesses without the need for synchronization. Other shared variables include the lowest cost and the coordinates of the bends of the corresponding wire. The only critical section in this implementation is when these are being updated.

Scheduling

With simulated cosine annealing for each wire we either find the best placement or pick a random one. In our implementation, we calculate whether we will pick a random placement before calculating the best path; this avoids wasteful computation but introduces a workload imbalance among wires. We make the design decision of handling all random placements in the main thread, and only spawning worker threads when calculating the optimal placement. Within the optimal path calculation there is also a choice of what scheduling policy to follow for different paths. *a priori* we can hypothesize that a static scheduling policy would work well since each thread has to calculate the cost along a specific wire path, and the variation in wire length wouldn't be that significant. To verify our assumptions we conduct ablation studies for three different scheduling policies: static, dynamic, and auto. We measure the speedups for different processor counts and the cache misses per processor; these results are illustrated in Figure 1.



(a) Total & computational speedup within wires



(b) Total & computational speedup across wires

Figure 1: Total speedups for different scheduling policies on the within wires approach tested on hard_4096

Based on the results we can see that the auto and static polices display nearly identical results across both metrics, and outperform dynamic scheduling when it comes to speedups. However, per processor cache misses were identical across scheduling policies.

Across Wires

In order to parallelize the algorithm using the across the wires approach, we first followed the pseudo given the handout. We had one pragma loop to calculate the wire placement of B wires. This meant that all the threads were working on one batch of wires. Then we had another non-parallelized loop to update the occupancy matrix based on the wire placements. Then, in order to increase the efficiency of the algorithm, we changed to using a parallelized loop, using pragma atomic to make sure all updates to occupancy matrix were atomic. This ensured that even while calculating multiple wires we maintained the correctness of the occupancy matrix. This increased the speedup and the efficiency of our algorithm as it allowed us to more updates in parallel. We only had to serialize when there was contention.

We also made the decision to use dynamic assignment for our loop that calculated the batched wires. We made this decision for two reasons. First, we noticed that simulated annealing would cause some threads to finish work faster. Therefore, dynamic assignment would allow fast threads to pick up more work. Second, we also did a series of tests, shown in the table bellow that showed that dynamic assignment was the best option.

Assignment Type	**Time**
Dynamic	1.418236109
Auto	4.726397814
Static	4.626757169

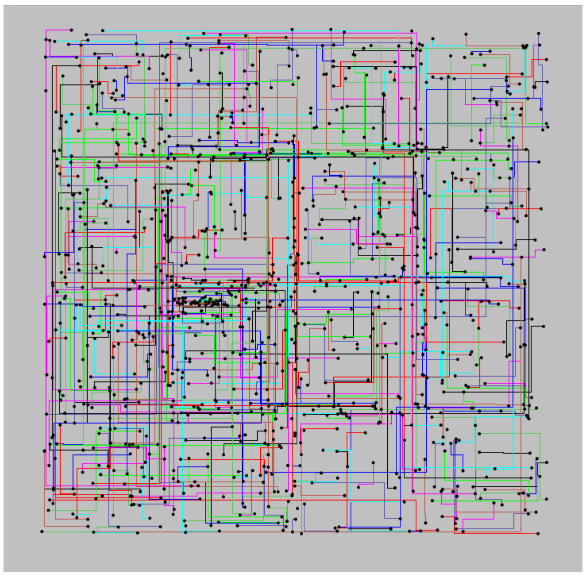
Table 1: Comparison of assignment types, using 8 threads and batch size of 512, on hard input

As we touched upon earlier the synchronization is in the implicit barrier between the two loops: one to calculate wire placement and other to update the occupancy matrix. There is also synchronization in the updates to the occupancy matrix by using pragma atomic to make sure updates happen atomically. As mentioned above, we used dynamic assignment to mitigate synchronization overhead. This helped make sure that threads that finished faster could get more work, mitigating the change that one slow thread would be a bottleneck.

The reason our code is unable to achieve perfect speedup is due to workload imbalance and synchronization cost. First, we though we used dynamic assignment, the workload still will not be perfectly balanced. This is due to many reasons such as threads picking up larger work that later. For example, a thread could have the first wires be randomly chosen, but the last wire need to be calculated. This would result in poor load balancing and prevent perfect speedup. Additionally, since there needs to be synchronization when updating the occupancy matrix this would cause updates then would need to be serial, preventing perfect speed up.

At higher thread counts, we noticed that the batch size also needed to increase to most efficiently use the threads available.

Routing Output


(a) WireGrapher output within wires

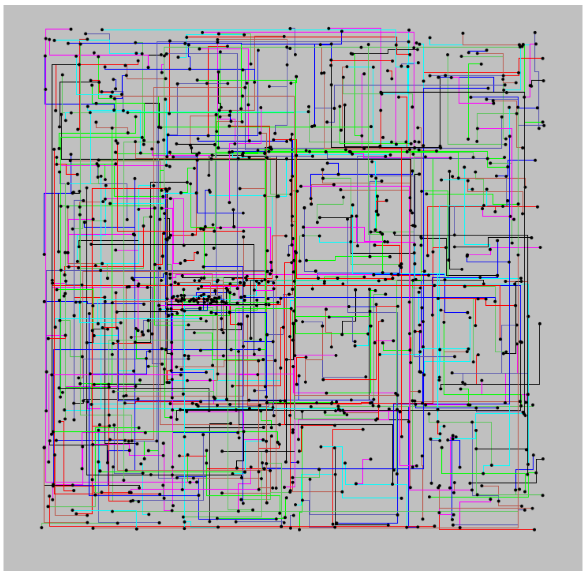

(b) WireGrapher output across wires

Figure 2: Routing visualizations within and across wires on medium_4096 with 8 threads.

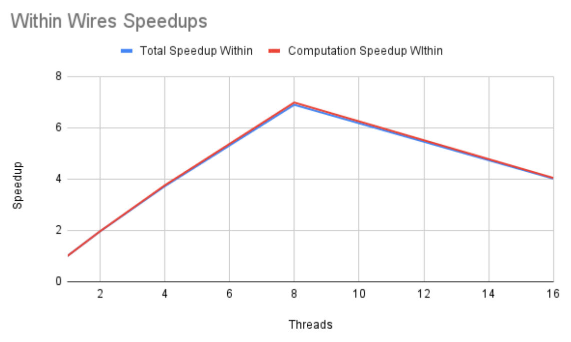
Experimental Results GHC

Speedups

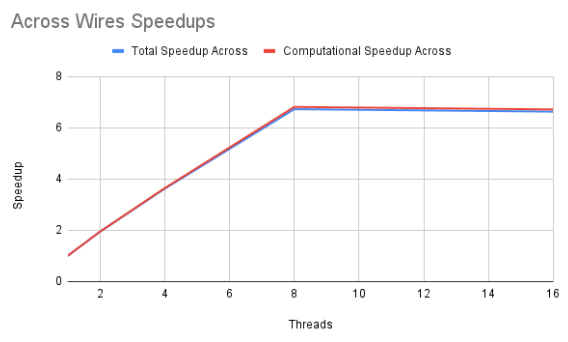
The total & computation speedups for both approaches are displayed in Figure 3. We observe near linear speedup going from 1 → 8 threads for both approaches, with the deviation from linear speedup increasing with the number of threads. For the within wires approach this is because as the number of threads increases so does the synchronization overhead as we have more threads competing for locks, and there is frequent synchronization to update the minimum cost and the best wire. For the across wires approach this is because of load imbalance due to some wires being placed randomly, and synchronization due to the occupancy matrix being updated.

In the range 8 → 16 the within wires speedup decreases to about 4×, while the across wires speedup remains at 8×. For the within-wires approach, each thread computes a certain number of paths for the placement of one wire, when the number of threads exceeds the number of processors the operating system needs to context switch which increases latency. Additionally, there are now more threads competing for locks leading to an increased synchronization overhead. In the within wires approach we have a synchronization point every time we find a lower cost path, which could happen frequently, this would lead to significant synchronization overhead as the number of threads increases. There is also a slight increase in cache misses, indicating that the context switching slightly worsens the spatial & temporal locality of the program since different threads might be reading from different parts of the occupancy matrix on the same processor.

For the across wires approach we see that the speedup remains at around 6.5× which is about the same as the speedup with 8 threads. In the across wires approach each thread calculates the placement of one wire. Crucially, each thread only writes to the shared data structure (occupancy matrix) once per function call, therefore it only has one synchronization point. This greatly reduces synchronization overhead incurred by additional threads which explains why we don't see the drop in speedup that we do with the within-wires approach, despite the degradation from context switching. Additionally, one of the main reasons for sub optimal speedup with this approach is load imbalance as some threads will be placed randomly, and others will have the optimal path calculation, increasing the number of threads doesn't change the bounds for the workload so it doesn't negatively affect speedup.



(a) Total & computational speedup within wires

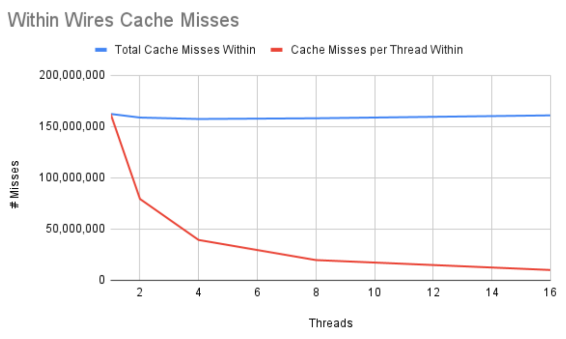


(b) Total & computational speedup across wires

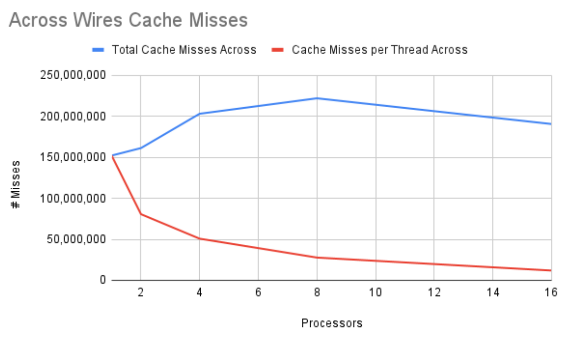
Figure 3: Total & computational speedups for both approaches tested on hard_4096

Cache Misses

For both approaches we see that the total number of cache misses stays constant even as the number of threads increases; cache misses per processor decrease greatly as the number of threads increases. This can be attributed to the fact that each thread is now responsible for less data. This can cause better cache locality, increasing the chance that the data a threads needs will already be in the cache. This keeps the total cache misses relativity constant and reduces the cache misses per thread. Also note that since we running a max of 16 threads on 8 CPUs, we do not run into the problem of too many threads trying to use the cache.



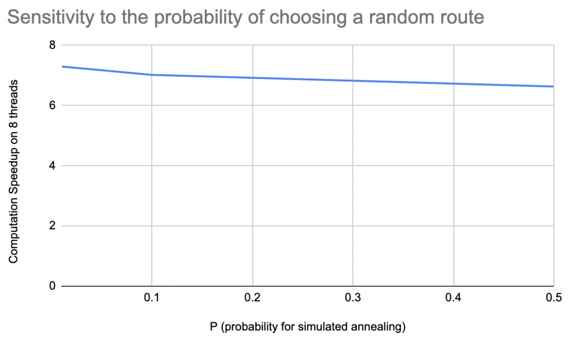
(a) Total & per thread cache misses within wires



(b) Total & per thread cache misses across wires

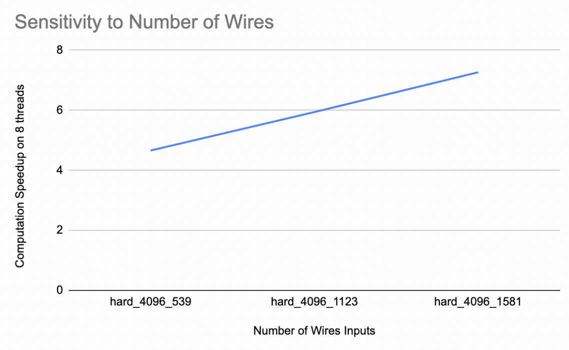
Figure 4: Total & per thread cache cisses for both approaches tested on hard_4096

Sensitivity Studies GHC

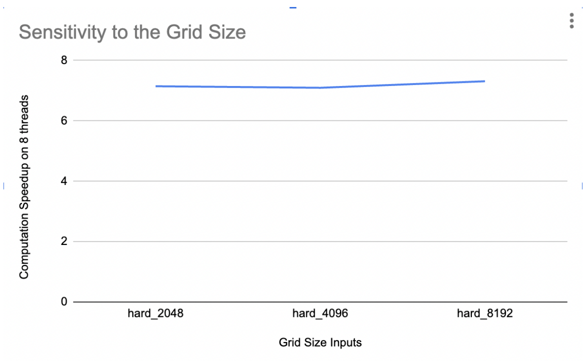


(a) Sensitivity to P

Figure 5: Sensitivity to the probability of choosing a random route



(a) Sensitivity to Number of Wires



(b) Sensitivity to Grid Size

Figure 6: Sensitivity to problem Size

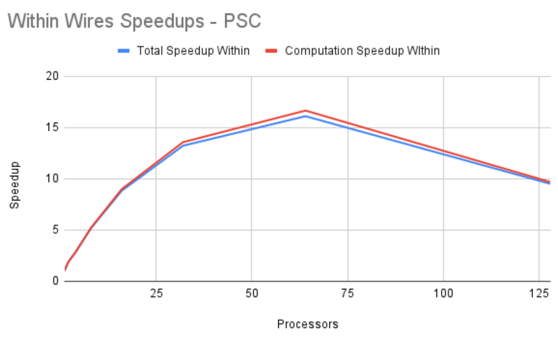
The speedup slightly decreases when the probability of choosing a random route through simulated annealing increases. This can be explained by load-balancing. Even though we use dynamic assignment in our across the wires approach to mitigate this issue, increasing the probability of the choosing a random wires increases the chances that threads get unequal work. This is because using a random wire placement is much less work than calculating the best wire placement. Thus, this can lead to inefficient use of resources and decrease speedup. Furthermore, as P increases, the single processor becomes faster due the decrease cost of choosing a random wire. This means that the overhead to manage multiple threads being relatively more costly.

The speedup increases as the number of wires increases, but stays the same when the grid size increases. Across the wires implementation is designed to be able to calculate the route of multiple wires at once. Therefore, as the number of wires increases, the amount of work that can be done in parallel increases. This would lead to higher utilization of threads and increased speedup. Since increasing grid size doesn't change the number of wires but the length of the wire, it follows that the speedup would be constant in this case. Since we are doing an across-the-wires approach, the amount of work that can be done in parallel stays the same causing speedup to also remain constant.

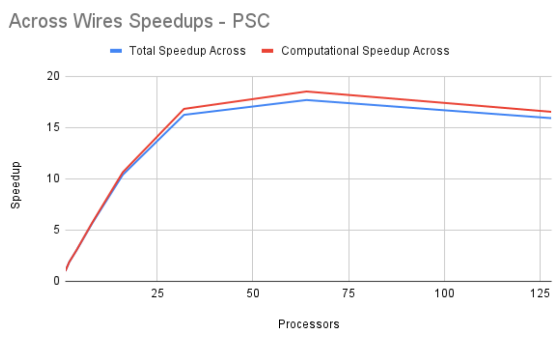
Experimental Results PSC

Speedups

In terms of general trends we can see that for both approaches speedup improves with number of processors sub-linearly up till 64 processors , and then worsens from 64 $\rightarrow$ 128. The reason for this decrease comes from the PSC RM node hardware, the specs are summarized in table 2. The key point here is that 64 CPUs map to a NUMA node, so these will have faster memory access, when we increase thread count beyond this, memory transfer needs to occur across nodes which has much higher latency. The reasons for suboptimal speedup are the same as discussed earlier, in a shared address space models, the number of additional threads leads to greater synchronization overhead. We can also see that within wires scales worse than across wires in part due to increasing cache misses.



(a) Total & computational speedup within wires

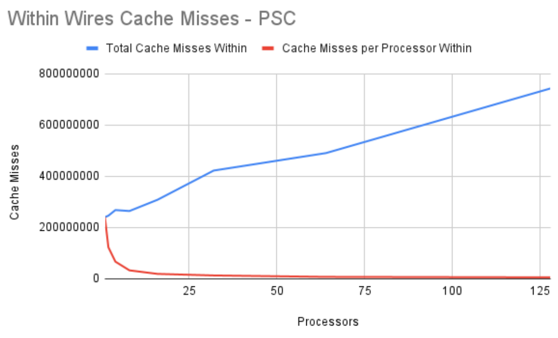


(b) Total & computational speedup across wires

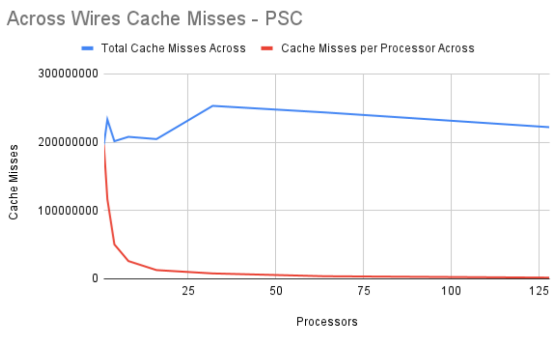
Figure 7: Total & computational speedups for both approaches tested on hard_4096 on a RM PSC Node

Cache Misses

While per processor cache misses still drop drastically across both approaches, the number of total cache misses for the within wires approach seems to increase linearly, while, for the across wires approach it remains relatively constant. This is due to the fact that the across wires approach operates at the granularity level of batches, with the batch size being constant, this leads to about the same total cache misses as we increase the number of threads. However, for the within wires approach, the granularity does not stay constant. Since each thread is working on its own wire, each thread needs to cache a certain amount of data. As the number of threads increase, the amount of data that needs to be cached also increases. However, the cache size stays the same, leading to an increased amount of total misses. Note that the cache misses per processor still decreases slightly as the number of processors increase.



(a) Total & per thread cache misses within wires



(b) Total & per thread cache misses across wires

Figure 8: Total & per thread cache cisses for both approaches tested on hard_4096 on a RM PSC Node

Attribute	**PSC**	**GHC**
CPU(s)	128	8
Thread(s) per core	1	1
Core(s) per socket	64	8
Socket(s)	2	1
L1d cache	32 KiB	32 KiB
L1i cache	32 KiB	32 KiB
L2 cache	512 KiB	256 KiB
L3 cache	16384 KiB	12288 KiB
NUMA node(s)	2	1
NUMA node0 CPU(s)	0-63	0-7
NUMA node1 CPU(s)	64-127	-

Table 2: Comparison of PSC and GHC Hardware Specifications