PDC Project Report [Facial Recognition using Grayscale Histograms]

Member 1: Abdul Tawab (22K-0504)
Member 2: Muzaffar Ali (22K-4549)
Member 3: Sahir Hassan (22K-4570)

Abstract:

This project implements a parallelized image matching system using MPI (Message Passing Interface) to identify the most similar image to a given input based on histogram comparison. The program distributes image data across multiple processes, computes local results, and reduces these to a global result using MPI operations. It ensures efficient handling of data and accurate similarity computation, demonstrating the effectiveness of parallel processing in computationally intensive tasks like image matching.

MPI Project:

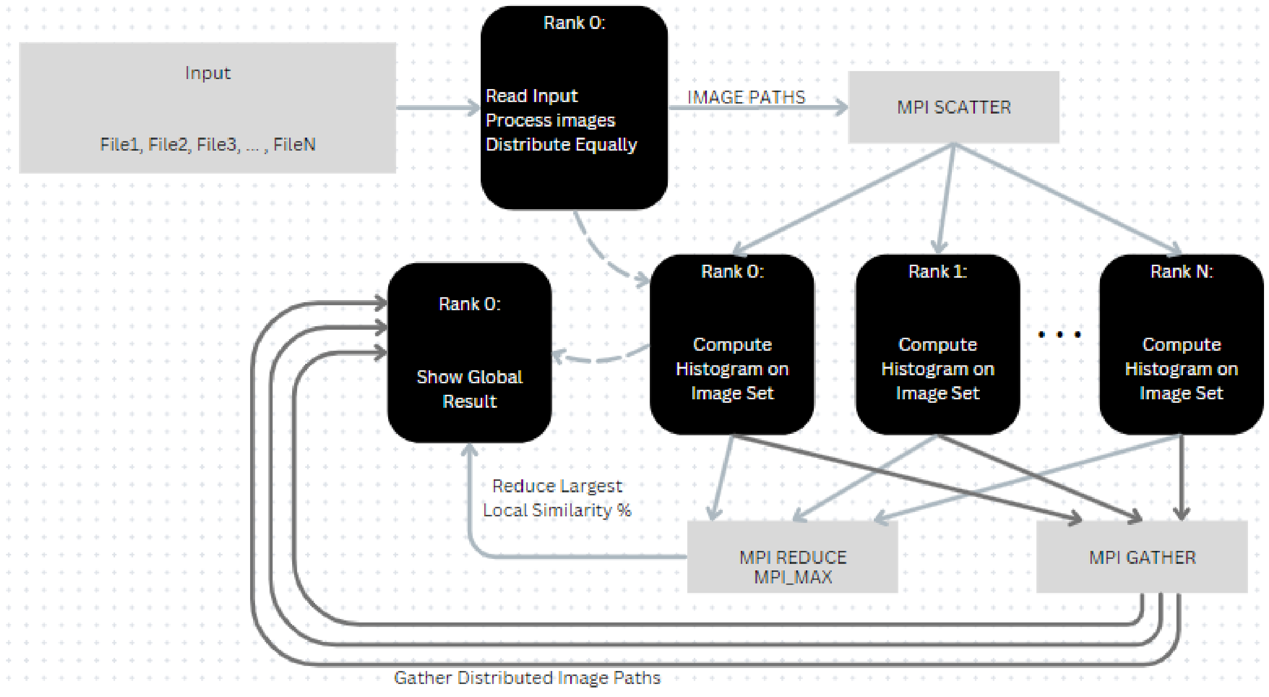
MPI Parallelization Overview

1. **Data Distribution**: Input image paths are split among processes.
2. **Independent Processing**: Each process computes similarities for its assigned paths in parallel.
3. **Synchronization**: MPI operations (`MPI_Reduce`, `MPI_Gather`) consolidate the results for final output.

Key MPI Functions:

Function	Purpose
`MPI_Scatter`	Distributes image paths from rank 0 to all processes.
`MPI_Reduce`	Aggregates the best similarity values into a global result.
`MPI_Gather`	Collects paths from all processes to rank 0 for final output processing.

Graphical Representation:


Input
File1, File2, File3, ... , FileN
Rank 0:
Read Input
Process images
Distribute Equally
IMAGE PATHS
MPI SCATTER
Rank 0:
Compute Histogram on Image Set
Rank 1:
Compute Histogram on Image Set
Rank N:
Compute Histogram on Image Set
Rank 0:
Show Global Result
Reduce Largest Local Similarity %
MPI REDUCE MPI_MAX
MPI GATHER
Gather Distributed Image Paths

Program Flow and Explanation

1. Initialization

The program begins by initializing the MPI environment:

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

- **MPI_Init**: Initializes the MPI environment.
- **MPI_Comm_rank**: Determines the rank (ID) of the process. Each process will handle a subset of images.
- **MPI_Comm_size**: Determines the total number of processes available for parallel computation.

At this stage, rank 0 reads all image file paths and distributes them among the processes.

2. Data Distribution

Rank 0:

1. **Loads image paths** from a directory.
2. Distributes the file paths among processes using `MPI_Scatter`.

```
MPI_Scatter(&allPaths[0],   chunkSize,   MPI_CHAR,   &localPaths[0],   chunkSize,
MPI_CHAR, 0, MPI_COMM_WORLD);
```

- Each process receives a subset (`localPaths`) of the total images for local computation.
- The `MPI_Scatter` ensures balanced data distribution across processes.

3. Local Computation

Each process:

1. Loads its assigned images and computes the histogram for each.
2. Compares each histogram to the target image using a **local matching function**.

This step uses the loop:

```
for (const auto &[path, histogram] : localLBPs) {
    double similarity = compareHistograms(testLBP, histogram);
    if (similarity > bestMatch) {
        bestMatch = similarity;
        bestImage = path;
    }
}
```

- Each process finds its **local best match** and stores it in a `Result` struct.

4. Global Reduction

After local computation, the processes synchronize their results using **MPI_Reduce** to find the overall best match:

```
MPI_Reduce(&localResult.similarity,   &globalBestSimilarity,   1,   MPI_DOUBLE,
MPI_MAX, 0, MPI_COMM_WORLD);
```

- **Operation**: `MPI_MAX` is used to find the maximum similarity value across all processes.
- The result is stored in rank 0 (`globalBestSimilarity`).

To gather the corresponding paths, **MPI_Gather** is used:

```
MPI_Gather(localResult.path,  256,  MPI_CHAR,  gatheredPaths,  256,  MPI_CHAR,  0,
MPI_COMM_WORLD);
```

- Rank 0 receives all paths (`gatheredPaths`) to determine the best matching path.

5. Final Output

Rank 0 processes the results from all ranks:

1. Finds the rank with the highest similarity.
2. Outputs the path and similarity value of the best match.

```
for (int i = 0; i < size; i++) {
    if (gatheredSimilarities[i] > highestSimilarity) {
        highestSimilarity = gatheredSimilarities[i];
        bestRank = i;
    }
}
cout << "Best match: " << gatheredPaths[bestRank] << " with similarity: " <<
highestSimilarity << endl;
```

Convention(s):

- **Input format**: Input images must be named in serial numbers/ iterations and they must be pngs.

Here's a section you can add to your project report detailing how you converted the MPI code to OpenMP, differences in parallelization, and an explanation of the OpenMP implementation.

Conversion from MPI to OpenMP

The first implementation achieved parallelism across multiple processes, with data distributed across nodes. Each process independently computed its results, and a reduction operation was used to find the global best match. This approach required explicit communication and synchronization between processes using MPI_Send, MPI_Recv, MPI_Reduce, and MPI_Gather.

In contrast, the OpenMP version leverages shared memory parallelism, where multiple threads execute in parallel within a single process. OpenMP simplifies parallelization by eliminating the need for explicit inter-process communication. Instead, threads share memory and can access the same data concurrently, enabling faster development and easier debugging.

Key Changes in the Code

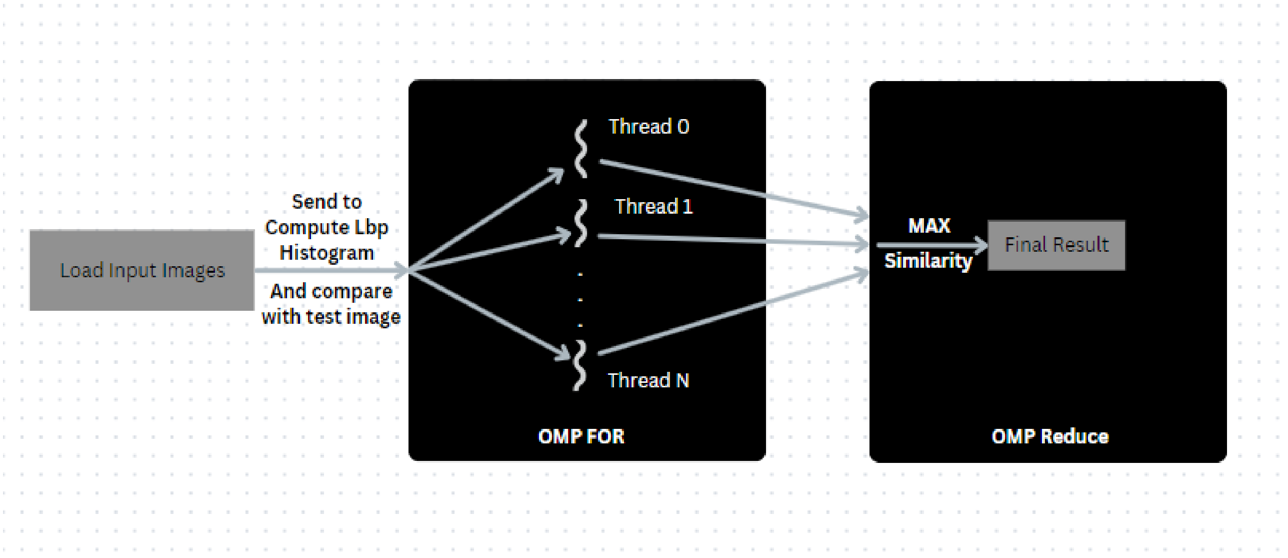
- Initialization of Threads:
  MPI required initializing and finalizing processes using MPI_Init and MPI_Finalize. These were replaced with OpenMP's #pragma omp parallel directives to spawn threads. The number of threads is controlled using the omp_set_num_threads function or the OMP_NUM_THREADS environment variable.

- Data Distribution:
  In MPI, data was explicitly split among processes, and each process operated on its subset. OpenMP relies on implicit thread parallelism, and data splitting was achieved using #pragma omp for, which divides iterations of a loop among threads.

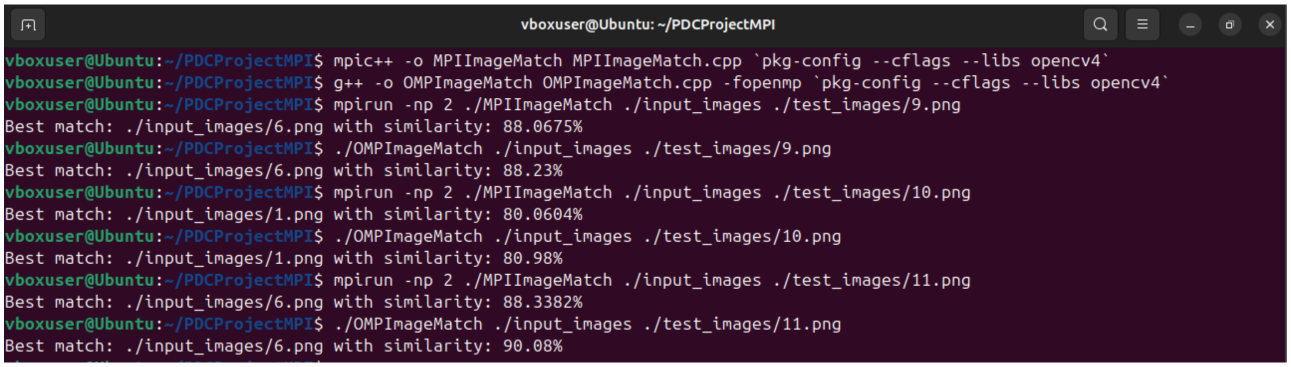
- Reduction Operation:
  MPI used MPI_Reduce to combine results from all processes. In OpenMP, this was replaced with the reduction clause, which allows threads to independently compute partial results and combine them at the end.

- Result Aggregation:
  MPI gathered results across processes with MPI_Gather. OpenMP eliminates this step since all threads share memory, and results are directly accessible without explicit communication.

Graphical Representation:



Accuracy:



```
vboxuser@Ubuntu:~/PDCProjectMPI$ mpic++ -o MPIImageMatch MPIImageMatch.cpp `pkg-config --cflags --libs opencv4`
vboxuser@Ubuntu:~/PDCProjectMPI$ g++ -o OMPImageMatch OMPImageMatch.cpp -fopenmp `pkg-config --cflags --libs opencv4`
vboxuser@Ubuntu:~/PDCProjectMPI$ mpirun -np 2 ./MPIImageMatch ./input_images ./test_images/9.png
Best match: ./input_images/6.png with similarity: 88.0675%
vboxuser@Ubuntu:~/PDCProjectMPI$ ./OMPImageMatch ./input_images ./test_images/9.png
Best match: ./input_images/6.png with similarity: 88.23%
vboxuser@Ubuntu:~/PDCProjectMPI$ mpirun -np 2 ./MPIImageMatch ./input_images ./test_images/10.png
Best match: ./input_images/1.png with similarity: 80.0604%
vboxuser@Ubuntu:~/PDCProjectMPI$ ./OMPImageMatch ./input_images ./test_images/10.png
Best match: ./input_images/1.png with similarity: 80.98%
vboxuser@Ubuntu:~/PDCProjectMPI$ mpirun -np 2 ./MPIImageMatch ./input_images ./test_images/11.png
Best match: ./input_images/6.png with similarity: 88.3382%
vboxuser@Ubuntu:~/PDCProjectMPI$ ./OMPImageMatch ./input_images ./test_images/11.png
Best match: ./input_images/6.png with similarity: 90.08%
```

After Compiling both programs and running them with the same test image and input image set. We can see same outcome from both implementations and as for accuracy of image matching/facial recognition here is the outcomes for both MPI and OMP.