Sure, let's go through the code line by line to understand each part:

cpp

Copy code

```cpp
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <string>
#include <cstring>
#include <cstdlib>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <mpi.h>
```

These are the include directives necessary for the code. They include standard C++ libraries for input/output (`iostream`), file input/output (`fstream`), string streams (`sstream`), vectors (`vector`), strings (`string`), C-style string manipulation functions (`cstring`), and POSIX shared memory APIs (`unistd.h`, `sys/types.h`, `sys/ipc.h`, `sys/shm.h`). Additionally, we include the MPI header file (`mpi.h`) for MPI function declarations.

```cpp
std::string trim(const std::string &str)
{
    std::string trimmedString(str);
    trimmedString.erase(0, trimmedString.find_first_not_of(" \t\n\r"));
    trimmedString.erase(trimmedString.find_last_not_of(" \t\n\r") + 1);
    return trimmedString;
}
```

This function `trim()` is used to trim leading and trailing whitespaces from a string. It takes a string `str` as input, creates a copy called `trimmedString`, and then uses `erase()` along with `find_first_not_of()` and `find_last_not_of()` to remove leading and trailing whitespaces, respectively. It then returns the trimmed string.

```
int main(int argc, char *argv[])
{
 MPI_Init(&argc, &argv);

 int world_rank;
 MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

 int world_size;
 MPI_Comm_size(MPI_COMM_WORLD, &world_size);

 if (world_size != 4) {
 std::cerr << "This program requires 4 MPI processes to run." << std::endl;
 MPI_Finalize();
 return 1;
 }
```

In the `main()` function, we initialize MPI with `MPI_Init()`. We then obtain the rank of the current process (`world_rank`) and the total number of processes (`world_size`) in the MPI communicator `MPI_COMM_WORLD`. If the total number of processes is not 4, we print an error message and finalize MPI before returning from the program.

```
std::filesystem::path rootFolderPath =
"/home/rnr/study/cmpe275/RRR-cmpe275/Mini-Project-2/airnow-2020fire/data"; //
Adapt this path if needed
```

Here, we define the path to the root folder containing the CSV files to be processed. You should adapt this path to match the location of your CSV files.

Continuing in the `main()` function:

```
// Define key for shared memory segment
 key_t key = 12345;

 // Create shared memory segment
 int shmid = shmget(key, sizeof(std::vector<std::vector<std::string>>) *
world_size, IPC_CREAT | 0666);
 if (shmid == -1) {
 std::cerr << "Failed to create shared memory segment." << std::endl;
 MPI_Finalize();
 return 1;
 }
```

Here, we define a key for the shared memory segment and then create the shared memory segment using `shmget()`. We specify the size of the shared memory segment as the size of a vector of vectors of strings (`std::vector<std::vector<std::string>>`) multiplied by the total number of MPI processes (`world_size`). We use the `IPC_CREAT` flag to create the segment if it does not already exist, and `0666` as the permissions for the shared memory segment.

If `shmget()` returns -1, it indicates an error in creating the shared memory segment, so we print an error message, finalize MPI, and return 1 to indicate failure.

```
// Attach shared memory segment
 std::vector<std::vector<std::string>> *all_data =
(std::vector<std::vector<std::string>> *)shmat(shmid, NULL, 0);
 if (all_data == (void *)-1) {
 std::cerr << "Failed to attach shared memory segment." << std::endl;
 MPI_Finalize();
 return 1;
 }
```

We attach the shared memory segment to the address space of the current process using `shmat()`. We cast the return value of `shmat()` to a pointer to a `std::vector<std::vector<std::string>>`, which represents the shared data structure `all_data`. If `shmat()` returns `(void *)-1`, it indicates an error in attaching the shared memory segment, so we print an error message, finalize MPI, and return 1 to indicate failure.

```
// Initialize shared memory
 if (world_rank == 0) {
 *all_data = std::vector<std::vector<std::string>>();
 }
```

We initialize the shared memory segment by creating an empty vector of vectors of strings (`std::vector<std::vector<std::string>>`) in the shared memory. This initialization is performed by the root process (rank 0) only.

```
// Barrier synchronization to ensure initialization is complete
 MPI_Barrier(MPI_COMM_WORLD);
```

We use `MPI_Barrier()` to synchronize all MPI processes, ensuring that the initialization of the shared memory segment is complete before proceeding.

The remaining code inside the `main()` function is similar to the original code and performs the following tasks:

- Iterates through the directory structure and processes CSV files, parsing them into vectors of strings.
- Adds the parsed data to the shared memory segment.
- Calculates the total number of rows across all processes using `MPI_Reduce()`.
- Detaches the shared memory segment using `shmdt()`.
- Removes the shared memory segment (only performed by the root process) using `shmctl()`.