StarFinder: Identifying Constellations from a Subset of Stars

Michelle Tang (mt3486), Stella Park (shp2147), Yumeng Bai (yb2542)

December 21, 2023

1 Introduction

1.1 Problem Statement

The foundation of this project aims to develop an efficient and parallelized constellation finder program in Haskell, capitalizing on the language's inherent strengths in concurrent and parallel programming. Our system takes a collection of 3D coordinates (chosen by the user) as input, and deduces the constellation to which the set of points belongs using a sophisticated pattern recognition algorithm.

1.2 Use Cases

Constellation-matching software, which identifies constellations based on a set of stars selected by a user, can have several important use cases across different fields:

- Astronomy Education: Our program can be a powerful tool and aid in teaching and learning astronomy. It can help students and astronomy enthusiasts learn about different constellations and their positioning in the night sky, and help identify unknown constellations based on location.
- Stargazing and Amateur Astronomy: Amateur astronomers and stargazing enthusiasts can use our tool to enhance their night-sky viewing experience. It can help them identify constellations quickly and learn more about the stars they are viewing by providing the program with the locations of the stars they're viewing.

- Astronomical Research: Researchers can use the program to quickly identify areas of the sky for their studies. It can aid in cataloging star patterns and in the study of the movement and evolution of constellations over time (of course, this assumes a more sophisticated future iteration of our program with more comprehensive celestial data).
- Augmented Reality (AR) and Virtual Reality (VR) Applications: In AR and VR environments, such software could provide an immersive educational experience, allowing users to explore the night sky in a simulated environment.
- Astrophotography: Astrophotographers can use the software to plan their shoots, identify constellations in their photographs, and provide educational content alongside their images.

1.3 Literature Review

There are several preexisting mobile apps and softwares that can be considered "state-of-the-art" techniques that tackle the same problem as our project. One such app is **Stellarium**, a popular open-source planetarium software. Stellarium is used by both amateurs and professionals for sky observing. It provides a realistic 3D sky with constellations, planets, and stars, and allows users to identify and learn about different celestial objects. Another app that we tested during our literature review was **Star Chart**, which offers an augmented reality experience, allowing users to point their mobile device at the sky and see a map of the stars and constellations overhead.

1.4 Proposed Approach

The core strategy involves several key components: first, the design will prioritize parallelizing the computationally intensive tasks involved in parsing the input dots against the constellation database. Leveraging Haskell's robust concurrency primitives and functional nature, the program will distribute workloads across multiple cores or processors to boost performance. Second, the implementation of rotation algorithms will accommodate various orientations of the input dots. By comparing rotated versions against the constellation database, pattern recognition techniques will be employed to identify the closest matches. Additionally, efficient management of the constellation database through optimized data structures like balanced trees

or hash maps will streamline lookup processes and enhance overall performance.

- The input is given as a set of coordinates, which are "stars."
- The reference database is a collection of constellations in coordinates, provided by [].
- The input set of coordinates can be rotated to be compared with the database.
- The input set can be a combination of subset of stars coming from more than one constellations.
- The output is a list of constellations that consist the input set. There can be more than one list as an output.

2 Approach Overview

2.1 Fingerprint Pattern Match

The goal of the project is to create a program to which a user can input coordinates of a set of stars and be able to identify a constellation from it if there is any match. The input set of stars may be translated, distorted, or rotated from the match in the database, which makes the naive coordinate match not a good algorithm for this problem. Fingerprint pattern match is commonly used to tackle the issue.

Traditionally, fingerprints are depicted as sets of features termed minutiae. In our context, we liken a fingerprint to a constellation and its minutiae to stars. Matching algorithms compute a score reflecting the likeness between the candidate minutiae set, $M = m_1, ..., m_l$, and the template minutiae set, $M' = m'_1, ..., m'_{l'}$. The challenge lies in calculating such similarity scores, influenced by various experimental parameters affecting minutiae comparison. Initial hurdles involve basic transformations like translation and rotation, yet minutiae can vanish (occlusion), emerge unexpectedly (noise), or undergo nonlinear distortion. Many matching algorithms attempt to align M' with M through translation and rotation, tackling nonlinear transformations using diverse heuristics. Bringer and Despiegel (BD)[2] address this challenge by defining "vicinity." This article introduces a Haskell implementation of BD's algorithm with parallelization.

2.2 Data Acquisition and Preprocessing

In order to build a constellation finder, the first step is to develop a comprehensive database of constellations and their corresponding celestial bodies, wherein each celestial body within each constellation is identified using a set of three-dimensional Cartesian coordinates (x, y, z). The pipeline for data acquisition and preprocessing includes several key phases: data collection, data preprocessing and parsing, and conversion of astronomical coordinates. Each of these steps are described in more detail below:

2.2.1 Data Collection

To begin the construction of our constellation database, we first searched for a reliable astronomical data source containing the celestial information that was necessary for our algorithm. Unfortunately, after surveying several astronomical catalogues, it became apparent that most preexisting intergalactic datasets (e.g., SIMBAD, Gaia) do not store the positional data of specific constellations–rather, each archive is an enormous repository of stars, exoplanets, galaxies, and other celestial objects, and without the necessary background knowledge and astronomy expertise, it would have been extremely difficult to correctly sort and group all the celestial data by constellation. Ultimately, we identified Wikipedia as a primary data source due to its extensive and publicly accessible repository of astronomical data. Wikipedia specifically provides detailed pages for each of the main 88 constellations, where each page consists of list of the constellation's constituent stars and their celestial coordinates (right ascension and declination) and distance from Earth (in light years), among other parameters.

Instead of manually saving the data from Wikipedia for all 88 constellations, we automated the process of data collection by developing a web scraper script using libraries such as `requests` for handling HTTP requests and `BeautifulSoup` for parsing HTML content. The script systematically visits each constellation's Wikipedia page and extracted relevant data, including star names and their positional data, including their celestial coordinates (right ascension and declination) and distance from Earth.

The format of the JSON file generated from this initial data scrape is:

```
{
    "Constellation_1": {
        "Star_1": [
            <right_ascension>,
            <declination>,
            <distance
```

```
        ], ...
    }, ...
}
```

Where the right ascension is given in terms of hours, minutes, and seconds; the declination is given in terms of degrees, minutes, and seconds, and the distance from Earth is given as the number of light years.

Below is a snippet of the JSON file generated from the initial scrape:

```
{
    "Andromeda": {
        "Alpheratz": [
            "00h08m23.17s",
            "+29\u00b0\u00a005\u2032\u00a027.0\u2033",
            "97"
        ],
        "HD 14622": [
            "02h22m50.36s",
            "+41\u00b0\u00a023\u2032\u00a047.5\u2033",
            "154"
        ], ...
    }, ...
     "Canis Major": {
        "SiriusA, B": [
            "06h45m09.25s",
            "\u221216\u00b0\u00a042\u2032\u00a047.3\u2033",
            "8.6"
        ], ...
    }, ...
}
```

2.2.2 Preprocessing and Parsing

From the JSON file snippet provided above, it's clearly apparent that the constellation data we collected using our web scraper script is muddled with irregular formatting and Unicode characters. Given these data inconsistencies, we implemented parsing logic to standardize the data format. This specifically included handling Unicode symbols for right ascension (RA) and declination (Dec) values to separate each string into individual numerical parameters, allowing us to converting our constellation data into a uniform representation of hours, minutes, and seconds for RA, and degrees, minutes, and seconds (with appropriate signs) for Dec.

Below is a snippet of the resulting JSON after parsing the initial data:

```
{
    "Andromeda": {
```

```
        "Alpheratz": {
            "ra_hours": 0.0,
            "ra_minutes": 8.0,
            "ra_seconds": 23.17,
            "dec_degrees": 29.0,
            "dec_minutes": 5.0,
            "dec_seconds": 27.0,
            "lr": 97.0
        }, ...
    }, ...
    "Canis Major": {
        "SiriusA, B": {
            "ra_hours": 6.0,
            "ra_minutes": 45.0,
            "ra_seconds": 9.25,
            "dec_degrees": -16.0,
            "dec_minutes": 42.0,
            "dec_seconds": 47.3,
            "lr": 8.6
        }, ...
    }, ...
}
```

2.2.3 Conversion to Cartesian

The RA and Dec positional coordinates provide the position of stars on the celestial sphere. While useful in an astronomical context, for the purposes of our pattern recognition algorithm, it's much more straightforward to calculate distance in a Cartesian space. Thus, to utilize this data in three-dimensional space analysis, we converted the RA and Dec of each star into Cartesian coordinates (x, y, z). The conversion from celestial to Cartesian coordinates involved astronomical calculations, taking into account the distance of each star from Earth.

The conversion equations are provided below, where α = RA and δ = Dec:

$$A = (15 \cdot \alpha_{\text{hrs}}) + (0.25 \cdot \alpha_{\text{min}}) + (0.004166 \cdot \alpha_{\text{sec}}) \tag{1}$$

$$B = \text{sign}(\delta_{\text{deg}}) \cdot (|(\delta_{\text{deg}}| + \frac{\delta_{\text{min}}}{60} + \frac{\delta_{\text{sec}}}{3600}) \tag{2}$$

$$C = \text{distance in light years} \tag{3}$$

And thus, our 3D Cartesian coordinates are calculated as follows:

$$(X, Y, Z) = \begin{cases} X = C \cos B \cdot \cos(A) \\ Y = C \cos B \cdot \sin(A) \\ Z = C \sin B \end{cases} \tag{4}$$

The final converted dataset, now containing all the stars per constellation with their Cartesian coordinates, was saved in a JSON file. This format was chosen for its balance of readability and structural integrity, and can be easily parsed by Haskell using the built-in `Data.aeson` library. A snippet of our final dataset is provided below:

```
{
    "Andromeda": {
        "Alpheratz": [
            -66.37885122910839,
            -2.4299721007308706,
            -70.68906100022524
        ], ...
    }, ...
    "Canis Major": {
        "SiriusA, B": [
            -1.6123346927655797,
            8.077357938719077,
            -2.473189351488567
        ], ...
    }, ...
}
```

3 Technical Details

3.1 Bringer and Despiegel (BD) Algorithm Implementation

3.1.1 Defining Stars and Vicinities

We implemented Bringer and Despiegel (BD) Algorithm [2, 4] to handle the constellation data and pattern match the user input point constellation to the database. Here, each star is given in a Cartesian coordinate (x, y, z). Instead of comparing each star to star to compare two distinct constellations, sets of stars are organized into groups called "vicinities," denoted as V_i. A vicnity V_i comprises a central star m_i and k-nearest neighboring stars. When each vicinity is compared, each star's relative position is compared in terms of their distance and their relative angle. This comparison is further

discussed in the next section. This approach naturally circumvents issues related to rotation and translation.

3.1.2 Vicinity Comparison

After defining the vicinity, rather than seeing a constellation M as a collection of stars, it can be viewed as a collection of vicinities. If M comprises n stars, it will similarly yield n vicinities. To compare two such vicinities, labeled as A and B (each containing the stars a_i and b_j respectively), we'll compare the a_is to the b_js in pairs. For every comparison between a_i and b_j, a matching score is calculated using a simplified scoring formula.:

$$s(a_i, b_j) = (x_{a_i} - x_{b_j})^2 + (y_{a_i} - y_{b_j})^2 + (z_{a_i} - z_{b_j})^2 + \frac{\sigma_x}{\sigma_y}(\theta_{a_i} - \theta_{b_j})^2 \quad (5)$$

where θ is an angle between the z-axis and a line connecting the star and the origin, σ_x represents the variance of the position (we assume that $\sigma_x = \sigma_y$), and σ_θ is the variance of the angle (orientation). σ values are determined experimentally depends on the database. For a pair of vicinities, since each vicinity contain k stars, a k x k matrix is constructed from the scores calculated from each pair of stars. Then the association score is calculated by Hungarian algorithm [3]. Here, the Hungarian algorithm is commonly used method to find the optimal assignment (or association) between two sets of equal size, minimizing the total cost (here, a score calculated from 5). This association score tells us the association between two vicinities.

3.1.3 Reference Star Set

To compare the input stars with each constellation in the database in consistent manner, it is necessary to give each constellation a metric to be compared with that of the input. If we directly calculate the association score for each vicinities in the input and in each constellation of the database, each vicinity in one constellation become independent, therefore we cannot consider the complete shape of the constellation. Thus, it is necessary to treat each vicinity association score as a "potential" (as in physics) and aggregate all vicinity scores of one constellation together to give one compact metric.

To allow the vicinity association score to be treated as a "potential," we need a reference star set. Here, the reference star set needs to be a discrete constellation (or a random set of stars) that is not included in the database to search for the pattern. In our implementation, "Triangulum" is chosen to be the reference star set and is excluded from the database.

3.1.4 Binary Feature Vector

To obtain a single metric for the input and each constellation in the database, the Binary Feature Vector is constructed. The vector V , of length N , representing the input is computed as follows:

- Extract all the vicinities (denoted F_j) from the input stars.
- For $1 \leq i \leq N$ compute the vicinity association scores S(F_j , R_i) where R_i is i-th vicinity in the reference star set.
- Given a certain threshold t, create the vector V, such that

$$V_i = \begin{cases} 1 & \text{if } \exists j \in \{1, \ldots, n\} \text{ such that } S(F_j, R_i) < t \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

Now this binary feature vector can be viewed as a metric for each set of stars or a constellation, and the comparison between two binary feature vector is done by calculating the Hamming Distance. A constellation from the database that has the shortest Hamming Distance from the input star set is regarded as a match.

3.2 Parallelization Implementation

The core of this project is to calculate the binary feature vector for every constellation in the database and compare it to that of the input. Therefore, the computations for each constellation are not only the same, but also independent of each other. The code snippet for parallelizaion is in the main function as following:

```
1    binaryVectors = parMap rdeepseq (\(n, stars) →
2            (n, createBinaryVector stars refVicinities k sigX
3                        sigTheta t)) templateFingerprints
4
5    hammingDistances = parMap rdeepseq (\(n, templateVector) →
6            (n, hammingDistance templateVector
7                        candidateBinaryVector)) binaryVectors
```

The parallel nature of the code is facilitated by the parMap function from the Control.Parallel.Strategies module. Firstly, parMap is used to dispatch the parallel execution of binary feature vectors. The use of rdeepseq ensures a deep evaluation of the result, allowing parallel execution of the createBinaryVector function for different elements. The function provided

to parMap takes a tuple (n, stars) from the database and computes createBinaryVector. The tuple represents each constellation with its name and the minutae that belong to it. The lamda function helps to map the name of the constellation to its calculated binary feature vector, allowing further computation.

Similarly, we can also calculate the hamming distances between each constellation's bianry feature vector and that of the input on a per constellation basis. The results are a list of tuples, including the name of the constellation and the numerical result, which is going to be sorted and output.

4 Experiments

To further understand parallelization in this program, we aim to investigate the relationship between the number of processing cores and the resulting speedup. By using threadscope, we can acquire data about time usage and compare real data to our expectation. Then, we will make sense of why or why not does the actual speedup follow our theory.

4.1 Setup

Due to the limitation of threadscope, we could not experiment on large inputs, as the eventlog file could be too big for threadscope to open and analyze. Therefore, we chose k = 15, with the stars of Andromeda as input. We run the experiment on an apple M1 chip with 8 cores available. It is worth noting that for apple M1 chip, each core is single-threaded. Therefore, we only ran the experiment up to 8 cores.

4.2 Experiment: parallelization speed up

Using the same input as experiment 2, we used commands

```
ghc -threaded -eventlog -rtsopts --make parse.hs

./parse +RTS -ls -N4
```

to record the time and eventlog for the same program under the same inputs with different number of cores. The results are shown in the table

#cores	time (s)
1	21.08
2	11.39
3	8
4	7.3
5	6.65
6	6.03
7	6.2
8	6.26
16	6.36

Figure 1: runtume vs number of cores

From the graph, we can see that from 1 core to 3 cores, there is an almost linear speed up. From 3 to 6 cores, the speedup is still linear but relatively smaller thn that of the first 3 cores. Then it plateaus until 8 cores.

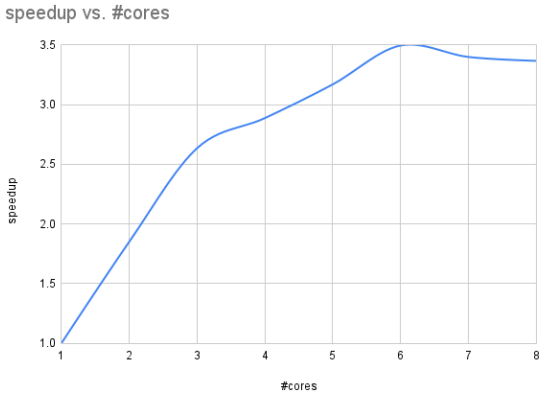

Figure 2: Speedup result

For the first part of the graph, using 2 cores have an almost ideal speedup, meaning that the speedup is almost the same as the number of cores we have.

Then, the speed up slowed down, which is expected in real life scenarios, as pointed out by Amdahl's Law that the bottleneck for speedup lies in the sequential part of the code. Additionally, the speedup plateaus after 6 cores, which actually contradicts our assumption that speed up should only stop after hitting the number of available cores for this machine. After further research, we found out that for the M1 chip we were utilizing, 6 out of the 8 cores are "performance cores" while the other two are "efficiency cores". Efficiency cores are optimized differently from its counterpart, thus creating an asymmetric multiprocessing systems causing different behaviors than expected after 6 cores [1].

5 Conclusions

Throughout this project, we have learned about fingerprint pattern matching, which is widely adopted for its reliability, uniqueness, and ease of use. It plays a vital role in enhancing security and providing seamless user authentication in a variety of applications.

Our exploration of parallelization in Haskell has provided valuable insights into leveraging concurrent processing for improved computational efficiency. In the realm of functional programming, Haskell's purity and laziness present unique opportunities for parallelism. Harnessing these features alongside parallelization strategies opens doors to building efficient and scalable applications. Understanding the nuances of parallel programming in Haskell is crucial for optimizing performance.

Experimentation with different core configurations using the allows us to observe the effects of parallelism on execution time and tailor our approach to achieve optimal results. Reasoning behind the speedup curve gave us a better understanding of differences between symmetric and asymmetric systems, and the variance between ideal and factual parallization.

References

[1] Optimize for apple silicon with performance and efficiency cores.

[2] Despiegel Julian Bringer, Vincent. Binary feature vector fingerprint representation from minutiae vicinities.

[3] Harold Kuhn. The hungarian method for the assignment problem.

[4] Hervé Chabanne Robin Champenois Jérémie Clément1 Houda Ferradi Marc Heinrich Paul Melotti David Naccache Antoine Voizard Thomas Bourgeat, Julien Bringer. New algorithmic approaches to point constellation recognition.

6 Appendix

6.1 Threadscope eventlog output

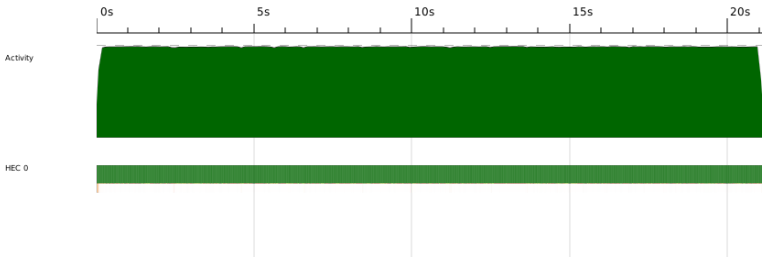

Figure 3: threadscope result for 1 core

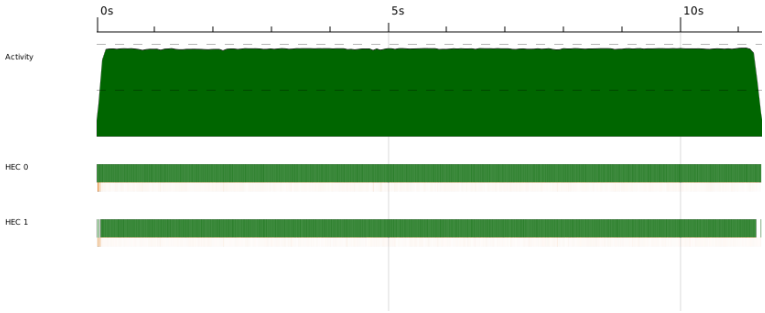

Figure 4: threadscope result for 2 cores

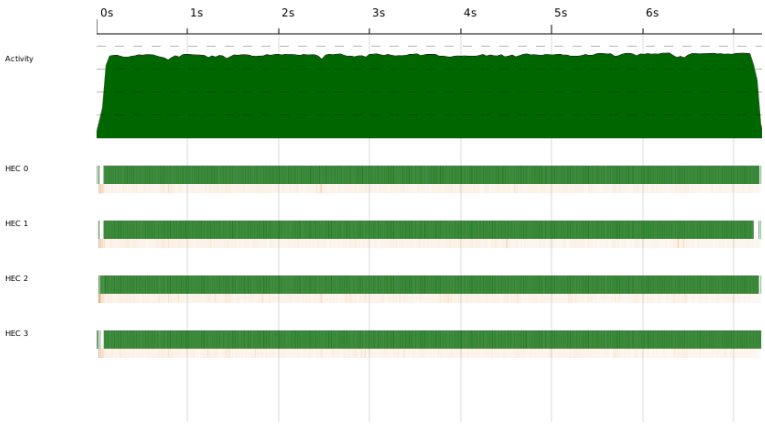

Figure 5: threadscope result for 4 cores

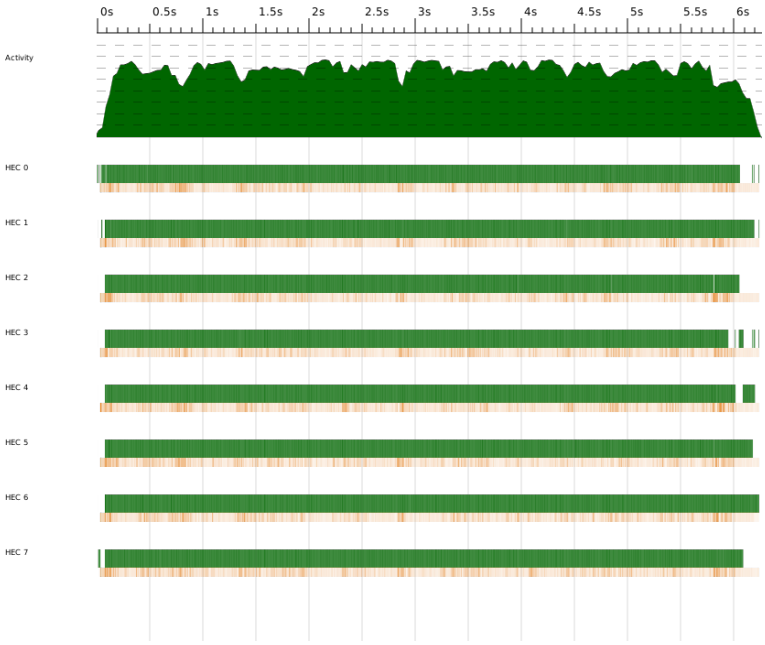

Figure 6: threadscope result for 8 cores