Formula 1 Pit Stop Strategy Simulation

High-Performance Computing Approach to Pit Stop Strategy

Elena Manneh

Fall 2024

1 Introduction

1.1 Formula 1 Race Strategies

Formula 1 race strategies are critical to achieve optimal performance on the track. A **stint** refers to the continuous period during a race that a car runs on a specific set of tires between two pit stops. The choice of tire compounds, their degradation rates, and stint durations significantly impact the outcome of a race.

In Formula 1, there are three main types of tire compounds provided by the official supplier:

- **Soft Tires:** Provide maximum grip but degrade quickly, making them suitable for short, high-performance stints.
- **Medium Tires:** Offer a balance between grip and durability, making them versatile for medium-length stints.
- **Hard Tires:** Designed for durability, with lower grip, suitable for long stints where pit stops need to be minimized.

The degradation rate of a tire is influenced by several factors, including:

- **Track Conditions:** Rough or abrasive tracks accelerate tire wear, while smoother tracks are less demanding on tires.
- **Driving Style:** Aggressive driving, such as late braking and high-speed cornering, increases degradation.
- **Ambient Conditions:** High temperatures exacerbate tire wear due to increased thermal stress.

Pit stops are a critical component of race strategy, as they involve a trade-off between time lost during the stop and the performance gained from fresh tires. A pit stop can cost upwards of 20 seconds, depending on the track and efficiency of the pit crew. Therefore, finding the optimal strategy—balancing stint durations, tire selection, and the number of pit stops—is crucial to minimize total race time and outpace competitors. The timing of a pit stop, relative to competitors and race conditions, can often determine the difference between victory and defeat.

2 Methodology

2.1 Project Workflow

The project workflow was divided into three key phases:

1. **Prerequisites:** Data collection and rate calculations to set up the simulation environment.
2. **Simulation:** Implementation and execution of the race strategy simulation, including parallelization using MPI and OpenMP.
3. **Validation:** Testing and profiling the simulation to ensure correctness and measure performance.

2.2 Prerequisites

Before executing the simulation, the following steps were completed:

2.2.1 Data Collection

Data was collected using the FastF1 library to retrieve historical Formula 1 race telemetry. The `fetch_data.py` script fetched lap times, tire compounds, track information, and weather data for races from the last five years. This data was processed and saved in JSON format. The key steps included:

- Retrieving track information, including the event name, track length, and total laps.
- Normalizing lap times per kilometer for accurate comparisons.
- Organizing data by driver and tire compound for each race.

A sample code snippet for the data collection is shown below:

```
for driver_id, laps in session.laps.groupby("Driver"):
    for compound in laps["Compound"].unique():
        compound_laps = laps[laps["Compound"] == compound]
        for _, lap in compound_laps.iterrows():
            lap_time = lap["LapTime"]
            time_per_km = lap_time.total_seconds() / track_length
```

The collected data provided a foundation for calculating tire degradation rates.

2.2.2 Rate Calculation

The degradation rates and average stint lengths for different tire compounds were calculated using the `rate_calculation.cpp` program. The JSON data from the collection phase was processed to calculate:

- Average degradation rate per lap for each tire compound.
- Average stint lengths, accounting for consecutive laps using the same tire compound.

The results were saved in a `rates.json` file, which was later used in the simulation phase. A sample degradation rate calculation is shown below:

```
double totalDegradation = 0.0;
for (size_t i = 1; i < sortedLaps.size(); ++i) {
    double degradation = sortedLaps[i].second - sortedLaps[i - 1].
        second;
    totalDegradation += degradation;
}
```

The rates were calculated directly from historical data instead of relying on a predefined function of track characteristics such as the number of turns. This decision was made to ensure that the model reflects the observed behavior of tires during real-world races rather than assumptions or simplified approximations. By using actual lap times normalized by track length, the model incorporates variations caused by factors like:

- Tire wear dynamics, which can vary unpredictably between compounds and conditions.
- Real-world track conditions, such as surface abrasiveness and weather influences, which are not easily captured by generalized equations.

Additionally, different degradation rates were chosen for different tracks because tracks have unique characteristics that significantly influence tire performance. For example:

- High-speed circuits like *Monza* (Italian Grand Prix) put greater thermal stress on tires due to prolonged high-speed runs.
- Technical circuits like *Monaco* require frequent acceleration and braking, leading to higher mechanical stress and wear.
- Abrasive tracks like *Bahrain* naturally cause faster tire degradation compared to smoother circuits like *Silverstone*

This track-specific approach ensures that the degradation rates are realistic and tailored to the unique demands of each track, allowing for more accurate simulations of race strategies.

2.3 Simulation

The actual simulation began on the SciNet cluster, where the parallelized race strategy simulator was executed.

2.3.1 Assumptions

The following assumptions were made for the simulation:

- Tire degradation is modeled as linear with a minor quadratic component for long stints.
- A penalty is applied for exceeding the recommended stint length to simulate unsafe tire usage.
- Pit stop penalties vary depending on the stint sequence.

2.3.2 Implementation

The simulation was implemented in `generate_strategies.cpp`, which automates the process of generating and evaluating race strategies. The primary functions of the implementation include:

- **Strategy Generation:** Generates all possible race strategies for a given track by iterating over all tire compounds and feasible stint lengths.
- **Race Simulation:** Calculates the total race time for each strategy, accounting for tire degradation and pit stop penalties.
- **Optimal Strategy Identification:** Compares race times across all strategies to determine the optimal configuration with the shortest total race time.

A key feature of the implementation is the **stint variable**, which is provided as a command-line argument. The stint variable defines the range of laps around the average stint length that can be considered for each tire compound. This gives precise control over the granularity of the generated strategies, allowing the simulation to explore more or fewer combinations depending on computational constraints. For example, a smaller stint variable narrows the range of possible stint lengths, reducing the number of strategies generated, while a larger stint variable broadens the range, enabling a more exhaustive exploration of potential strategies.

The strategy generation process dynamically explores all combinations of tire compounds and stint lengths using a recursive helper function. The following code snippet shows how the stint variable is used to define the range of valid stint lengths for each compound:

```
void generateStrategiesHelper(std::vector<Strategy>& strategies,
    Strategy currentStrategy,
                              const json& trackData, int totalLaps,
                                   int remainingLaps,
                              const std::vector<std::string>&
                                  compounds, int stintVariable) {
    if (remainingLaps == 0) {
        strategies.push_back(currentStrategy); // Base case:
            strategy complete
        return;
    }

    for (const auto& compound : compounds) {
        if (!trackData.contains(compound)) {
            continue; // Skip invalid compounds
        }

        int avgStint = trackData[compound]["Average Stint Length"];

        // Generate valid stint lengths within the range defined by
             the stint variable
        for (int stintLength = std::max(1, avgStint - stintVariable
            );
             stintLength <= avgStint + stintVariable; ++stintLength
                 ) {
            if (stintLength > remainingLaps) {
                stintLength = remainingLaps; // Ensure the final
                    stint fits
            }

            // Create a new strategy with the current stint
            Strategy newStrategy = currentStrategy;
            newStrategy.stints.push_back({compound, stintLength});

            // Recur with updated laps
            generateStrategiesHelper(strategies, newStrategy,
                trackData,
                                     totalLaps, remainingLaps -
                                         stintLength, compounds,
                                         stintVariable);

            if (stintLength == remainingLaps) {
                break; // Stop further exploration if the last
                    stint is added
            }
        }
    }
}
```

This approach ensures that the simulation remains flexible and configurable, allowing users to balance computational cost with strategy precision by adjusting the stint variable as needed. The stint variable is specified when invoking the program as follows:

```
./generate_strategies <track_name> <stint_variable> <total_laps> <
    starting_lap_time>
```

For example:

```
./generate_strategies "Italian Grand Prix" 10 50 90.0
```

In this example, the stint variable is set to 10, meaning that for each tire compound, the stint lengths considered will range from 10 laps below to 10 laps above the average stint length for that compound.

Once the strategies are generated, the simulation evaluates the total race time for each strategy by modeling tire degradation and pit stop penalties. The core simulation function is shown below:

```
double simulateRace(const Strategy& strategy, double
    startingLapTime, const json& trackData) {
    double totalRaceTime = 0.0;
    const double basePitStopPenalty = 25.0; // Pit stop time in
        seconds

    for (size_t i = 0; i < strategy.stints.size(); ++i) {
        const auto& stint = strategy.stints[i];
        double degradationRate = trackData[stint.tyreType]["Average
             Degradation"];
        int maxStintLength = trackData[stint.tyreType]["Average
            Stint Length"];

        // Simulate each lap in the stint
        for (int lap = 0; lap < stint.laps; ++lap) {
            double nonlinearDegradation = degradationRate * (1 +
                0.02 * lap);
            totalRaceTime += startingLapTime + lap *
                nonlinearDegradation;
        }

        // Apply penalty for exceeding recommended stint length
        if (stint.laps > maxStintLength) {
            totalRaceTime += 100.0; // Unsafe usage penalty
        }

        // Add pit stop penalty for all but the last stint
        if (i < strategy.stints.size() - 1) {
            double pitStopPenalty = basePitStopPenalty + 5.0 * (i +
                 1); // Incremental penalty
            totalRaceTime += pitStopPenalty;
        }
    }

    return totalRaceTime;
}
```

2.3.3 Parallelization

To handle the computational complexity of evaluating a large number of strategies, the simulation utilized parallel computing with a hybrid MPI and OpenMP approach.

MPI for Inter-Process Distribution The strategies were distributed across processes using MPI, allowing each process to independently evaluate a subset of strategies. The rank and size of the MPI communicator were used to partition the work as follows:

```
int chunkSize = (numStrategies + size - 1) / size; // Divide
    strategies among processes
int start = rank * chunkSize;
int end = std::min(start + chunkSize, numStrategies);

// Gather local results from all processes
MPI_Gather(localResults.data(), localResults.size(), MPI_DOUBLE,
           globalResults.data(), localResults.size(), MPI_DOUBLE,
               0, MPI_COMM_WORLD);
```

OpenMP for Thread-Level Parallelism Within each MPI process, OpenMP was used to parallelize the evaluation of strategies. Each thread simulated a portion of the assigned strategies concurrently:

```
#pragma omp parallel for
for (int i = start; i < end; ++i) {
    localResults[i - start] = simulateRace(strategies[i],
        startingLapTime, trackData);
}
```

Hybrid Integration The combination of MPI and OpenMP enabled efficient use of both distributed and shared memory architectures. The workflow involved:

1. Broadcasting the strategy data to all processes.
2. Parallelizing strategy simulation within each process using OpenMP.
3. Gathering the results from all processes to identify the optimal strategy.

The following snippet demonstrates the hybrid workflow:

```
if (rank == 0) {
    // Master process generates and broadcasts strategies
    strategies = generateStrategies(totalLaps, trackData, compounds
        , stintVariable);
    std::string serializedData = serializeStrategies(strategies);
    int dataSize = serializedData.size();
    MPI_Bcast(&dataSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(serializedData.data(), dataSize, MPI_CHAR, 0,
        MPI_COMM_WORLD);
} else {
    // Worker processes receive the data
    int dataSize;
    MPI_Bcast(&dataSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
    char* buffer = new char[dataSize];
    MPI_Bcast(buffer, dataSize, MPI_CHAR, 0, MPI_COMM_WORLD);
    strategies = deserializeStrategies(std::string(buffer, dataSize
        ));
    delete[] buffer;
}
```

This approach ensured efficient parallelization while maintaining scalability across multiple nodes and threads.

3 Profiling

Profiling was conducted to identify performance bottlenecks and optimize the race strategy simulation program. Both instrumented and flat profiling approaches were used to gain insights into the execution times of key functions.

3.1 Profiling Methodology

The profiling process consisted of the following steps:

- **Compilation with Profiling Flags:** The program was compiled with the `-pg` flag to enable `gprof` profiling and with additional instrumentation for manual profiling.
- **Execution with Representative Inputs:** The simulation was run with various input parameters to generate a significant workload. This ensured that profiling data reflected real-world usage.
- **Analysis of Results:** Flat profiles and instrumented logs were analyzed to identify functions consuming the most execution time.

3.2 Flat Profiling with `gprof`

Flat profiling was performed using `gprof`. The output highlighted the time spent in various functions and their call counts. However, due to the use of a template-heavy JSON parsing library (`nlohmann::json`), the profiling output was dominated by low-level template instantiations and utility functions. A portion of the `gprof` output is shown below:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.00      0.01     0.01    15892     0.00     0.00  nlohmann::
    json_abi_v3_11_3::detail::lexer::get()
  0.00      0.01     0.00    49381     0.00     0.00  std::__cxx11
      ::basic_string::_M_data() const
...
```

Key observations:

- The output was dominated by utility functions from the `nlohmann::json` library, such as `lexer::get()` and other low-level template instantiations.

- Template-heavy libraries, while flexible and powerful, can make flat profiles less intuitive by distributing execution time across many small functions.
- Functions critical to the simulation, such as `generateStrategiesHelper()` and `simulateRace()`, were less prominent in the flat profile, despite being key computational components.

3.3 Instrumented Profiling

To gain more granular insights into the performance of key functions, manual instrumentation was added. Execution times were measured for critical sections using high-resolution timers. A subset of the profiling log is shown below:

```
generateStrategiesHelper execution time: 16 microseconds
generateStrategiesHelper execution time: 32 microseconds
simulateRace execution time: 9 microseconds
simulateRace execution time: 18 microseconds
```

The instrumented profiling revealed:

- `generateStrategiesHelper()` was invoked `132,340,850` times, with execution times ranging from `16 microseconds` to over `1 second`.
- `simulateRace()` was called `22,840,033` times, with execution times averaging around `9 microseconds`.

Instrumented profiling allowed for precise measurement of function-level execution times, complementing the flat profile's high-level view.

3.4 Profiling Insights

The profiling revealed several key insights:

- **JSON Parsing Overhead:** The dominance of JSON library functions in the `gprof` output highlighted significant overhead during input processing. The template-heavy nature of the `nlohmann::json` library led to an explosion of small functions, making it challenging to pinpoint specific bottlenecks in the flat profile.
- **Strategy Generation Costs:** The recursive nature of `generateStrategiesHelper()` contributed significantly to execution time, as evidenced by its high call count in the instrumented logs.
- **Effective Parallelization:** The distributed workload for `simulateRace()` demonstrated effective use of MPI and OpenMP, with well-balanced execution across threads and processes.

3.5 Future Optimizations

Based on the profiling results, the following optimizations are recommended:

- **Optimize JSON Handling:** Preprocessing JSON data into a binary format or switching to a lightweight library could significantly reduce parsing overhead.
- **Improve Strategy Generation:** Introduce early pruning in `generateStrategiesHelper()` to reduce unnecessary recursive calls and computational complexity.
- **Profiling Beyond `gprof`:** Consider alternative profiling tools that better handle template-heavy codebases, such as `perf` or VTune, for deeper insights.

Profiling demonstrated the trade-offs between the flexibility of template-heavy libraries and the clarity of performance analysis. Instrumented profiling complemented flat profiling by providing detailed execution metrics for critical functions, guiding targeted optimizations.

4 Scaling Analysis

4.1 Methodology

To analyze the performance of the race strategy simulator, we evaluated its **strong scaling** and **weak scaling** performance using the SciNet HPC cluster. The purpose was to measure how well the program utilizes additional computational resources under two scenarios:

- **Strong Scaling:** Fixed total workload with increasing processor counts.
- **Weak Scaling:** Increasing workload proportionally with the number of processors, ensuring constant workload per processor.

Strong Scaling

- **Setup:** Fixed workload with `Stint = 10` and `Laps = 50`. Process counts: $P = 1, 2, 4, 8, 16, 32$.
- **Code Snippet:**

```
for P in "${PROCESSORS[@]}"; do
  echo "Running with $P processes..."
  mpirun -np $P ./generate_strategies "$TRACK_NAME" 10 50 90.0
  # Record execution time
  END_TIME=$(date +%s.%N)
  WALLTIME=$(echo "$END_TIME - $START_TIME" | bc)
  echo "$P,$WALLTIME" >> strong_scaling_results.txt
done
```

Weak Scaling

- **Setup:** Workload scaled proportionally with processors:
  - `Stint = Base Stint` $\times P$.
  - `Laps = Base Laps +` $\lfloor \frac{\texttt{Base Laps}}{10} \cdot P \rfloor$.

  Process counts: $P = 1, 2, 4, 8, 16, 32$.
- **Code Snippet:**

```
for P in "${PROCESSORS[@]}"; do
  STINT=$(($BASE_STINT * $P))
  LAPS=$(($BASE_LAPS + $BASE_LAPS / 10 * $P))
  mpirun -np $P ./generate_strategies "$TRACK_NAME" $STINT $LAPS
      90.0
  # Record execution time
  END_TIME=$(date +%s.%N)
  WALLTIME=$(echo "$END_TIME - $START_TIME" | bc)
  echo "$P,$STINT,$LAPS,$WALLTIME" >> weak_scaling_results.txt
done
```

4.2 Results

Strong Scaling Results

Processes	**Walltime (s)**	**Speedup**	**Efficiency**
1	2.46	1.00	1.00
2	1.90	1.30	0.65
4	2.31	1.07	0.27
8	0.38	6.48	0.81
16	0.35	7.12	0.45
32	0.35	7.09	0.22

Table 1: Strong Scaling Results

```
Speedup vs Processes:
+-----------------------------------------------------+
| Speedup                                             |
| 9                                                   |
| 8            o                                      |
| 7     o      o                                      |
| 6            o                                      |
| 5                                                   |
| 4                                                   |
| 3 o                                                 |
| 2 o                                                 |
| 1 o                                               |
+----------------------------------------------------+
  1    2     4     8     16      32  (Processors)
```

Weak Scaling Results

Processes	Stint	Laps	Walltime (s)	Efficiency
1	10	55	1.84	1.00
2	20	60	0.71	2.58
4	40	70	1.74	1.06
8	80	90	0.31	5.94
16	160	130	0.31	6.02
32	320	210	0.31	6.01

Table 2: Weak Scaling Results

```
Walltime vs Processes:
+----------------------------------------------------+
| Walltime                                           |
| 2                                                  |
| 1 o                                                |
| 0            o                                     |
|              o                                     |
|              o                                     |
|              o  o                                  |
|              o                                     |
+----------------------------------------------------+
  1    2     4     8     16      32  (Processors)
```

```
Efficiency vs Processes:
+----------------------------------------------------+
| Efficiency                                         |
| 1 o                                                |
| 0.8                                                |
| 0.6            o                                   |
| 0.4                     o                          |
| 0.2                          o             o       |
+----------------------------------------------------+
  1    2     4     8     16      32  (Processors)
```

4.3 Conclusions

Strong Scaling

- **Trends:** Speedup is non-linear, with diminishing returns for $P \geq 8$.

- **Limitations:** Bottlenecks arise from serial portions and communication overhead.
- **Improvements:** Optimize serial code, reduce synchronization points, and balance workload distribution.

Weak Scaling

- **Trends:** Walltime stabilizes for $P \geq 8$, indicating good weak scaling performance.
- **Strengths:** Effective parallelization enables handling proportional workload increases efficiently.
- **Limitations:** Minor inefficiencies observed at $P = 4$ due to workload distribution.

Summary: The program exhibits strong weak scaling performance, while strong scaling requires further optimization to address bottlenecks in serial sections and communication overhead.

Challenges and Improvements

- Minor inefficiencies were observed at $P = 4$ due to workload imbalance.
- Optimizations ensured proportional workload distribution and minimized communication overhead.

5 Results and Discussion

5.1 Simulation Results

The simulation produced optimal race strategy configurations for various Formula 1 tracks based on input parameters such as total laps and starting lap times. The results are summarized in Table 3.

Track Name	**Optimal Strategy**	**Total Laps**	**Total Race Time (s)**
Belgian Grand Prix	HARD (22), HARD (22)	44	4933.79
Italian Grand Prix	MEDIUM (13), MEDIUM (13), MEDIUM (13), MEDIUM (14)	53	5000.21
Monaco Grand Prix	SOFT (18), SOFT (18), SOFT (18), SOFT (18), SOFT (6)	78	6121.79

Table 3: Optimal Strategies and Performance Metrics for Simulated Tracks

5.2 Discussion of Results

The simulation results demonstrate the effectiveness of the race strategy optimization approach, but they also highlight areas where assumptions in the model may lead to outcomes that are either realistic or unrealistic.

5.2.1 Realistic Results

The results for the **Belgian Grand Prix** and **Italian Grand Prix** align well with typical Formula 1 race strategies:

- **Belgian Grand Prix:** The use of two stints on hard tires is consistent with a track known for its long straights and high tire wear. The relatively long stint lengths (22 laps each) are realistic given the durability of the hard compound on a track like Spa-Francorchamps.
- **Italian Grand Prix:** The strategy of multiple medium tire stints is appropriate for a track like Monza, where medium tires strike a balance between speed and durability. The inclusion of a slightly longer final stint (14 laps) suggests a calculated approach to maximize performance in the closing laps.

5.2.2 Unrealistic Results

In contrast, the results for the **Monaco Grand Prix** appear less realistic:

- The use of five stints on soft tires, including a short final stint of 6 laps, seems overly aggressive for a narrow street circuit where overtaking is rare. In reality, teams would likely aim to minimize pit stops to avoid losing track position.
- The total race time of 6121.79 seconds (approximately 1 hour and 42 minutes) is within a reasonable range but may not fully account for the high fuel loads and frequent safety car deployments that are typical at Monaco.

5.2.3 Reasons for Unrealistic Results

The discrepancies in some results can be attributed to several factors:

- **Simplified Degradation Model:** The tire degradation model is based on average degradation rates per lap, which do not account for specific track characteristics like cornering forces, elevation changes, or surface abrasiveness.
- **Pit Stop Assumptions:** The simulation assumes a fixed time penalty for pit stops, without considering the variability introduced by factors such as pit crew performance or traffic in the pit lane.
- **Track-Specific Dynamics:** Unique track features like narrow corners at Monaco or high-speed straights at Monza are not explicitly modeled, leading to strategies that may not align with real-world constraints.
- **Starting Lap Time Estimates:** The starting lap times used in the simulations may not fully reflect the impact of fuel loads or specific driver performance on the opening laps.

5.2.4 Future Improvements

To enhance the realism of the simulation results, future iterations of the simulator could:

- Incorporate dynamic track conditions such as weather changes, fuel load effects, and safety car deployments.
- Develop a more granular degradation model that accounts for specific track and driving conditions.
- Introduce variability in pit stop penalties to reflect real-world uncertainties.
- Validate the simulator against historical race data to refine assumptions and improve accuracy.

Overall, the results provide valuable insights into race strategy optimization, but they also underscore the importance of refining the model to better capture the complexities of real-world Formula 1 racing.