Summary

In 18th Century European classical music, there are various types of constraints to follow when it comes to composing music. For example, certain chords (group of notes) cannot directly follow each other, and certain notes are not allowed in some situations. There are also more simple, hard constraints, such as ensuring that the voice lines of each singers' part are not too high nor too low. In this project, we aimed to automate this process of writing four-part music that is "correct" subject to these constraints. Given an input of melody line, our program will harmonize the three other parts, and output a valid four-part music. Since melody lines are often quite long, the search space size also grows larger, leading to longer computation time for sequential algorithms. Therefore, we have also implemented a few parallel algorithms using OpenMP, and will be comparing the results between the sequential and parallel algorithms.

Background

Data Structures

One of the biggest challenges in this project was to determine a good representation of all the various notes, voicings, and chords. An ideal representation would allow us to convert back into MIDI format, add additional constraints, and add additional valid chords with relative ease. More importantly, the representation must allow for algorithms that can be easily parallelized, as that is one of the main goals of this project. Note that to ensure best cache performance, we made each datatype used be as small as possible. This means more of our data can fit within one cache lines, improving cache performance.

There are three main data structures that are used in our project.

1. Note

Notes are the most basic data structure in music. A note corresponds to a particular pitch. One universal way of representing notes is to use midi numbers, which is a one-to-one mapping from musical notes to integers. This is a simple representation, but is lacking in many aspects. For example, most of the constraints do not directly relate to a pitch in particular, but instead, is related to certain movements from one pitch to another. If we solely used midi numbers to represent notes, many repeated calculations will have to be made in comparing two notes (or pitches) to decide whether or not the constraints are satisfied. As a result, we have chosen to represent notes as a “relative data structure” i.e. notes are stored as the relative distance from a particular starting point (we will omit details regarding this, as it is not necessary to understand the algorithm and is too music-related). Specifically, notes are defined by a tuple of scaleDegree and relativeOctave (which we call *Note*). Scale degrees represent a pitch, and is 1-indexed. The octave represents how high the pitch is, and consists of 7 scale degrees, which means that the next note after (6, 1) is (1, 2).

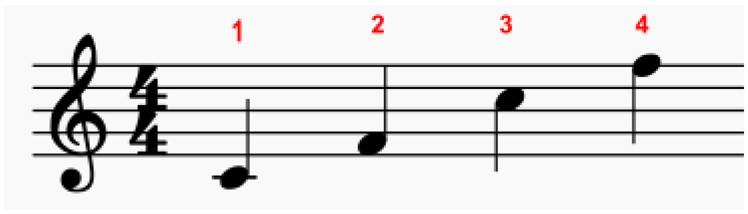

Figure 1: Basic notes on a musical staff

Figure 1 shows four ascending notes. In our representation, if we decided that the first note is the starting point, then it would be represented as (1, 0). The second note is three scale degrees higher, so it would be (4, 0). The third and fourth note have the same pitch as the first two, but are both on a higher octave, so they would be (1, 1) and (4, 1), respectively.

In our algorithm, we use both this relative representation (*Note*), as well as the actual midi numbers of the notes. This is because certain constraints, such as ensuring that the pitch is within a singable range, require us to use a more absolute representation of the notes. Additionally, we have to take in inputs as the MIDI format, and output as a MIDI format.

2. Voicing

Since we are writing four-part music, we decided on a data structure that would represent all the notes that are to be played at once. In figure 2, we have a standard four-part harmony music writing. In music, the horizontal axis represents the time or the rhythm, and essentially, all the notes in the red rectangle are supposed to be played at once.

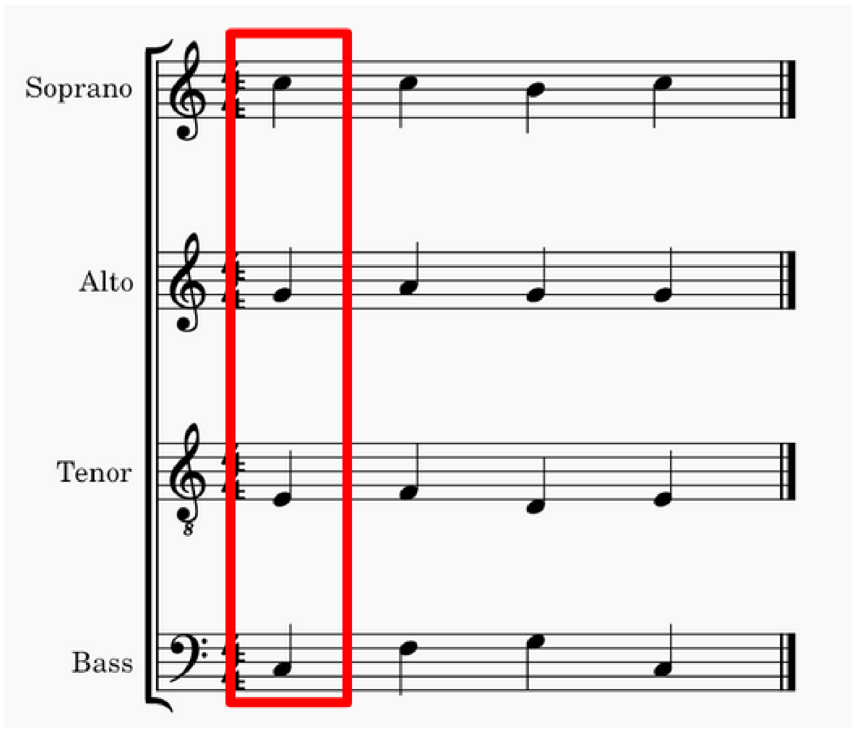

Figure 2: Four-part harmony. The red box shows one voicing.

This is represented by the *Voicing* data structure in our program, which is simply a *Note* array of size 4, where each index into an array is the note of one part. For example, index 0 represents the Soprano's *Note*.

The final output of our program's solver function is an array of *Voicing*, which is then converted into MIDI so that we can listen to it.

2. Chord

In music, chords are group of notes that form a particular sound. Many chords are often used together to achieve a certain effect. This is called a "Chord Progression". In 18th Century European Classical music, there are certain Chord Progressions that sound so bad, they made it "illegal" to use in part writing. This is one type of constraint when it comes to harmonizing the melody line. Additionally, notes in one *Voicing* must exclusively be notes that are in a "valid chord". Essentially, this means that there are certain chords that can be used, and our constraints have to ensure that all the notes in a *Voicing* must use notes from one particular chord (for example, you cannot use two notes from one chord, and two from another. All four will have to be notes of a chord).

In our program, we currently only support three note chords. Therefore, we represent a *Chord* as an array of three notes, with a variety of functions that checks the validity of an input *Voicing* against the notes of one *Chord*.

Representing Constraints

Here are all the different types of constraints in our program, and how we represent them

1. Voicing Invariants

These constraints relate to invariants that must be true within **one**, individual voicing.

- the absolute pitches corresponding to notes within a voicing must be sorted from high to low according to soprano, alto, tenor, bass order
- the distance of absolute pitches between each part must not be too great. For example, the distance between soprano and alto notes must not be greater than 7
- the absolute pitches corresponding to each notes within a voicing must be within a suitable range for that particular part e.g. soprano's notes must not be too low

The first two constraints listed here can be simply checked by comparing the *Notes*'s scale degree and relative octave. The last constraint regarding valid ranges require converting the *Note* (relative representation) back to the midi number, and checking that the pitch is not too high or too low.

2. Inter-voicing Invariants

These constraints relate to invariants that must be true between **two** consecutive voicings.

- "parallel fifths" - there cannot be a musical interval of a fifth between the same two voices in consecutive voicings
- "parallel octaves" - there cannot be a musical interval of an eighth between the same two voices in consecutive voicings

These constraints are verified by looping through the two *Voicing*s and finding the difference between the two, then asserting that the interval difference is not five or eight.

3. Voicing Constraints Relative to a Chord

These constraints ensure that the *Voicing* generated by our program is correct relative to the chord specified.

- a chord voicing must contain only notes from the chord (e.g. a IV chord voicing must only contain notes with degrees ^4, ^6, and ^2)
- the bass scale degree of the voicing must be equivalent to the first note of the chord
- the second note of the chord must only appear **exactly once** in the chord voicing
- The third note of the chord must appear **at least once** in the chord voicing

All of these constraints are checked as part of the *Chord* class methods. The *Voicing* is passed in as an input, and the various checks are performed.

4. Inter-Chord Constraints

These constraints are checked between two chords. They ensure that the chord progression is valid.

- "Retrogression" - V chord cannot immediately precede a IV chord
- No consecutive chords of the same type. For example, a IV chord cannot immediately precede a IV chord.
    - We would like to note that this is actually allowed, but imposing this constraint limits the search space, and allows for more variety.
    - Ideally, this should be a "soft constraint".

These constraints are also checked as part of the *Chord* class methods. They are checked when a valid chord progression is trying to be generated.

Algorithms

Our sequential algorithm is as follows:

1. Read in MIDI melody line and convert to a vector of *Note*s
2. Read in the key (musical key)
3. Given the vector of *Note*s, generate all possible, valid chord progressions (that satisfies inter-chord constraints)
4. For each chord progression, generate all *Voicing* within a valid range, and check for other constraints
5. Output valid *Voicing*s as MIDI

There are a number of expensive computations in the 3rd and 4th step of the algorithm. This is because there are a lot of repeated, simple computations, which would immediately benefit from simple parallelism.

To parallelize the solver itself in the 4th step, however, is not so trivial. This is because to generate a sequence of valid *Voicing*, there are inter-voicing constraints, which will require some communication if the computations were to be split up between various threads.

Approaches to Parallelism

We decided to use OpenMP for parallelizing our algorithm. This is due to a number of reasons. We believe that a simple, starter parallel implementation will be easiest to implement with OpenMP. Additionally, a more sophisticated parallel implementation would still be possible using OpenMP's tasks. Finally, we feel like we understood how to use the OpenMP library best, as we spent a considerable amount of time understanding it in Assignment 3.

1. Chord Progression Parallel Approach

Currently, our algorithm tries to generate a valid voicing for all the valid chord progressions returned in step 3. A very simple, yet effective way of parallelizing the algorithm would be to divide the chord progressions up between many threads. We chose a dynamic scheduling approach as each chord progression will likely take similar amount of computation time, and so this will lead to a more balanced workload between different threads.

This approach is so effective, because all the valid chord progressions are of the same length, and even though some of them may not yield any valid voicings, a comparable amount of computation will still have to be done for constraint checking.

However, this implementation is not exactly ideal for many reasons. We don't necessarily want to generate all possible voicings for each chord progression in the future. We may want to add soft constraints on the chord progressions returned before attempting to find valid voicings, which may greatly limit the number of chord progressions. Additionally, many melody lines may only yield very few valid chord progressions, which will then limit the amount of parallelism possible. If only two valid chord progressions exist, then there is no point in using 16 threads to parallelize this step.

2. Solver Parallel Approach

One possible implementation that will not suffer from this problem is a solver-parallel approach. This means that instead of parallelizing the calls to the solve function (once per chord progression), we can parallelize the solve function itself. This can be done by with a parallel divide-and-conquer algorithm, where we solve each half in parallel, before merging the results together. This implementation is likely to perform better for melody lines with few valid chord progressions, and will also work if we wanted to provide the solver with a specific chord progression.

Results

Much like in the previous assignments, the performance in our program will be measured by the computation time. The results presented here will attempt to describe various aspects of the problem, including the relation between input size and valid solution size, input size and computation time, and number of threads and computation time. Unless specified otherwise, the results will be from running the chord progression parallel approach on GHC68.

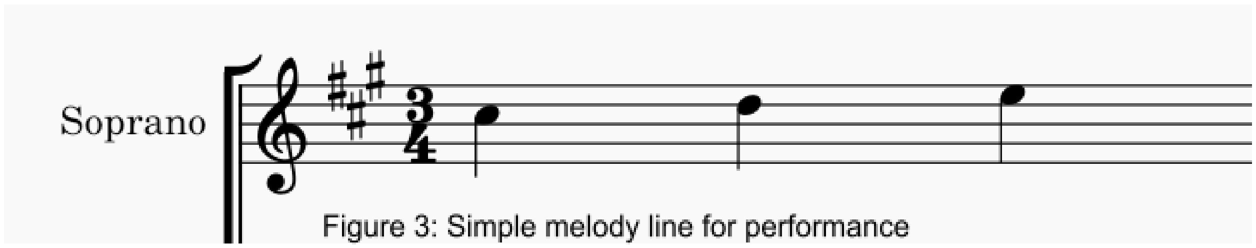

Figure 3: Simple melody line for performance evaluation.

Figure 3 shows a simple three-note melody line. This will be the base of what is used for our various inputs. To increase the input size, we will simply duplicate the same melody line multiple times. This means that our input sizes will consist of multiples of 3 (3 notes, 6 notes, 9 notes etc.). We chose to do it this way because adding a random note could drastically change the number of valid solutions as a result of how the constraints are defined in this problem.

<u>1. Input Size and Valid Solution Size</u>

First, we will show how an increase in the input size changes the number of valid chord progressions. Then, we will show how an increase in the input size changes the number of valid solutions. This will hopefully provide a clear picture on why computation time changes as it does as the input size increases. Note that the results for this does not depend on the implementation, as the valid solution size for each implementation should still remain the same.

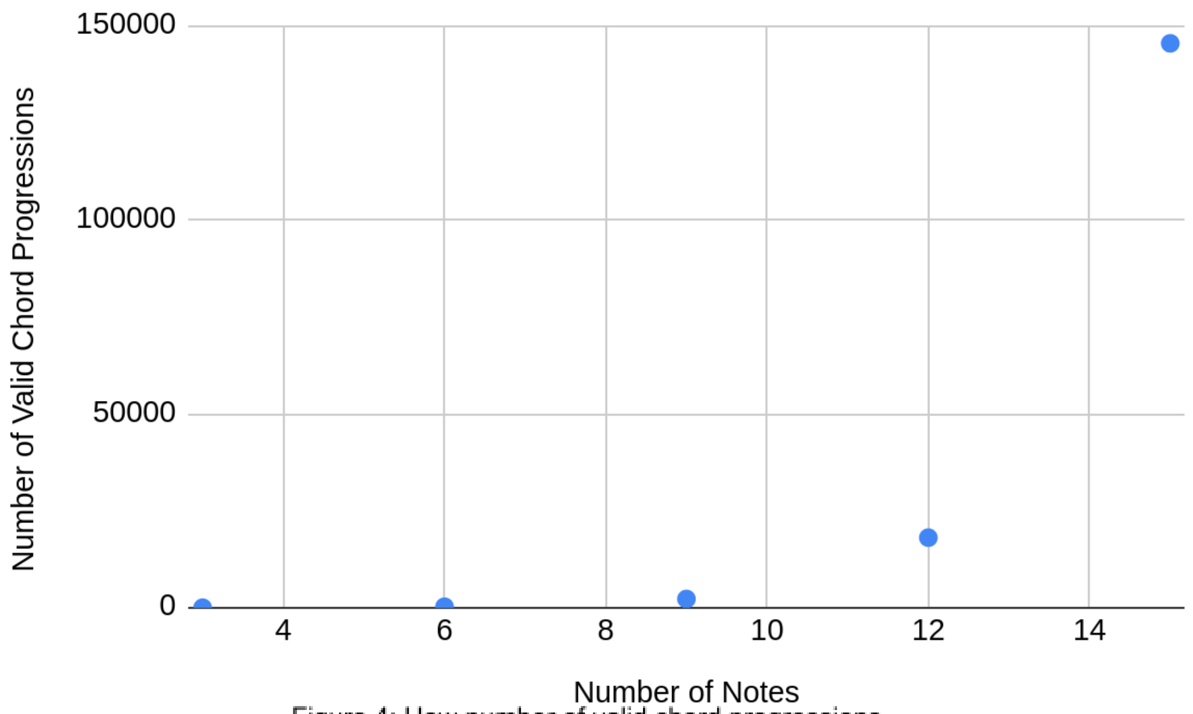

Figure 4: How number of valid chord progressions grows as the input size increases.

Figure 4 shows that the number of valid chord progressions grows exponentially as the input size increases, and Figure 5 shows a similar trend for the number of valid solutions. Therefore, we would also expect to see an exponential growth in the computation time as the input size increases. This result also illustrates how effective the chord progression parallel approach can be, although a lot of usual melody lines will not allow for this many valid chord progressions.

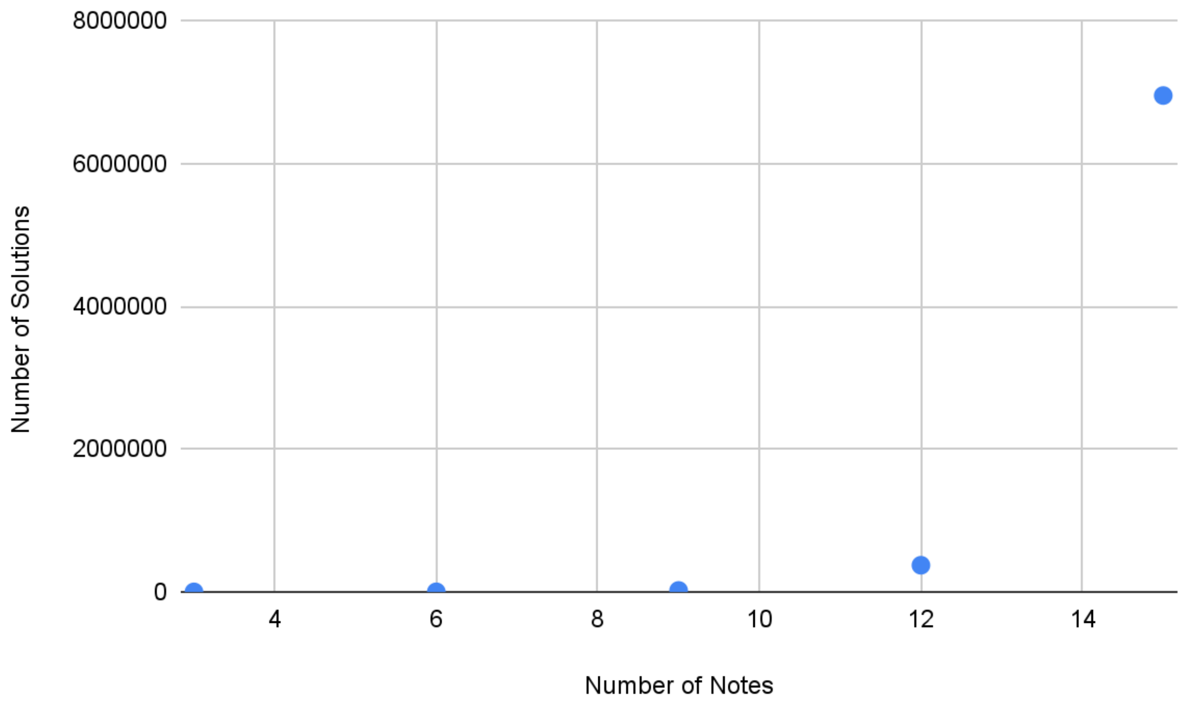

Figure 5: How valid solution size grows as the input size increases.

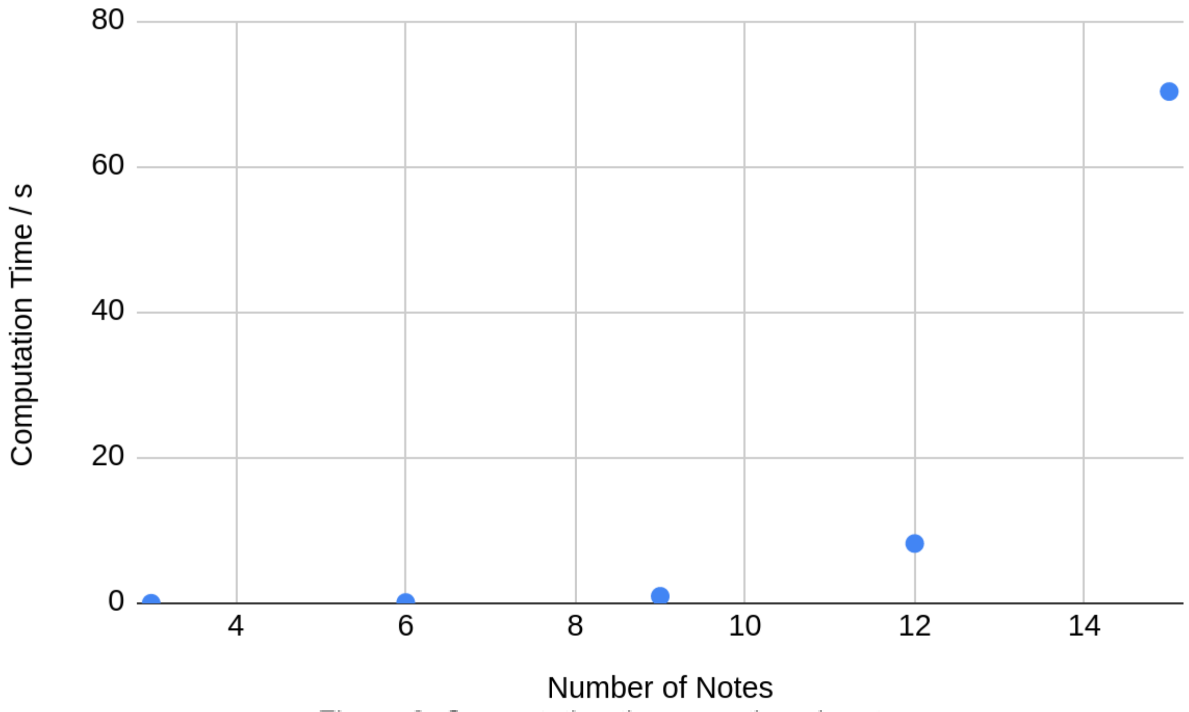

Figure 6: Computation time growth as input size increases. Using 16 threads.

2. Input Size and Computation Time

As expected, the computation time grows exponentially as the input size increases. This is due to the exponential increase in number of valid chord progressions.The implementation used to produce this result is the Chord Progression Parallel implementation (on GHC68), which would likely lead to the lowest computation time just because of the sheer number of valid chord progressions at higher number of notes.

3. Number of Threads and Computation Time for Chord Progression Parallel Approach

For this result, we evaluated the performance of the Chord Progression Parallel approach on the GHC machines and the PSC machines. This is because the GHC machines stopped gaining speed-ups after 64 cores (as seen in Figure 7), which we believe to be due to resource limitations.

The PSC machines gave us the expected results, achieving near ideal speed-up for each increase in number of threads until 64 threads, where we believe communication costs were just too high to get ideal-speedups at that point. This can be seen in Figure 8, where the speed-up gain slows down after 32 threads.

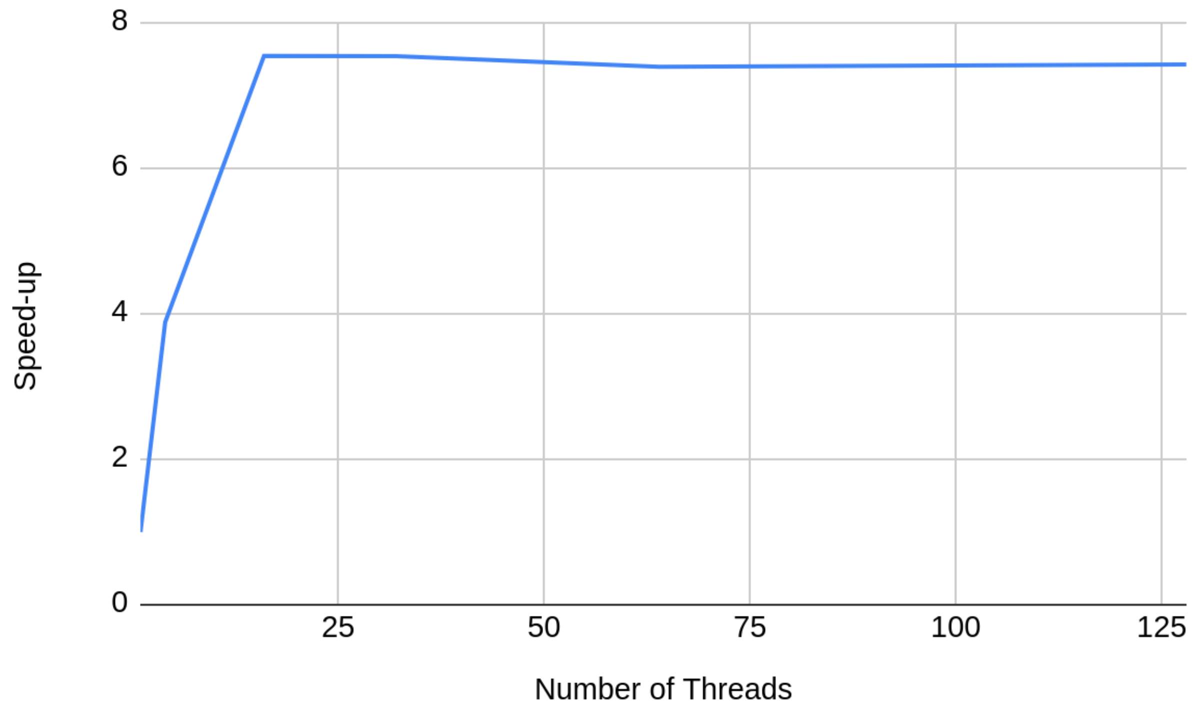

Figure 7: Speed-up as number of threads increase. Evaluated on GHC68.

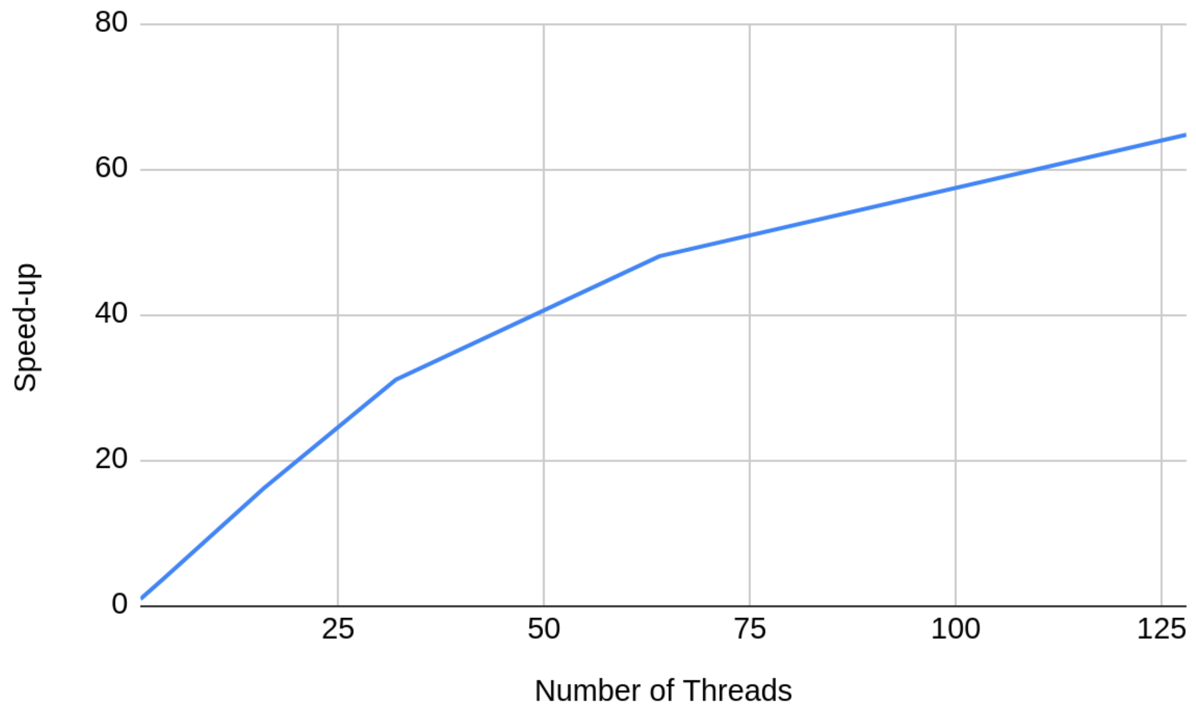

Figure 8: Speed-up as number of threads increase. Evaluated on PSC machine's RM partition.

All of these near-ideal speed-ups from this implementation are only achievable due to the fact that the input melody line allows for many chord progressions. Figure 9 is one melody line that only has 32 possible chord progressions.

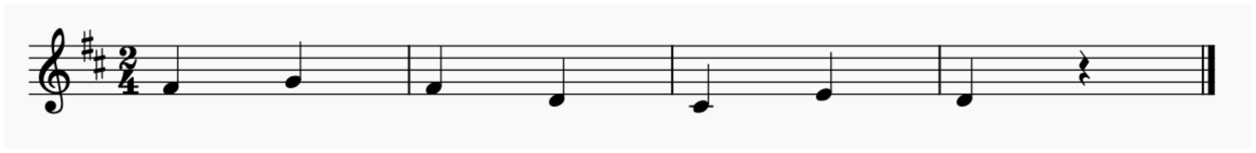

Figure 9: Simple melody line with few valid chord progressions.

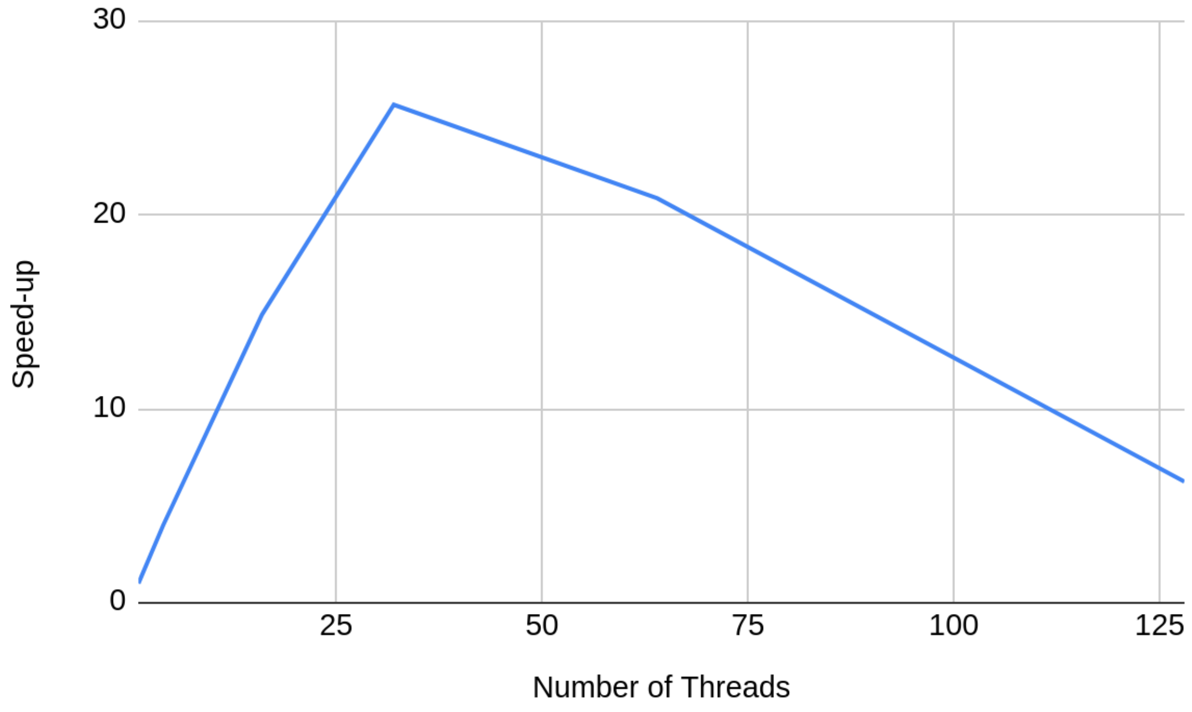

Figure 10: Speed-up graph for melody line with fewer valid chord progressions (PSC RM).

From Figure 10, we can see that the speed-up drops off significantly after a certain point due to the low number of valid chord progressions. The near ideal speed-up around 32 cores is due to the melody line having 32 valid progressions.

Furthermore, in many cases, the number of “valid” chord progressions is actually highly inflated. In the same melody above, only 6 of the “valid” 32 progressions actually have valid voicing solutions associated with them. This is due to the relative lax constraints on chord progressions. Many of the inter-voicing constraints, such as rejecting parallel fifths and octaves, actually imply invalidity for certain chord progressions. With further iteration, it should be possible to filter these false positive chord progressions—so it is likely the number of “valid” chord progressions will be not nearly enough, leading to suboptimal performances with this implementation.

4. Number of Threads and Computation Time for Solver Parallel Approach

Figure 11 shows the computation time for each implementation when the number of threads exceed the number of valid chord progressions. Ideally, we would be able to show that a solver parallel approach would do better. We believe that the reason why our implementation is currently quite slow is due to our inefficient use of vectors. Currently, we are copying many vectors every single recursive call to the solver, leading to long computation times.

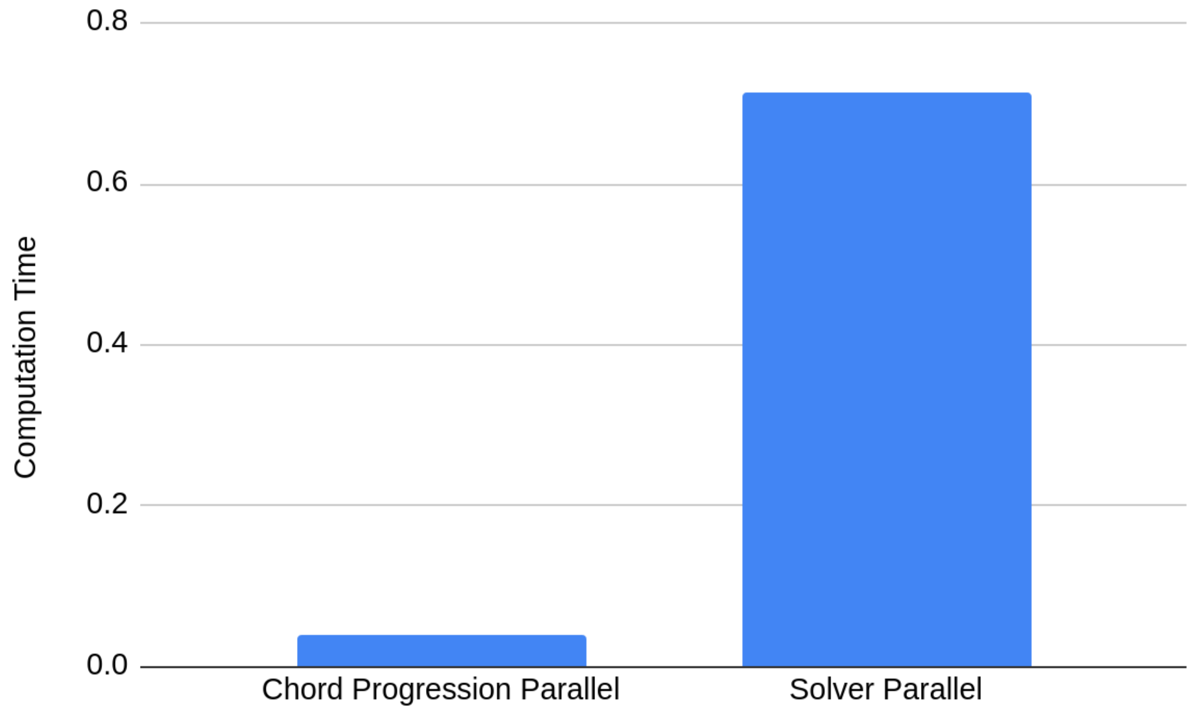

Figure 11: Computation time comparison between two implementations at 128 threads (PSC RM).

Conclusion

Although there has been some prior work on this, our unique contribution to this particular area is in the way we represent constraints and our approach to the solver algorithm. Previous work in this area uses techniques such as genetic algorithms or backtracking to generate valid solutions. Our approach uses a generate-and-test algorithm and attempts to completely exhaust the search space to generate all possible solutions. The “test” component of the algorithm is of particular interest because it suggests applications to autograders. Given a potential solution to the harmonization problem, our approach could be used to determine if the given solution is valid, and additionally, with how our constraints are represented, the algorithm could also be used to determine exactly where a solution fails – providing valuable feedback in the process.

References

https://www.researchgate.net/publication/221399337_Valued_Constraint_Satisfaction_Problems_Applied_to_Functional_Harmony

https://www.andrew.cmu.edu/user/johnito/music_theory/harmony1and2/HarmMain.html

https://midifile.sapp.org/

Work by each Student

Report - Bas

Video - Jonny

Everything else was done together. The credit should be 50%-50%.