High Performance Data Compilation for Sports Analytics

Written by: Tyler Gourley

1. Abstract

The main goal of this project is to optimize—using parallel techniques—a data compilation algorithm for a Sports Analytics problem. There was much need for an optimized solution since the initial algorithm was incredibly slow. To compile the data for all 330,000+ soccer matches that are contained within our database, we estimate it to take 340 days of constant compute-time on a dedicated machine. Obviously, that is not feasible; in this paper we will show the process of shrinking the time it takes to process that data. The other goal of this project is to ensure that the code is transferrable to other research students that may come after me. Thus, the hardware platform that I am targeting is my laptop (which has multiple CPU processors).

2. Background

We are asking the question: *Can we predict the outcome of soccer matches based on team chemistry?* To answer this, we are compiling upper triangular 11 x 11 NumPy arrays that contain the number of minutes played between every combination of teammates. We will call these 11 x 11 NumPy arrays Team Chemistry Arrays (TCAs). These TCAs will be used as inputs to train a convolutional neural network. TCAs can be represented graphically, as complete weighted graphs. *Figure 2.1* shows an example of how a 4-4-2 formation might be represented as a graph.

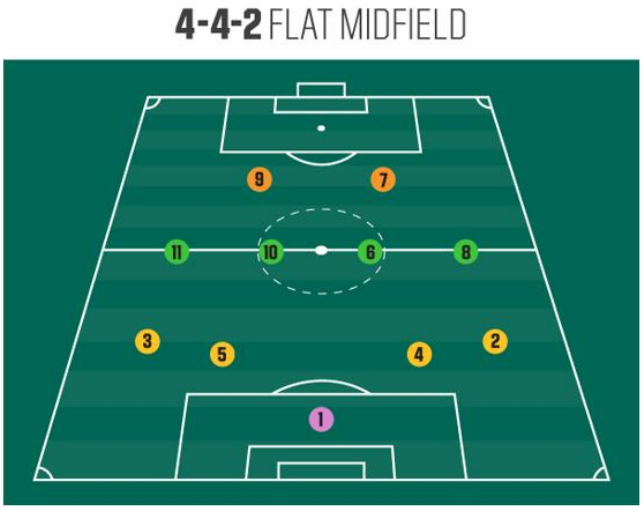

Figure 2.1A: *Example of a 4-4-2 formation*

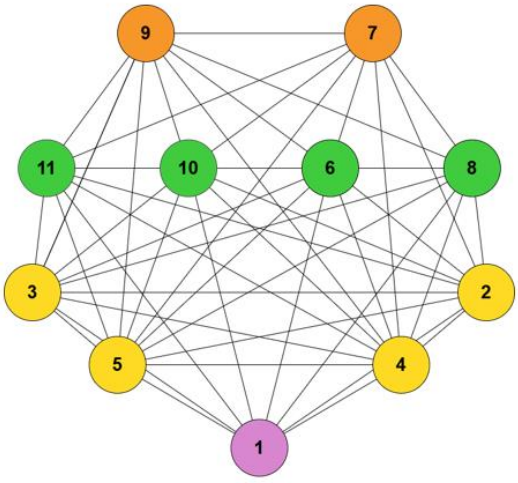

Figure 2.1B: *Complete Graph of a team using a 4-4-2 formation*

In these graphs, each node represents a player, and each edge represents their chemistry (quantified by the number of minutes the two players have played together in their professional careers). *Figure 2.2A* illustrates an example of what a graph might look like for a given TCA (shown in *Figure 2.2B*).

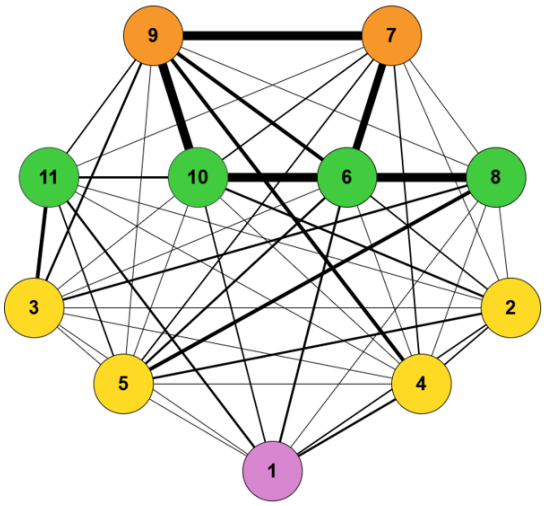

Figure 2.2A: *Graphical representation of a Team Chemistry Array (TCA) using a complete, weighted graph*

0	900	810	900	810	720	810	900	990	900	630
0	0	810	810	810	720	630	810	720	540	360
0	0	0	720	630	990	900	720	810	810	630
0	0	0	0	810	990	900	810	1440	810	630
0	0	0	0	0	990	900	1530	990	810	630
0	0	0	0	0	0	2250	2160	1350	2250	630
0	0	0	0	0	0	0	900	2160	720	630
0	0	0	0	0	0	0	0	990	810	630
0	0	0	0	0	0	0	0	0	2070	630
0	0	0	0	0	0	0	0	0	0	630
0	0	0	0	0	0	0	0	0	0	0

Figure 2.2B*: Upper Triangular NumPy array representing a TCA (cells in blue correspond to large edge weights)*

The graphical representations allow the TCAs to be more easily understood by humans. From this we can start asking questions like: what kind of team chemistry configurations are most advantageous? Are there any matchups that provide a particular advantage for a team, for example Team A having high team chemistry between their forwards and strikers playing against Team B with weak chemistry between their goalie and defenders? Does team chemistry have an observable effect on winning at all?

But in order to answer these questions we need to first create the TCAs, so we can create a training data set to use as an input to a machine learning algorithm. We have access to a database that contains the playing time and lineup data for over 330,000 professional soccer matches over a 23-year span. Each soccer match has 2 NumPy arrays that need to be created—one for each team.

3. Approach

The first step in the optimization process is optimizing the algorithm. The areas where the program spends its time are querying the database, searching the data for matches.

The current serial implementation to create TCAs is unoptimized (see *Figure 3.1A*). It uses a quadruple nested for loop—leading to a time complexity of $O(n^4)$. The bulk of the complexity comes from the inner two for loops (lines 7-12 in *Figure 3.1A*) which search for matches that the two players have played in common. There are also redundant SQL queries. We will use set intersection to optimize this algorithm. Set intersection is an O(n log(n)) algorithm, as compared to a double nested for loop search which is $O(n^2)$. The other major improvement comes from querying the database outside the loop and storing the results from those queries in sets (lines 5-9 in *Figure 3.1B*).

Algorithm 1: Compile Team Chemistry Data

```
1:  matches ← query database (date_range)
2:  for match in matches
3:      for team in match
4:          init team_chem_array
5:          for col in team_chem_array
6:              for row in team_chem_array
7:                  lineup_data1 ← query database (player_id1)
8:                  for p1_data in lineup_data1
9:                      lineup_data2 ← query database (player_id2)
10:                     for p2_data in lineup_data2
11:                         if p1_data.match_id = p2_data.match_id
12:                             team_chem_array[row][col] += 90
13:                         end if
14:                     end for
15:                 end for
16:             end for
17:         end for
18:         save team_chem_array
19:     end for
20: end for
```

Figure 3.1A: *Unoptimized Algorithm to compile TCAs*

Algorithm 2: Compile Team Chemistry Data (Using Sets)

```
1:  matches ← query database (date_range)
2:  for match in matches
3:      for team in match
4:          init team_chem_array
5:          L ← 0
6:          for player_id in team
7:              S ← query database (player_id)
8:              L[player_id] ← S
9:          end for
10:         for row in team_chem_array
11:             for col in team_chem_array
12:                 R ← L[row] ∩ L[col]
13:                 team_chem_array[row][col] += 90 * |R|
14:             end for
15:         end for
16:         save team_chem_array
17:     end for
18: end for
19:
20:
```

Figure 3.1B: *Algorithm that has been optimized to eliminate redundant SQL queries and use set intersection*

We will label the optimized algorithm (Algorithm 2 as shown in *Figure 3.1B*) as our fastest serial implementation, T*(n), to which we will compare our parallel application. We will use the Python Multiprocessing package in our parallel implementation. The problem of compiling TCAs is embarrassingly parallel because creating one TCA has no effect on another. This makes our program an ideal candidate for Python Multiprocessing since there is no communication that needs to happen between the processes. Additionally, I chose not to use Numba to implement parallelism because there is no heavy computation being done in this data compilation project.

In order to make the code parallelized, I chunked the data into tasks (at a match granularity). In essence, each match became a task. Using Python Multiprocessing, I created a task pool. Next, I added each of the tasks (corresponding to matches) to the pool. Finally, I spawned N number of processes, where each process takes a task from the pool, compiles the data and creates the TCA, saves the TCA into the results, and

then returns to the pool to grab the next task. These processes are run independently. This continues until the task pool is completely empty. Importantly, when adding tasks to the task pool, I used the `imap_unordered()` function (see *Figure 3.2* line 131). This is a non-blocking function that also does not restrict a certain order for the resulting TCAs, allowing the processes not to have to wait for other processes to finish before returning.

```
113     # Query matches data from the database
114     matches = select_matches(start_date='2015-01-01', end_date='2024-01-01', limit=NUM_MATCHES)
115
116     # put the matches data (query result) into a pandas dataframe
117     df = pd.DataFrame(matches.fetchall())
118     df.columns = ["match_id", "date", "home_team_id", "away_team_id", "home_team_goal", "away_team_goal"]
119
120     # Divide the input data into chunks to send to the processes
121     chunk_size = 1 # if using a value other than 1, we need to modify our process_chunk function
122     chunks = [df.iloc[i:i + chunk_size] for i in range(0, df.shape[0], chunk_size)]
123     # print (chunks) # Uncomment to verify what the chunks look like
124
125     print(f'beginning parallel compilation')
126     print(f'compiling data for {NUM_MATCHES} matches')
127     print(f'with {NUM_PROCS} processors')
128
129     # Begin Parallelism
130     pool = multiprocessing.Pool(processes = NUM_PROCS) # create task pool
131     results = pool.imap_unordered(process_chunk, chunks)
132     pool.close() # signify that we are not adding any more tasks to the pool
133     pool.join() # blocking, waits for the entire task pool to be dried up
134     # End Parallelism
135
```

Figure 3.2: *Code snippet of the parallel implementation using Python Multiprocessing*

Additionally, it should be noted that in the parallel implementation, there is a section before the processes are created that is run serially. From *Figure 3.2*, the serial section can be seen in line 114, which involves initializing a connection to the database, and performs the initial SQL query to get the data for the matches we want to analyze. Because of this, the parallel optimizations that we implement cannot speed up that section of code—no matter how many processes we throw at it. Therefore, our total speedup will be limited as described by Amdahl's Law (*Equation 3.1*).

$$S(n) = \frac{1}{(1 - P) + \frac{P}{n}}$$

Equation 3.2: *Amdahl's Law*

- ***S(n)*** is the speed-up achieved by using ***n*** cores or threads.
- ***P*** is the fraction of the program that is parallelizable
- ***(1 - P)*** is the fraction of the program that must be executed serially.

4. Results

I benchmarked my parallel implementation and compared it to T*(n). For the benchmark, I ran the test data on a small sample of 200 matches that is representative of the entire 330,000+ matches contained in the database. The timing statistics were taken over the entire program runtime, including the portion of the program that must be computed serially. *Figure 4.1* shows the results of execution time compared to the number of processes. We see decreasing execution times until 6 processes, then the graph flattens out.

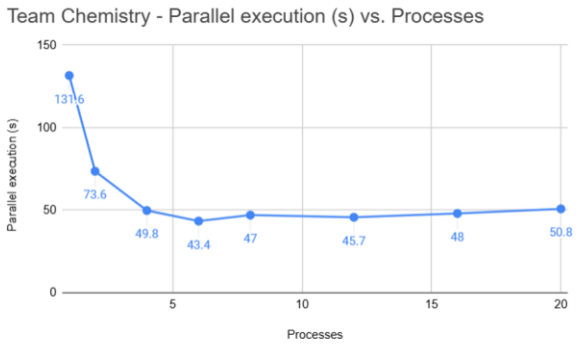

Figure 4.1: *Execution Time vs. Processes*

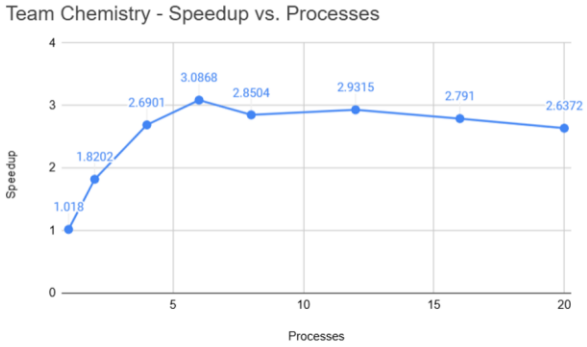

Figure 4.2: *Speedup vs. Processes*

Figure 4.2 shows the results for speedup compared to processes. We can see that the program ran the quickest when using 6 processes. From the results in *Figure 4.2*, we can see that the max speedup was 3.09, at 6 processes. It makes sense that the optimal speedup occurred using 6 processes because the code was run on my laptop that sports an Intel® Core™ i7-9750H Processor, which has 6 physical cores.

Figure 4.3 shows the results for efficiency compared to processes. With 6 processes, we saw an efficiency of 0.514 (see *Figure 4.3*). We would hope the value for efficiency would be much closer to 1.0 meaning we achieved linear speedup. However, there are two major factors that negatively impact efficiency. The first is the overhead that is involved with parallel—namely creating processes, chunking the input data into tasks, and adding those tasks to the task pool. Secondly, we are limited by Amdahl's Law (See *Equation 3.1*). The initial database query must be performed serially before we can spawn parallel processes. With these considerations, we are happy with a 3x speedup and an efficiency of 0.514.

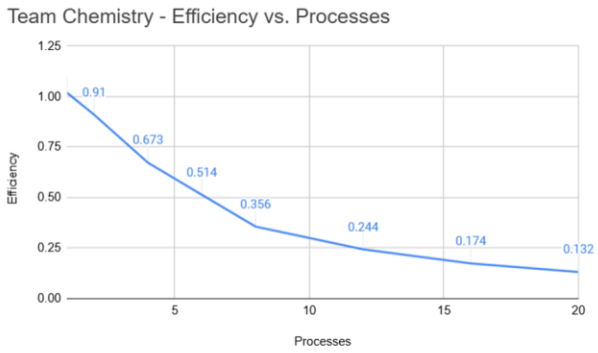

Figure 4.3: *Efficiency vs. Processes*

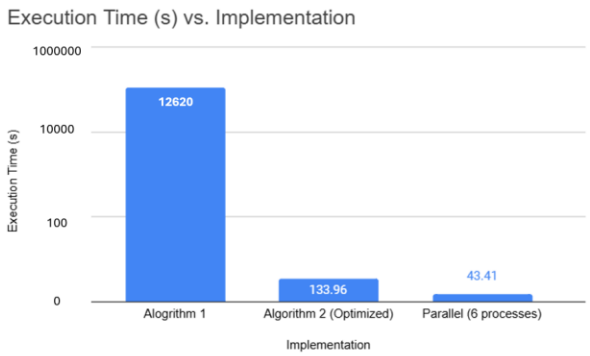

Figure 4.4: *Execution Time vs Implementation*

Figure 4.4 shows program execution time compared to the three implementations that we used. This graph indicates that the bulk of the decrease in execution time came from the initial algorithm optimization—before we even attempted to create a parallelized solution. While the speedup from parallelism was 3.06, the speedup from optimizing the serial implementation of the algorithm was about 94. Combining these 2 optimizations (the algorithm optimization using sets and reduced queries and parallelization) we observed a speedup of 290.

5. Conclusion

Overall, the fastest parallel version was about 290x faster than the original unoptimized algorithm. That brings our estimate to compile the TCAs for all 330,000+ matches from 340 days down to just over 28 hours. This project was a success, because dedicating a laptop or server PC for 1 day is much more feasible than 340 days.

In conclusion, parallelization can only achieve so much speedup, especially when we are limited to a certain target hardware platform with a set number of cores. It is very important to ensure that the algorithm is optimized in its serial state before attempting to parallelize it.