Parallel solver for the frequency assignment problem for a system of wireless mics

Shannon Ding
sding2@andrew.cmu.edu

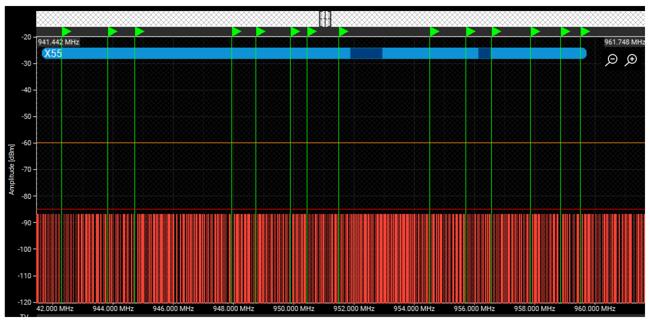
1 Summary

This project implements a parallel algorithm to find a set of compatible radio frequencies for a large set of wireless mics. It uses ISPC and a shared memory model and is run on GHC's 8-core Intel Core i7 machines. There are multiple implementations used to benchmark performance: a sequential, parallel single core, and parallel multicore program. The parallel multicore version achieves 21x speedup compared to the sequential version for checking compatibility.

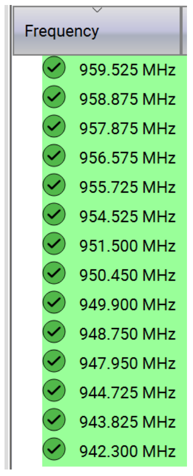
1.1 Background

The project attempts to solve a variant of the "Frequency Assignment Problem" which reduces to a vertex coloring problem, an NP-complete problem. This problem is complex because adding a wireless mic into a system creates interference, called intermodulation (IM) products, with every combination of every other mic in the system throughout the available frequency range. As a result, solvers need to perform an exhaustive search or some other computationally expensive approximation. We try to speed this search up by running it in parallel.

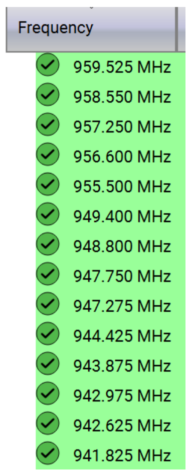
Below is an example of a maximally filled frequency assignment problem.



The green lines are the frequencies of successfully added mics. Each mic must be located at a frequency divisible by 0.025MHz. The red lines are intermodulation frequencies that each mic needs to be a certain distance away from, based on its compatibility profile. The number of IM products grow rapidly. The blue bar is the frequency range the mics have access to. This ~18MHz range can fit 14 mics using the following frequencies. There are multiple solutions to fitting 14 mics.

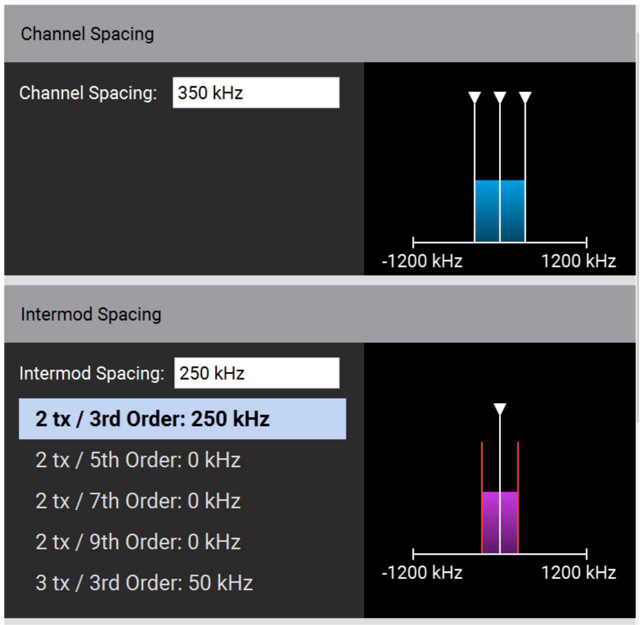


Frequency
✓ 959.525 MHz
✓ 958.875 MHz
✓ 957.875 MHz
✓ 956.575 MHz
✓ 955.725 MHz
✓ 954.525 MHz
✓ 951.500 MHz
✓ 950.450 MHz
✓ 949.900 MHz
✓ 948.750 MHz
✓ 947.950 MHz
✓ 944.725 MHz
✓ 943.825 MHz
✓ 942.300 MHz



Frequency
✓ 959.525 MHz
✓ 958.550 MHz
✓ 957.250 MHz
✓ 956.600 MHz
✓ 955.500 MHz
✓ 949.400 MHz
✓ 948.800 MHz
✓ 947.750 MHz
✓ 947.275 MHz
✓ 944.425 MHz
✓ 943.875 MHz
✓ 942.975 MHz
✓ 942.625 MHz
✓ 941.825 MHz

The mics have the following compatibility profile:


Channel Spacing

Channel Spacing: 350 kHz

Intermod Spacing

Intermod Spacing: 250 kHz

2 tx / 3rd Order: 250 kHz

2 tx / 5th Order: 0 kHz

2 tx / 7th Order: 0 kHz

2 tx / 9th Order: 0 kHz

3 tx / 3rd Order: 50 kHz

1.2 Data structures & operations

Spectrum: Store & lookup which intermodulation products are on each frequency

```
static inline int index_has_im(uniform int (&spectrum)[], int index, IMProduct im) {
    return spectrum[index * NUM_OF_IM_PRODUCTS + im];
}
```

Mic inventory: Get number and specs of each mic

```
// all frequency units are in 0.xxx kHz which are discrete multiples of 0.025kHz
struct Mic {
        int frequency; // the channel frequency it's assigned
        int band_low; // the lowest possible frequency allowed for the channel
        int band_high; // the highest possible frequency allowed for the channel
        int channel_spacing; // how far away the channel frequency should be from other channels
        int intermod_spacing[]; // how far away the channel frequency should be from IM products
};
```

Intermodulation products: Compute the result of each type of intermodulation product. We only consider five types of intermodulation product.

```
export enum IMProduct {
    im_chan, im_2T3O, im_2T5O, im_2T7O, im_2T9O, im_3T3O
};

static inline int check_im_product(int f1, int f2, int f3, IMProduct im)
{
    switch (im) {
        case im_2T3O:
            return 2 * f1 - f2;
        case im_2T5O:
            return 3 * f1 - 2 * f2;
        case im_2T7O:
            return 4 * f1 - 3 * f2;
        case im_2T9O:
            return 5 * f1 - 4 * f2;
        case im_3T3O:
            return f1 + f2 - f3;
        default:
            return 0;
    }
}
```

1.3 Algorithms

Analyze

Given a set of mics, answer the decision problem of if it is a compatible set of mics.

1. Outer loop through each mic with its compatibility profile
2. Inner loop through all possible combinations of IM products
3. Save the number of conflicts to memory

Optimize: Given a set of possible mics to fit, assign frequencies to fit as many mics as possible.

1. Keep track of the global spectrum by looping through an array for the entire frequency range.
2. Sequentially add one mic to the system at a time and update the spectrum.
3. After several failures to add a mic, end the algorithm

1.4 Parallelization

Analyze

- Workload is finding every single IM product in the system: O(k * N^3) where k is the number of IM products which is a constant and N is the number of mics in the system. Then, checking to see if any mics overlap with these IM products using an O(N) loop for a total of O(k * N^4) work
- Any of the IM product computations can be done independently. Any of the IM product to mic conflicts can be done independently. As a result, we can make the span anywhere from 1 to O(k * N^4) so we experiment to pick a span that balances both parallelism and overhead.
- We choose parallelization over mics to have a span of N.
- Data is arranged in a “struct of arrays” format so accesses are all continuous
- Each mic considers the system without it and checks if it can fit into the system safely

Optimize

- Workload is to sequentially add one mic at a time to an available frequency until no more frequencies are available which is an O(N) loop. Each iteration of the loop will need to calculate O(k * n^3) where 0 <= n <= N conflicts and check over the f available discrete frequencies. The total work is O(k * f * N^4).
- Instead of recalculating all IM products each iteration, we save this information to a spectrum array that stores all IM information from all existing mics. Each iteration, we only need to calculate all new IM products caused by the new mic, which is O(k * n^2). Using this piece of state reduces the total work to O(k * f * N^3).
- We parallelize over the primary frequency of a mic for a span of f.
- Data is similarly arranged to reduce gathers when possible. Unfortunately, random access to the entire spectrum is needed as IM products show up all over the spectrum and are not localized. This is performance bottleneck and why ISPC isn’t suitable for the optimize function.
- Each frequency checks if it would be okay to insert a mic at that point through

lookups on the spectrum rather than computing like analyze.

2 Approach

2.1 Infrastructure

The hardware used is 8x Intel i7 cores. These cores support AVX-512 which allows a vector of 16 integers (32 bit) to be computed on at once. The API used was ISPC, which compiles down to vector intrinsics.

2.2 Algorithms

Analyze: Given a set of mics, answer the decision problem of if it is a compatible set of mics.

1. Outer loop through each mic with its compatibility profile
    a. Split the N mics into 8 tasks, one for each core.
    b. In each task, use the foreach construct to assign a mic to each member of the gang.
    c. The computation experienced by each mic is identical, so the utilization should be 100%.
2. Inner loop through all possible combinations of IM products
    a. Each gang member will compute the IM products created by all N-1 other mics and check that it doesn't affect the mic it's in charge of.
    b. They do this through loops through various combinations of mics and calling check_im_product(f1, f2, f3, im) and verifying the IM product is farther than the compatibility profile of the mic it's in charge of.
3. Save the number of conflicts to memory
    a. Each gang member keeps a local running sum of all conflicts and saves it to it's own slot in a results[] array
    b. The results array is checked through after the ISPC program returns to give the final answer.

Optimize: Given a set of possible mics to fit, assign frequencies to fit as many mics as possible.

1. Keep track of the global spectrum by looping through an array for the entire frequency range.
    a. Split the f 0.025kHz frequency buckets into 8 tasks, one for each core.
    b. In each task, use the foreach construct to assign a frequency bucket to each member of the gang.
    c. Each gang member sees if assigning a mic to its frequency will be valid under current state of the spectrum.
    d. static bool check_self_chan_compatible_in_spectrum(uniform int n, int fnew_index, uniform const int (&compatibility)[], uniform int S, uniform int (&spectrum)[]); checks if the new mic won't experience interference
    e. static inline bool check_existing_compatible_in_spectrum(uniform int global_low,

uniform int global_high, int fnew, uniform const int N, uniform const int n, uniform int (&frequencies)[], uniform int (&spectrum)[]); checks if all existing mics won't experience interference caused by adding the new mic.

   f. If both checks pass, the frequency is marked in an array as valid.

2. Sequentially add one mic to the system at a time and update the spectrum.
   a. We only add one mic at at time because the frequency assignment problem is interconnected enough that adding multiple mics at a time will be difficult to move toward a clear solution. Note that this is a polynomial approximation and not the optimal solution, which is an exhaustive exponential search.
   b. Receive a list of valid frequencies for the mic. Randomly choose a valid frequency and update the spectrum state with this new mic added.
3. After several failures to add a mic, end the algorithm
   a. Repeat until as many mics as possible are added and there are no more valid frequencies for any mics.

2.3 Other Design

The serial algorithm was written to easily map to the parallel algorithm by replacing the outer loop of the serial algorithm with a parallel loop.

One point of experimentation was when to parallelize by mic vs. by frequency. The problem size grows O(N^4) by mics but linearly O(f) by frequency. Past a certain number of mics, it becomes more efficient to calculate once and store & lookup the IM product. This was demonstrated by the implementation of analyze (parallelization by mics) vs. optimize (parallelization by frequency) and had different performance implications in the serial vs. parallel runs.

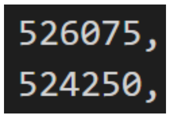
3 Results

3.1 Test Suites

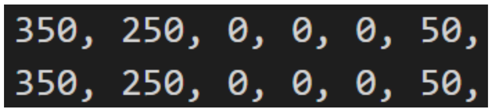
The program accepts a file with the number of mics, their frequency ranges, and their compatibility specs.

```
./wwb 16 hello-16.txt
```

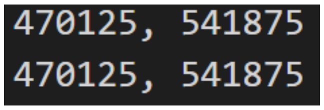
The first 16 lines of hello-16.txt should be 16 frequencies to check. The next 16 lines should be compatibility specs following the order of the IMProduct enum. The last 16 lines should be frequency range.



```
526075,
524250,
```



```
350, 250, 0, 0, 0, 50,
350, 250, 0, 0, 0, 50,
```



```
470125, 541875
470125, 541875
```

The program outputs a list of valid frequencies and a spectrum array of where the primary and intermodulation frequencies are located.

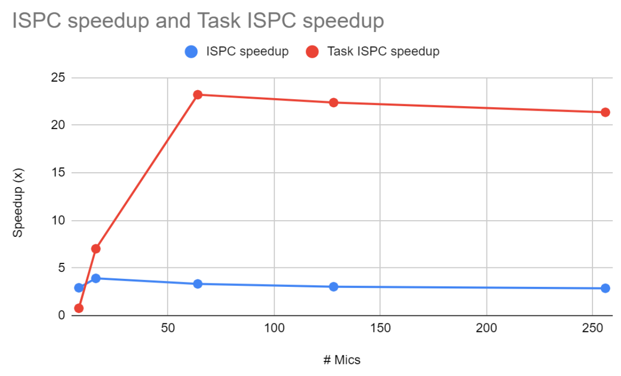
```
fit 16 frequencies
483325 512525 517475 515475 483500 475125 534900 501825
495875 530825 529500 537450 532550 510375 480825 480875
```

3.2 Analyze

Single core ISPC experiences consistent speedup even as problem size grows. This indicates the performance is limited by the SIMD width.

Task ISPC experiences drastic speedup past 64 mics. There are 8 tasks run simultaneously as there are 8 hardware cores.

Tasks are able to achieve almost 8x improvement on ISPC alone as the number of mics increase as speedup from ISPC alone decreases.

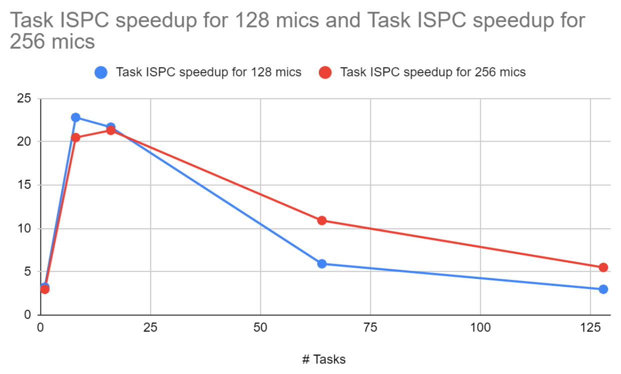


# Mics	ISPC speedup	Task ISPC speedup
8	2.92	0.77
16	3.91	7.02
64	3.33	23.21
128	3.03	22.38
256	2.87	21.36

The optimal span is 8 tasks for 128 mics.

The tasks should be very uniform since there are few conditionals in the program. ISPC speedup should preferably be 16x (the number of integers that can be run at once on the hardware), so this behavior is not completely expected.

The slowdown from task ISPC is caused by each individual task being too short – with 256 mics, the optimal span becomes 16.



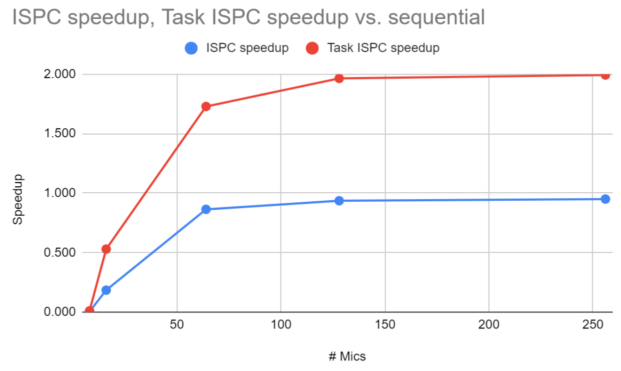
# Mics	Task vs. ISPC-only speedup
8	0.2636986301
16	1.795396419
64	6.96996997
128	7.386138614
256	7.442508711

# Tasks	Task ISPC speedup for 128 mics	Task ISPC speedup for 256 mics
1	3.23	2.97
8	22.81	20.49
16	21.69	21.32
64	5.92	10.91

128	2.98	5.5

3.3 Optimize

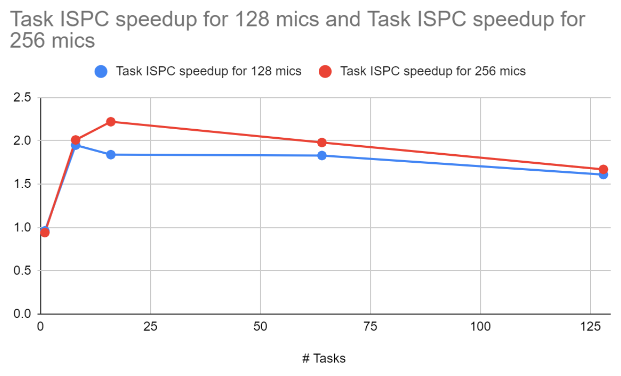
The first observation is that speedup is extremely poor, but does seem to scale with number of mics. It also appears to flatten out at 2x speedup for task ISPC vs. sequential.



# Mics	ISPC speedup	Task ISPC speedup	ISPC to Task Speedup
8	0.003	0.010	2.996
16	0.184	0.529	2.873
64	0.864	1.728	2.001
128	0.936	1.964	2.099
256	0.950	1.993	2.098

The optimal span is 8 tasks for 128 mics.

The tasks are not perfectly uniform since there are now loops that depend on conditionals. This is represented by task ISPC not having a “perfect” 8x speedup compared to single core ISPC.



# Tasks	Task ISPC speedup for 128 mics	Task ISPC speedup for 256 mics
1	0.96	0.94
8	1.95	2.01
16	1.84	2.22
64	1.83	1.98
128	1.61	1.67

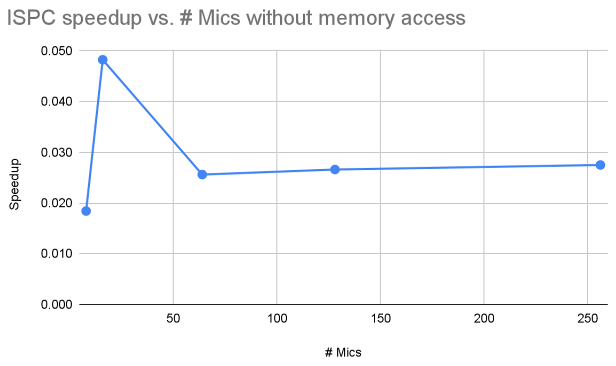
We have several hypotheses for the poor speedup.

1. Gather operation when reading the spectrum state
2. Overhead from starting ISPC programs
3. Expensive varying division operation in each iteration.

To test the first two hypotheses, we remove all gather memory accesses for the sequential and ISPC programs and see the speedup. Step 2 of the optimize algorithm involves starting a new ISPC program for each mic, so we also benchmark over the number of mics.

# Mics	Sequential time (ms)	ISPC time (ms)	ISPC vs. Sequential Speedup

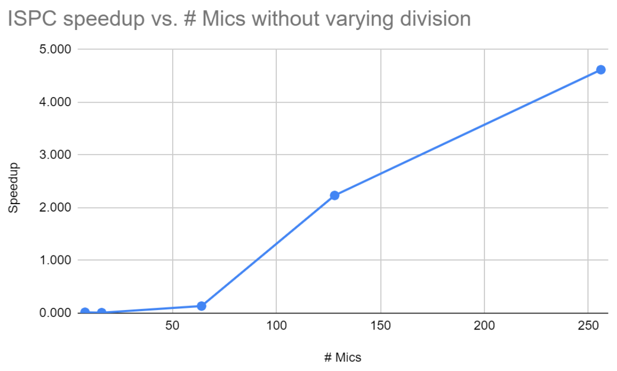
8	0.017	0.923	0.018
16	0.022	0.456	0.048
64	0.312	12.184	0.026
128	1.964	73.819	0.027
256	13.471	489.924	0.027



Speedup doesn't improve at all and instead gets worse when memory accesses are removed. This seems to indicate that the gather operation is not a major factor in the slowdown. Speedup does gradually increase with the number of mics (a.k.a. the number of ISPC function calls) so it appears the "overhead from starting ISPC programs" hypothesis is more convincing.

This still isn't a good hypothesis since overhead should more drastically decrease between 128 and 256 mics. We try the third hypothesis by replacing the ISPC division on varying values with a constant.

# Mics	Sequential time (ms)	ISPC time (ms)	ISPC vs. Sequential Speedup
8	0.013	0.735	0.018
16	0.008	0.789	0.010
64	0.274	2.037	0.135
128	1.974	0.884	2.233
256	12.202	2.642	4.618

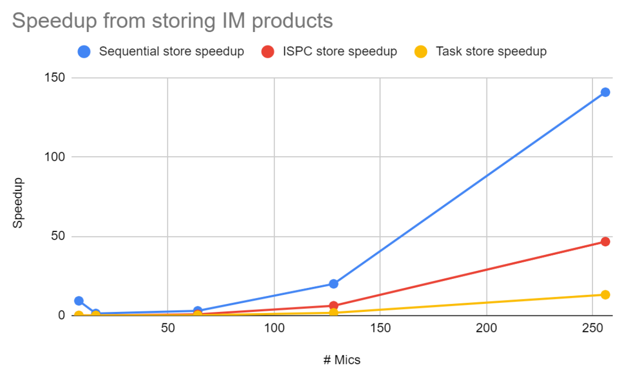


This results in the most drastic speedup and is the likely answer. Division is an expensive operation in general and this varying division operation happens a constant number of times per gang member. This doesn't seem significant but possibly since there are thousands of gang members, how division is implemented with vectors is very disruptive and causes gang members to almost run sequentially. There wasn't an immediately effective way to remove this varying division without impacting correctness, so it is kept in the code.

3.4 Storing vs. Recalculating Intermodulation Products

Although the speedup for optimize was poor (even after fixing the varying division issue), the absolute time spent on the optimize algorithm was quite efficient compared to analyze. Optimize does everything that analyze does, and even does it multiple (up to f times) so this performance improvement is drastic.

The main difference between optimize and analyze is that optimize uses state, an array called spectrum with all the IM products so far. Analyze recalculates all of these IM products each time. We look at how this scales with the number of mics.



Generally, we know that individual arithmetic operations are more efficient than memory accesses. This problem scales in O(N^4) in terms of arithmetic but only O(f) in terms of memory access, so the efficiency of memory access quickly starts to overtake recalculations.

# Mics	Sequential store speedup	ISPC store speedup	Task store speedup
8	9.285714286	0.01102539573	0.1252456742
16	1.330487805	0.06270295123	0.1003312495
64	2.992136026	0.7760674856	0.2227911483
128	20.01544163	6.18191877	1.756503753
256	140.8212315	46.61264759	13.13772772

There is 9x improvement from storing IM products from the sequential algorithm right away and rises up to 140x. The real improvement is even higher as optimize does more than analyze.

On and above 128 mics, storing IM products becomes more efficient than recalculating them.

4 Conclusion

4.1 Parallelism

There are multiple avenues for parallelism and each has their strengths and weaknesses on the frequency assignment problem.

1. Parallelism per mic to check whether it experiences any conflicts from the rest of the mic system.
    a. This form of parallelism scales with the problem, so it's helpful to keep the work per parallel worker fairly split. Unfortunately, the problem scales faster than the number of mics O(N^4) so there should be another axis of parallelism.
    b. This creates very uniform work with high utilization, but a lot of duplicate work. Past a certain number of mics, it's recommended to save the intermediate values for other parallel workers to use. This will increase the synchronization cost though.
    c. Memory accesses are generally random since IM products appear all over the spectrum.

2. Parallelism per type of intermodulation product to simultaneously compute a list of all unusable frequencies caused by each type of conflict
    a. This could be a good path for SPMD parallelism because each type of IM product uses the same calculation. The wireless mic frequency assignment problem only uses 6 types of IM products so this doesn't scale well.
    b. For task ISPC, there are too few types of frequency conflict to even make full use of all cores.
3. Parallelism per frequency range
    a. Performs very poorly with sparse mics as there is a large number of memory accesses, although they are more contiguous.
    b. This doesn't quite scale with the problem – the frequency range tends to stay fixed and the number of mics increases.
    c. This type of parallelism is good for the optimize algorithm.

Sequential behavior is still needed in the frequency assignment problem as it benefits from synchronization. All mics affects all other mics.

4.2 Infrastructure

The importance of synchronization and the non-contiguous properties of IM products makes ISPC not a great choice for the problem.

The strengths of ISPC are:

1. The structure of the analyze algorithm maps very well to ISPC foreach.
2. The behavior of each parallel worker is almost identical so utilization of the SIMD vector is supposed to be high.

The much better choice would be OpenMP.

1. OpenMP supports sequential behavior (only use a single thread) for optimize's sequential algorithm but can branch out into many parallel threads.
2. OpenMP has much better synchronization primitives than ISPC (nonexistent). This could be used to update the IM products caused by each mic and possibly even make the algorithm less sequential.
3. OpenMP is also shared memory but uses threads which can randomly access their own local variables without the gather penalty. This is good for the random access read only behavior of the spectrum state.

The other parallel libraries we used wouldn't work as well. CUDA requires very high parallelism, such as pixels of an image, that the frequency assignment problem doesn't quite reach. MPI could work although the number of IM products that need to be communicated across all workers each iteration could get excessive.

4.3 Next Steps

There were several uncertain conclusions from the data, so it would be helpful to address them first as it may uncover flaws in the program design.

Otherwise, the largest source of improvement seems to be rewriting the program in OpenMP, since many of the issues can be resolved.

4.4 Other

Project website: https://shannoding.github.io/15418-wireless-frequency

Github: https://github.com/shannoding/15418-wireless-frequency

REFERENCES

"Intermodulation With Wireless Microphones"
https://drewbrashler.com/2016/intermodulation-with-wireless-microphones/

"What is intermodulation?"
https://service.shure.com/s/article/what-is-intermodulation?language=en_US

"Finding, Solving, and Preventing Intermodulation Problems"
https://www.softwright.com/faq/support/intermod_finding_solving.html

"Understanding Intermodulation Distortion in Broadband Signals"
https://resources.system-analysis.cadence.com/blog/msa2021-understanding-intermodulation-distortion-in-broadband-signals

"Frequency planning and ramifications of coloring" in Discussiones Mathematicae Graph Theory 22 (2002) 51–88
https://bibliotekanauki.pl/articles/743541.pdf

"Understanding the Power of Distributed Coordination for Dynamic Spectrum Management"
https://people.cs.uchicago.edu/~htzheng/publications/pdfs/lb-monet.pdf