Implementing Simple Multi-Threader

This implementation provides a modular and efficient framework for parallelizing both 1D and 2D iterative computations using Pthreads in C++. It abstracts the complexities of thread management by introducing two primary functions, parallel_for for 1D and 2D loops, which distribute workloads among multiple threads to achieve concurrency. The framework leverages lambda expressions to allow users to define custom operations over specified ranges, with thread-safe execution and proper synchronization. Additionally, helper functions are included to manage thread creation and joining, ensuring clean and maintainable code.

Explanation of Each Function in the Implementation

1. processVectorRange

- **Purpose**: Implements the task for each thread in a 1D parallel_for loop.
- **Implementation**:
    - Accepts a pointer to a thread_args_vector structure containing the range of indices (low to high) and the lambda function.
    - Iterates over the range and calls the lambda function for each index.
    - Deletes the dynamically allocated thread_args_vector to prevent memory leaks.
- **Functionality**:
    - Ensures each thread executes a portion of the 1D workload independently.

2. processMatrixRange

- **Purpose**: Implements the task for each thread in a 2D parallel_for loop.
- **Implementation**:
    - Accepts a pointer to a thread_args_matrix structure containing the range of indices (low1, high1 and low2, high2) and the lambda function.
    - Iterates over the 2D range and calls the lambda function for each pair (i, j).
    - Deletes the dynamically allocated thread_args_matrix to prevent memory leaks.
- **Functionality**:
    - Ensures each thread executes a portion of the 2D workload independently.

3. createThreads

- **Purpose**: Creates threads for either 1D or 2D processing.
- **Implementation**:
    - Loops through numThreads, creating a thread for each using pthread_create.
    - Associates each thread with the corresponding task (vector or matrix) and arguments.
    - Exits the program if a thread fails to create.

- **Functionality**:
    - Handles thread creation, linking each thread to its task and arguments.

4. joinThreads

- **Purpose**: Ensures all threads complete execution before proceeding.
- **Implementation**:
    - Loops through numThreads, joining each thread using pthread_join.
    - Exits the program if a thread fails to join.
- **Functionality**:
    - Ensures the main program waits for all threads to finish before proceeding.

5. parallel_for (1D)

- **Purpose**: Implements a parallelized 1D loop using multiple threads.
- **Implementation**:
    - Validates the number of threads and the range (low < high).
    - Divides the range [l, h) into chunks of size chunk and distributes them among numThreads.
        - Handles any leftover indices (remainder) by assigning them to the first few threads.
    - Creates thread_args_vector dynamically for each thread.
    - Calls createThreads and joinThreads to manage execution.
    - Records and reports the total execution time.
- **Functionality**:
    - Distributes the 1D workload among multiple threads for parallel processing.

6. parallel_for (2D)

- **Purpose**: Implements a parallelized 2D loop using multiple threads.
- **Implementation**:
    - Validates the number of threads and the ranges (l1 < h1 and l2 < h2).
    - Divides the outer range [l1, h1) into chunks and distributes them among numThreads.
        - Handles any leftover rows (remainder1) by assigning them to the first few threads.
    - Creates thread_args_matrix dynamically for each thread.
    - Calls createThreads and joinThreads to manage execution.
    - Records and reports the total execution time.
- **Functionality**:
    - Distributes the 2D workload among multiple threads for parallel processing.

7. user_main and main

- **Purpose**: Acts as an entry point for user-defined programs.
- **Implementation**:

    - The user's program logic is implemented in user_main.
    - main simply redirects execution to user_main and returns its result.
- **Functionality**:
    - Makes the multithreading implementation independent of user programs by allowing user-specific logic in user_main.

Code Flow for vector.cpp:

1. Initialize Problem Size:

- Reads the number of threads (numThread) and the size of the vectors (size) from the command-line arguments.
- Defaults:
    - numThread = 2
    - size = 48,000,000

2. Allocate Vectors:

- Dynamically allocates three vectors (A, B, and C) of size size.
- A and B are input vectors, and C is the output vector.

3. Initialize Vectors:

- Fills A and B with 1s and C with 0s using std::fill.

4. Perform Parallel Addition:

- Calls parallel_for(0, size, ...) with a lambda function: C[i] = A[i]+B[i]
- Workload:
    - The range [0, size) is divided into chunks, and each thread processes its assigned indices independently.
    - Each thread accesses A[i] and B[i], performs the addition, and writes the result to C[i].

5. Verify Result:

- Loops through the range [0, size) to check that C[i] == 2 for every index.
- If any value is incorrect, assert(C[i] == 2) will terminate the program.

6. Print Success Message:

If all elements in C are correct, prints: Test Success

7. Clean Up Memory:

- Frees the dynamically allocated memory for A, B, and C using delete[].

Code Flow for matrix.cpp:

1. Initialize Problem Size:

- Reads the number of threads (numThread) and the size of the matrices (size) from the command-line arguments.
- Defaults:
    - numThread = 2
    - size = 1,024

2. Allocate Matrices:

- Dynamically allocates A, B, and C as arrays of pointers to arrays (int**).
- Initializes each row in parallel using parallel_for(0, size, ...):
  A[i] = new int[size];

  B[i] = new int[size];

  C[i] = new int[size];

3. Initialize Matrices:

- Inside the same parallel_for, fills each row of A and B with 1s and each row of C with 0s using std::fill.

4. Perform Parallel Multiplication:

- Calls parallel_for(0, size, 0, size, ...) to perform matrix multiplication:
  C[i][j] += A[i][k] * B[k][j];
- Workload:
    - The outer loop (i) is divided into chunks among threads.
    - Each thread processes a subset of rows in parallel.
    - The inner loop (j) calculates the dot product of the ith row of A and the jth column of B.

5. Verify Result:

- Loops through the matrix to check that C[i][j] == size for all (i, j):
    - Each element of C should be the sum of size multiplications (1 * 1 for every k).
- If any value is incorrect, assert(C[i][j] == size) will terminate the program.

6. Print Success Message:

- If all elements in C are correct, prints: Test Success.

7. Clean Up Memory:

- Calls parallel_for(0, size, ...) to delete rows of A, B, and C in parallel:
  delete[] A[i];

  delete[] B[i];

  delete[] C[i];

- Deletes the outer arrays:
  delete[] A;

  delete[] B;

  delete[] C;

GitHub Repository link: https://github.com/Abhinav0821/MultiThreading.git

Contribution:
1. Abhinav Kashyap, 2023022: Thread Management, 2D parallel loop, Thread Argument for 2D, Documentation, Execution time evaluation

2. Harsh Sharma, 2023233: Thread Management, 1D parallel loop, Thread Argument for 1D, Documentation, Execution time evaluation