Part 2: Trading Specific Algorithms

Problem overview

The task involves building a C++ project to connect to Binance USD(S)-M Futures API, receive and parse streams of aggregated trades, and print each trade's details. Additionally, we are required to measure and comment on the speed of the parsing algorithm and its complexity.

Steps taken to solve the task:

The project is divided into key components:

1. **Researching Binance's API:** Research the Binance USD(S)-M Futures API.
2. **Building Connectivity with Binance API**: Use Binance's API to fetch aggregated trade data using the GET /fapi/v1/aggTrades endpoint.
3. **Fetching and Parsing Trades**: Parse the incoming stream of trades and output each in a predefined format.
4. **Trade Output Format:** Print trades in required format.
5. **Handling Polling and User Input:** Incorporate keyboard input to control polling behavior.
6. **Measuring Trade Parsing Speed**: Measure how fast singular trades are parsed.
7. **Algorithmic Complexity**: Analyse the complexity of the parsing algorithm.

1. Researching Binance's API:
I started by reviewing Binance's API documentation for Futures at the provided URL. The GET /fapi/v1/aggTrades endpoint was identified as the correct method for fetching aggregated trades. The response format contains an array of trade objects, each having fields like aggregate tradeId, price, quantity, first tradeId, last tradeId, timestamp, and buyer maker status.

2. Building Connectivity with Binance API:
I implemented connectivity to Binance's GET /fapi/v1/aggTrades in BinanceAPI.cpp using the libcurl library for making HTTP requests. The request is formed by constructing the correct API URL with the symbol and other optional parameters. It is assumed that the symbol used to fetch aggregated trades is **BTCUSDT**. This is a common symbol representing the Bitcoin to USDT trading pair, and it simplifies development by focusing on a well-known pair. However, the code can be extended later to handle different symbols dynamically.

The response from the API contains a list of trade data, each trade containing fields like tradeId, price, quantity, timestamp, etc.

To make network requests in C++, the **libcurl** library was chosen because it is a widely-used and robust library for handling HTTP requests. Libcurl can handle network protocols like HTTP(S) and supports features like connection timeout, response parsing, and error handling, which are essential for interacting with the Binance API.

The function responsible for fetching trades from the Binance API was implemented as `getAggregatedTrades()` in the `BinanceAPI.cpp` file.

The Binance API requires a properly formatted URL to request aggregated trades for a given symbol. The URL is constructed using the symbol and optional parameters such as fromId.

- **Base URL**: https://fapi.binance.com/fapi/v1/aggTrades
- **Query Parameters**:
    - symbol (e.g., BTCUSDT).
    - fromId (optional): Helps avoid fetching duplicate trades by specifying the starting trade ID.
    - limit: Binance allows up to 1000 trades in one request.

In `BinanceAPI.cpp`, this is achieved using `std::ostringstream` to build the complete URL.

Once the URL is constructed, the next step is to set up cURL for making the HTTP GET request. Here's a breakdown of how the cURL setup is handled:

- **cURL Initialization**: `curl_easy_init()` is called to initialize the cURL session.
- **Set the Request URL**: The URL constructed earlier is passed to `curl_easy_setopt()` to specify the API endpoint.
- **Response Callback**: A callback function (`writeCallback`) is used to collect the response data from the API.

Once the request is made, cURL will fetch the data and store it using the `writeCallback` function. This function appends incoming data to a string buffer, which will later be parsed as a JSON response.

This callback function is passed to cURL, which calls it each time it receives data from the Binance API. The data is appended to the response string.

After making the request, the status of the request is checked to ensure it was successful. If the request failed (e.g., due to network issues or invalid parameters), an error message is logged. Additionally, cURL cleanup is done to free resources.

Finally, once the response has been received and stored, the data is returned as a string. This string contains the JSON-formatted response from the Binance API, which will later be parsed to extract individual trade details.

3. Fetching and Parsing Trades:

Once the trades were fetched from the Binance API in JSON format, the next step was to parse them. `nlohmann::json` library was chosen for handling JSON parsing. The solution aimed to extract each trade from the response and format it as per the requirement.

To handle the JSON data in C++, the `nlohmann::json` library was chosen. This library is a modern, fast, and easy-to-use JSON library for C++ that allows us to easily parse JSON strings and work with the data.

After the response is fetched from the API, it is passed as a string to the `printParsedResponse` function, which handles the trade parsing and processing. This function uses `nlohmann::json` to parse the response into a structured format.

Once the JSON response is parsed, the next step is to loop through each trade object and extract the relevant fields (`a, p, q, f, l, T, m`). The trades are stored in an array format within the JSON object, so a simple for-loop is used to process each trade.

The `lastTradeId` is updated with the value of the current trade's aggregate trade ID ("a"). This is crucial for ensuring that the next batch of trades fetched from the API starts from the correct point (i.e., it avoids re-fetching the same trades).

Although the response from the Binance API is expected to be well-formed JSON, it's important to handle any potential parsing errors, such as malformed JSON or network issues that return an incomplete response. To achieve this, the JSON parsing is wrapped inside a try-catch block to handle exceptions gracefully.

To handle the trade parsing asynchronously and avoid blocking the main thread, the parsing and printing of trades are done using a separate thread. This ensures that the system can continue polling for new trades while the current batch is being processed. (or if a websocket is used in the future) The thread is detached so that it runs independently of the main thread.

4. Trade Output Format:
The trade details are printed in the specified format using the `printParsedResponse` method in `BinanceAPI.cpp.`

5. Handling Polling and User Input:
One of the most important aspects of the project was managing the polling mechanism. The solution requires user interaction to either manually trigger polling, quit the program, or start continuous polling with a delay. A **WebSocket** approach was considered for real-time trade streaming since it would reduce the need for repeated polling and offer lower latency. However, due to the instructions specifying the use of the HTTP polling mechanism via the /fapi/v1/aggTrades endpoint, WebSocket was not used for this implementation.

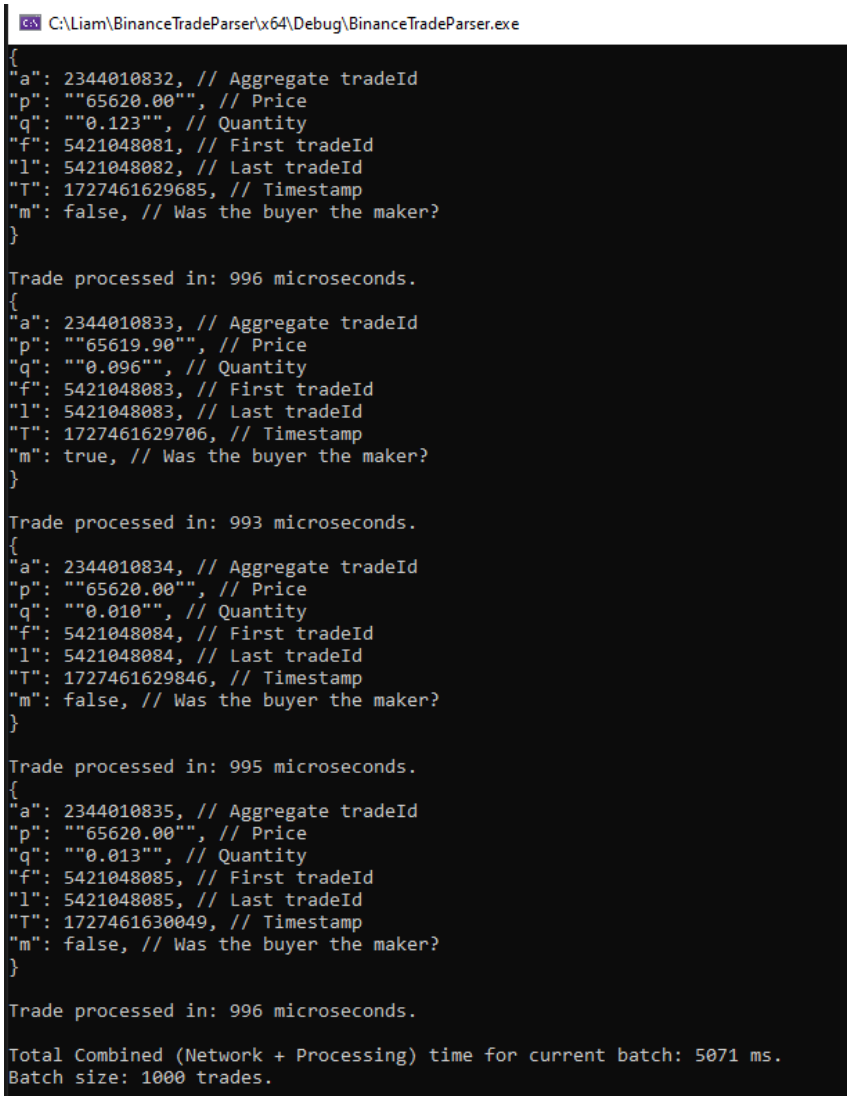
User input options:

- **Manual Polling**: The user presses the spacebar to manually initiate the next polling request.
- **Continuous Polling**: The user presses 1 to enable continuous polling, with a 5-second delay between each batch.
- **Quitting**: The user presses q to stop the program.

6. Measuring Trade Parsing Speed:

The final step was to measure the speed at which each trade was parsed. I added performance measurement by capturing the time taken to parse each batch of trades. The time is measured using std::chrono::high_resolution_clock to track how long parsing takes per trade.

The average time taken for these trades is roughly **1000 microseconds (or 1 milliseconds)** per trade, with some variability as seen in the output below.



7. Algorithmic Complexity:

The parsing algorithm has **O(n)** time complexity, where n is the number of trades in a batch. Parsing each trade requires constant time as it involves accessing fields in a JSON object and printing them, which is an **O(1)** operation. As a result, the overall complexity for parsing n trades is **O(n)**.

O(1) Operation for Each Trade:

- Each trade is represented as a JSON object, and for each trade, we access its fields (such as aggregate tradeId, price, quantity, timestamp, etc.).
- Accessing these fields from the JSON object, printing them, and updating the lastTradeId are constant-time operations, each taking **O(1)** time.

O(n) for the Entire Batch:

- Given that there are n trades in a batch, and we iterate over all of them once in the for loop, the time complexity for processing the entire batch is **O(n)**. This is because we perform **O(1)** operations for each of the n trades.
- Thus, the overall time complexity for parsing and processing the entire batch of trades is linear, **O(n)**.