DECEMBER 8, 2024

OS PROJECT REPORT

SAHAL SAEED, FARDEEN FARHAT, ZAIN UL WAHAB
FAST NUCES
ISLAMABAD

Contents

Group Members

- Sahal Saeed (22i-0476)
- Fardeen Farhat (22i-0638)
- Zain Ul Wahab (22i-0491)

File: globals.h

This file declares global variables and includes essential libraries. It defines the shared resources and synchronization mechanisms necessary for parallel processing.

1. **`#include <vector>`**
    - This header is included to use the `std::vector` container, which stores collections of data. Vectors are utilized for maintaining intermediate and final key-value pairs.
2. **`#include <pthread.h>`**
    - This header provides support for multithreading using POSIX threads (`pthreads`). The threads allow the parallel execution of tasks during the Map and Reduce phases.
3. **`#include <string>`**
    - This header allows usage of the `std::string` class, which represents text data in the form of strings.
4. **`using namespace std;`**
    - This eliminates the need to use the `std::` prefix with standard library objects like `vector` and `string`.
5. **Global Variables**
    - These variables are shared across different parts of the MapReduce system to facilitate the exchange of intermediate and final results among threads.
    - **`shared_map_data`**
        - A `vector<pair<string, int>>` used to store key-value pairs produced during the Map phase. It acts as a shared space where intermediate map results are stored before shuffling.
    - **`shuffled_data`**
        - A `vector<vector<pair<string, int>>>` used in the Shuffle phase to group key-value pairs by key. Each inner vector corresponds to a specific key and contains all the associated values.
    - **`reduced_data`**
        - A `vector<pair<string, int>>` used to store the final results after the Reduce phase. Each key is aggregated with its associated values to produce a single result.
    - **`map_mutex`**

        - A `pthread_mutex_t` mutex used to synchronize access to `shared_map_data` during the Map phase. It prevents race conditions when multiple threads simultaneously write to the shared resource.
6. **`#define MAX_THREADS 20`**
    - This preprocessor directive defines the maximum number of threads that can be created during the MapReduce process. It ensures the program doesn't exceed a predefined thread limit.
7. **Header Guards**
    - **`#ifndef GLOBALS_H` / `#define GLOBALS_H` / `#endif`**
        - These guards prevent multiple inclusions of this header file, avoiding redefinition errors during compilation.

File: globals.cpp

This file provides the definitions (allocations) for the global variables declared in `globals.h`.

1. **`#include "globals.h"`**
    - This includes the `globals.h` file to ensure that the variables declared in the header are properly allocated in this file.
2. **Global Variables Initialization**
    - **`shared_map_data`**
        - Allocated as an empty `vector<pair<string, int>>` to hold the intermediate results from the Map phase.
    - **`shuffled_data`**
        - Allocated as an empty `vector<vector<pair<string, int>>>` to hold grouped key-value pairs during the Shuffle phase.
    - **`reduced_data`**
        - Allocated as an empty `vector<pair<string, int>>` to store the final results from the Reduce phase.
    - **`pthread_mutex_t map_mutex`**
        - Initialized using `PTHREAD_MUTEX_INITIALIZER`. This ensures the mutex is ready for use and provides mutual exclusion when threads access `shared_map_data`.

Purpose of the Code

- **Global State Management:**
    - These variables allow the different phases of the MapReduce system (Map, Shuffle, and Reduce) to communicate by sharing data structures.
- **Thread Synchronization:**
    - The `map_mutex` ensures thread-safe operations on shared data in a multithreaded environment.

- **Scalability:**
    - Using a thread limit (`MAX_THREADS`) and shared data structures ensures the MapReduce system is scalable and adaptable to varying workloads.

File: split_sentence.h

This header file declares the interface for the sentence-splitting operation.

1. Includes

- `<vector>` and `<string>`: These headers support the `vector` container and `string` class needed to store and manipulate words.

2. Function Prototype

- **split_sentence(void* arg)**: A thread function that splits a sentence into a list of words. It returns a pointer to a dynamically allocated `vector` of words.

File: split_sentence.cpp

This source file contains the implementation of the `split_sentence` function.

1. Receiving the Input

- The argument `arg` is cast back to a `string` pointer:
    - `string sentence = *(string*)arg;`
    - This means the input sentence is passed as a pointer and then dereferenced to get the actual sentence string.
- **Purpose**: The thread extracts this sentence, splits it into words, and stores these words in a dynamically allocated `vector`.

2. Creating a New Word List

- A new `vector<string>` named `list_of_words` is dynamically allocated:
    - `vector<string>* list_of_words = new vector<string>();`
    - This is allocated on the heap because it is passed back to the calling thread using `pthread_exit`. The caller must ensure that it properly cleans up this memory later.

3. Splitting the Sentence into Words

- A loop iterates through each character in the sentence.
- **Checking for Spaces**:
    - If a space character (`' '`) is encountered, it means the end of a word:

        - The current `temp_word` is pushed back into `list_of_words`.
        - `temp_word` is then reset to an empty string to start collecting the next word.
    - If a non-space character is encountered, it is appended to `temp_word`.
- **Handling the Last Word**:
    - After the loop finishes, there might be a remaining word in `temp_word` (if the sentence does not end with a space).
    - This word is added to `list_of_words` before the function terminates.

4. Exiting the Thread

- `pthread_exit((void*)(list_of_words));`
    - Since the thread function returns a dynamically allocated `vector<string>`, it casts the pointer to `void*`.
    - The main thread (or the calling code) must handle this pointer correctly, converting it back into a `vector<string>` and cleaning up the memory once it's no longer needed.

Purpose of the Code

1. **Splitting Sentences into Words**
   Each thread processes a sentence independently and splits it into words. This enables each word to be fed into the Map phase to build word counts.
2. **Dynamic Memory Management**
   Allocating `list_of_words` on the heap allows returning a pointer to a `vector`. This ensures the memory persists even after the thread terminates.
3. **Concurrency**
   The use of threads enables parallel sentence processing, which can improve performance when dealing with large datasets or multiple sentences simultaneously.

File: map_operations.h

This header file declares the interface and necessary shared resources for the map operation.

1. **Includes**
- `<vector>` and `<string>`: These are standard C++ libraries for using the `vector` container and `string` class.
2. **Using Namespace**
- `using namespace std;`: Simplifies access to the C++ standard library classes and functions like `vector`, `pair`, and `string`.
3. **Global Variables Declaration**

- `shared_map_data`: A shared `vector` that stores the key-value pairs `(string, int)` representing words and their counts.
- `map_mutex`: A `pthread_mutex_t` that ensures safe access to `shared_map_data`. This prevents race conditions during concurrent modifications by multiple threads.

4. **Function Prototype**

- `add_to_map`: This is the thread function to process words in a chunk and map each word to a key-value pair `(word, 1)`.

File: map_operations.cpp

This source file implements the `add_to_map` function for the map phase.

1. **Includes**

- `<iostream>` and `<pthread.h>`: For basic input-output operations and threading support.
- `"map_operations.h"` and `"globals.h"`: Include the interface and global definitions to access shared variables and functions.

2. **`add_to_map` Function**

- The function takes a `void*` argument, which points to a `vector<string>` containing a chunk of words.
- The cast `(vector<string>*)arg` is used to convert the argument back to a `vector<string>` pointer.

3. **Thread Synchronization**

- The `pthread_mutex_lock` call locks the `map_mutex` to prevent multiple threads from concurrently modifying the `shared_map_data`. This ensures thread safety.
- The loop iterates over the `vector<string>` passed to it, adding each word to the `shared_map_data` as a key-value pair `(word, 1)`.

4. **Releasing the Lock**

- After processing, `pthread_mutex_unlock` unlocks the mutex, allowing other threads to access `shared_map_data`.

5. **Return Statement**

- The function returns `nullptr` as it does not need to pass any value back to the calling thread. This follows the POSIX convention for thread functions.

Purpose of the Code

- **Parallel Processing**: The code uses multiple threads to map words in parallel, which increases processing speed and scalability.
- **Data Aggregation**: It stores key-value pairs `(word, 1)` in the `shared_map_data` vector, which will later be grouped by key in the Shuffle phase.
- **Synchronization**: The `map_mutex` ensures that shared data is modified safely by coordinating access among threads.

File: shuffle.h

This header file declares the interface and shared resources for the shuffle operation.

1. **Includes**
- `<vector>` and `<string>`: These headers allow the use of the `vector` container and `string` class.
2. **Global Variable Declarations**
- **shared_map_data**: A global `vector` containing key-value pairs `(string, int)` representing words and their occurrence counts.
- **shuffled_data**: A 2-dimensional `vector` where each sub-vector stores key-value pairs grouped by a shared key.
- **map_mutex**: A `pthread_mutex_t` that ensures only one thread accesses `shared_map_data` at a time to avoid race conditions.
3. **Function Prototype**
- *shuffle_pairs(void* arg): A thread function that consolidates key-value pairs by sorting and grouping similar keys into `shuffled_data`.

File: shuffle.cpp

This source file implements the `shuffle_pairs` function for consolidating key-value pairs.

1. Acquiring the Mutex Lock

- The `pthread_mutex_lock` call ensures that only one thread accesses `shared_map_data` at a time.
- This prevents race conditions where multiple threads could simultaneously modify the shared vector, corrupting the data.

2. Sorting Key-Value Pairs

- The `sort` function arranges `shared_map_data` by key in ascending order.
- Since key-value pairs are now sorted, all identical keys are grouped consecutively in memory. This makes it easier to group them into sub-vectors in the subsequent loop.

3. Grouping Key-Value Pairs

- A temporary `current_group` vector is created to collect key-value pairs with the same key.
- The loop iterates through the `shared_map_data`:
    - If `current_group` is empty or the key of the current element matches the last key in `current_group`, the key-value pair is added to `current_group`.
    - If the key changes, it pushes `current_group` into `shuffled_data`, resets `current_group`, and starts a new group with the current key-value pair.

4. Finalizing Groups

- After the loop ends, any remaining `current_group` is added to `shuffled_data`.
- This ensures that all key-value pairs are grouped correctly, even if the last group isn't fully processed within the loop.

5. Releasing the Mutex Lock

- The `pthread_mutex_unlock` call unlocks the mutex, allowing other threads to safely access `shared_map_data` for any further operations.

6. Return Statement

- The function returns `nullptr`, following the POSIX thread convention where no value is returned to the calling thread.

Purpose of the Code

1. **Efficient Grouping of Key-Value Pairs**
   Sorting ensures that key-value pairs with identical keys are grouped consecutively, and the grouping operation consolidates these into sub-vectors within `shuffled_data`.
2. **Thread Safety**
   The use of `pthread_mutex_t` guarantees that only one thread modifies the shared `shared_map_data` at a time, maintaining data integrity during concurrent access.
3. **Preparing Data for Reduce Phase**
   The `shuffled_data` structure serves as input to the Reduce phase, where computations or aggregations (such as summing counts or finding averages) are performed on the grouped data.

File: reduce.h

This header file declares the interface and shared resources for the reduce operation.

1. Includes

- `<vector>` and `<string>`: These headers support the `vector` and `string` containers needed to manage key-value pairs.
- `map` is also included in `reduce.cpp` to store and aggregate key-value pairs efficiently.

2. Global Variables

- **shared_map_data**: Contains key-value pairs generated by the Map phase.
- **shuffled_data**: A 2-dimensional vector containing grouped key-value pairs after the Shuffle phase.
- **reduced_data**: A vector storing the aggregated key-value pairs after the Reduce phase.
- **map_mutex**: A `pthread_mutex_t` to ensure thread-safe operations on `reduced_data`.

3. Function Prototypes

- **reduce_group(void* arg)**: A thread function that processes each group of key-value pairs and aggregates their counts.
- **parallel_reduce()**: Manages the creation and execution of multiple threads to process all groups in parallel.

File: reduce.cpp

This source file implements the reduce operations using threads.

1. reduce_group Function

- **Input Handling**:
  - `arg` is cast back to a pointer to a `vector` containing key-value pairs.
  - `group` contains all key-value pairs with the same key after the Shuffle phase.
  - `map<string, int> key_count` is created to aggregate key occurrences efficiently.
- **Aggregation**:
  - For each key-value pair `(kv.first, kv.second)` in `*group`:
    - `key_count[kv.first] += kv.second` sums up the counts for each key.
    - This ensures that each key's count is aggregated correctly across the group.

- **Thread-Safe Insertion**:
  - `pthread_mutex_lock(&map_mutex)` ensures that only one thread updates the `reduced_data` vector at a time.
  - The aggregated key-value pairs are then pushed into the global `reduced_data` vector.
  - `pthread_mutex_unlock(&map_mutex)` releases the mutex lock to allow other threads to access `reduced_data`.
- **Return Statement**:
  - The thread function returns `nullptr` as per the POSIX standard.

2. parallel_reduce Function

- **Creating Threads**:
  - A `vector<pthread_t>` named `threads` stores the thread IDs.
  - For each group in `shuffled_data` (each group corresponds to a unique key group):
    - A thread is created using `pthread_create`.
    - Each thread calls `reduce_group` to aggregate key-value pairs in the group independently.
- **Joining Threads**:
  - The `for` loop iterates through the `threads` vector, using `pthread_join` to wait for each thread to finish its execution.
  - This ensures that all reduce operations are completed before the main thread continues execution.

Purpose of the Code

1. **Efficient Aggregation**
   The `reduce_group` function aggregates key-value pairs within each group, summing their counts to produce a consolidated result for each unique key.
2. **Parallel Execution**
   The `parallel_reduce` function leverages multithreading to process different groups simultaneously. Each group is handled by a separate thread, ensuring efficient and faster aggregation.
3. **Thread Safety**
   The use of `pthread_mutex_t` guarantees that concurrent threads safely update the shared `reduced_data` vector without race conditions or data corruption.

File Overview (main.cpp)

- The main function handles the workflow across different phases (Splitting, Mapping, Shuffling, and Reducing).
- It uses POSIX threads to execute operations concurrently and utilizes mutex locks (`map_mutex`) to ensure thread safety.

Step-by-Step Breakdown

1. File Input Handling

- **Reading the Input File**:
  - The program opens the input file `test2.txt`.
  - It reads the entire file content into a string `input` until the end of the file (`'\0'`).
  - If the file cannot be opened or if it is empty, an error message is printed, and the program exits.
  - After reading the input, unnecessary characters (punctuation, special characters) are removed from `input` to retain only alphanumeric characters and spaces. This cleaning process helps in processing words more effectively.

2. Initializing the Mutex

- **pthread_mutex_init**:
  - The `pthread_mutex_init` function initializes the global mutex `map_mutex`. This mutex is critical to protect shared data structures (like `shared_map_data`) from race conditions during multithreading operations.

3. Splitting Phase

- **Purpose**:
  - The `split_sentence` phase splits the input sentence into individual words.
- **Thread Creation**:
  - A thread `split_sentence_tid` is created to execute the `split_sentence` function.
  - This thread receives a reference to the `input` string and processes it to split the sentence into a `vector<string>` containing words.
- **Joining the Thread**:
  - The main thread waits for the `split_sentence` thread to finish using `pthread_join`.
  - The result (a pointer to the `vector<string>` containing words) is stored in `result`.
  - The `result` pointer is then cast back to a `vector<string>*` and assigned to `words`.

- **Splitting Words into Parts**:
  - The `words` vector is divided into three parts (`part1`, `part2`, `part3`).
  - This division ensures that the workload is distributed across multiple threads during the Mapping phase.

4. Mapping Phase

- **Purpose**:
  - In the Mapping phase, the `add_to_map` function maps each word to a key-value pair `(word, 1)`.
- **Thread Creation**:
  - Three threads (`thread1`, `thread2`, `thread3`) are created, each responsible for adding a part of words to the shared `shared_map_data` vector.
  - `pthread_create` calls pass a reference to each part's data to the `add_to_map` function.
  - The `add_to_map` function updates the global `shared_map_data` while using `pthread_mutex_lock` to ensure thread safety.
- **Joining the Threads**:
  - `pthread_join` ensures that the main thread waits until all mapping threads have completed.
  -
- **Output**:
  - The `shared_map_data` vector contains key-value pairs representing the word frequencies after the Mapping phase.
  - It is printed to the console for verification.

5. Shuffling Phase

- **Purpose**:
  - In the Shuffle phase, the `shuffle_pairs` function groups key-value pairs based on their keys.
- **Thread Creation**:
  - A single thread `shuffle_thread` is created to execute the `shuffle_pairs` function.
  - The `shuffle_pairs` function:
    - Locks the mutex `map_mutex`.
    - Sorts the `shared_map_data` based on keys.
    - Groups key-value pairs into `shuffled_data` (a 2-dimensional vector).
- **Joining the Thread**:
  - The main thread waits for the `shuffle_thread` to finish using `pthread_join`.
  - The `shuffled_data` contains grouped key-value pairs where each group contains a unique key and its associated word counts.

- **Output**:
    - The contents of `shuffled_data` are printed to show the groups of key-value pairs after the Shuffle phase.

6. Reducing Phase

- **Purpose**:
    - In the Reduce phase, `parallel_reduce` consolidates the grouped key-value pairs into final aggregated key-value pairs.
- **Thread Creation**:
    - The `parallel_reduce` function creates multiple threads, each working on one group of `shuffled_data`.
    - Each thread calls `reduce_group` to aggregate the key-value pairs within its group, summing the counts.
- **Joining Threads**:
    - `pthread_join` ensures that all threads finish execution before proceeding.
- **Output**:
    - The `reduced_data` vector stores the final aggregated key-value pairs (words and their frequencies).
    - The contents of `reduced_data` are displayed to show the word frequencies obtained after reducing.

Resource Cleanup

- **Memory Cleanup**:
    - The dynamically allocated `vector<string>` (created during the `split_sentence` phase) is explicitly deleted to free memory.
- **Destroying the Mutex**:
    - `pthread_mutex_destroy` cleans up the `map_mutex` resource to prevent memory leaks or resource exhaustion.

Explanation of making_test_case.py

The Python script is designed to generate a test input file named `test2.txt` for our MapReduce program. This test file contains a large volume of repetitive text to simulate a real-world, large-scale dataset and evaluate the program's performance and correctness.

Step-by-Step Breakdown

1. **Opening the File**
   The script opens a file named `test2.txt` in write mode using `with open("test2.txt", "w") as file:`.
   - If the file exists, it is overwritten.
   - If it does not exist, it is created.
2. **Defining the Text Block**
   A multi-line string `block` contains sample text about data science and related topics. The content includes phrases like "data science is amazing" and "machine learning and deep learning are subsets of artificial intelligence."
   - This ensures the dataset has repeated words for testing the map, shuffle, and reduce phases.
3. **Cleaning the Text**
   The script processes the block to remove unnecessary whitespaces and condense it into a single line using `' '.join(block.split())`.
   - This removes any extra spaces, making it easier for your program to process the text.
4. **Generating Repeated Content**
   The cleaned text is repeated 1000 times and written to the file using `file.write(cleaned_block * 1000)`.
   - This step creates a significantly large file, ensuring the test case simulates high computational demand.

Purpose of the Script

The test case created by this script is ideal for:

- **Evaluating Performance**: Ensures the program can handle large-scale datasets efficiently.
- **Testing Correctness**: Confirms the MapReduce implementation correctly handles repeated data and produces accurate results.
- **Identifying Bottlenecks**: Helps you analyze the scalability of multithreaded operations.

Activity Diagram