Optimizing Image Classification with Parallel Computation Techniques: A Case Study Using the Fashion-MNIST Dataset

SIST British University Associate of Cardiff Met University
CIS6007 Parallel and Distributed Systems by Mr. Adam El Khaldi

By Mohamed El Bachrioui

Abstract

The rapid progress in computational technologies has greatly enhanced the skills of image classification, especially in the fields of facial and object recognition. This work investigates the incorporation of parallel computation techniques, such as multithreading and Message Passing Interface (MPI)/OpenMPI, to improve the speed and effectiveness of picture classification algorithms. These approaches enhance the efficiency and precision of computational processes, making them well-suited for real-time applications.

Introduction

Parallel computation has transformed picture processing. It turned basic task execution into complex recognition systems that work effectively in dynamic environments. Initially, image processing commands were processed consecutively. The increasing growth of digital video and image data rendered this method ineffective. This inefficiency was caused by the processor's idle time, which involved long delays while it finished one activity before starting another. Delays in image processing can be detrimental, especially when a single image requires many operations.

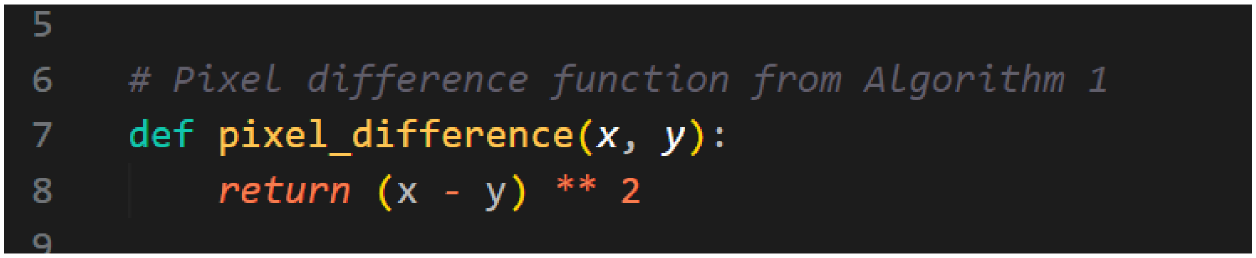
Parallel computing revolutionized processing approaches by shifting from sequential to parallel processing. Parallel computing, in contrast to its predecessor, involves the division of a task into smaller subtasks that are executed concurrently by many computer units. This strategy not only decreases the amount of time it takes to process, but also optimizes the utilization of the computing resources. Parallel computing in image processing enables simultaneous execution of tasks such as feature identification, pixel modification, and object recognition on various parts of an image or even many images.

Part I :

Five algorithms, fundamental to the tasks of facial and object recognition, have been developed and are detailed in an accompanying Python file. These algorithms leverage advanced image processing libraries and tools for parallel processing, specifically threading and OpenCV, to execute complex computations efficiently.

Task A (algorithm 1): Pixel difference calculation

This function directly implements the necessary operation $(x-y)^2$, which will serve as the foundation of your picture comparison technique. This calculation is an essential component of numerous image processing algorithms, particularly in tasks such as image distinction or error calculations.



```python

# Pixel difference function from Algorithm 1
def pixel_difference(x, y):
    return (x - y) ** 2

```

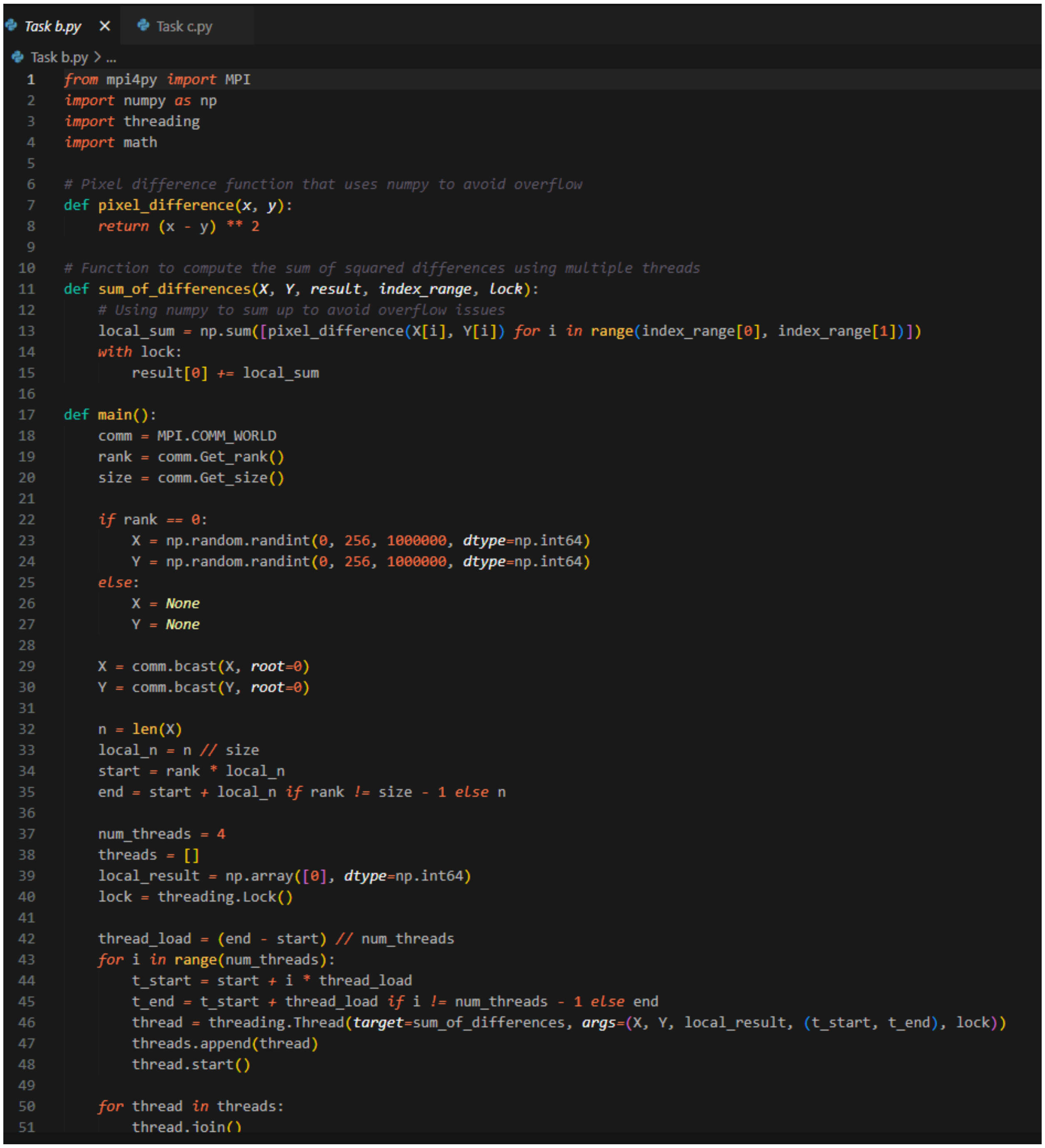
Role in Image Comparison: It improves the detection of edges and features, which is crucial for evaluating the similarity between images.

Task B (algorithm 2) : Sum of Differences for 1D Arrays

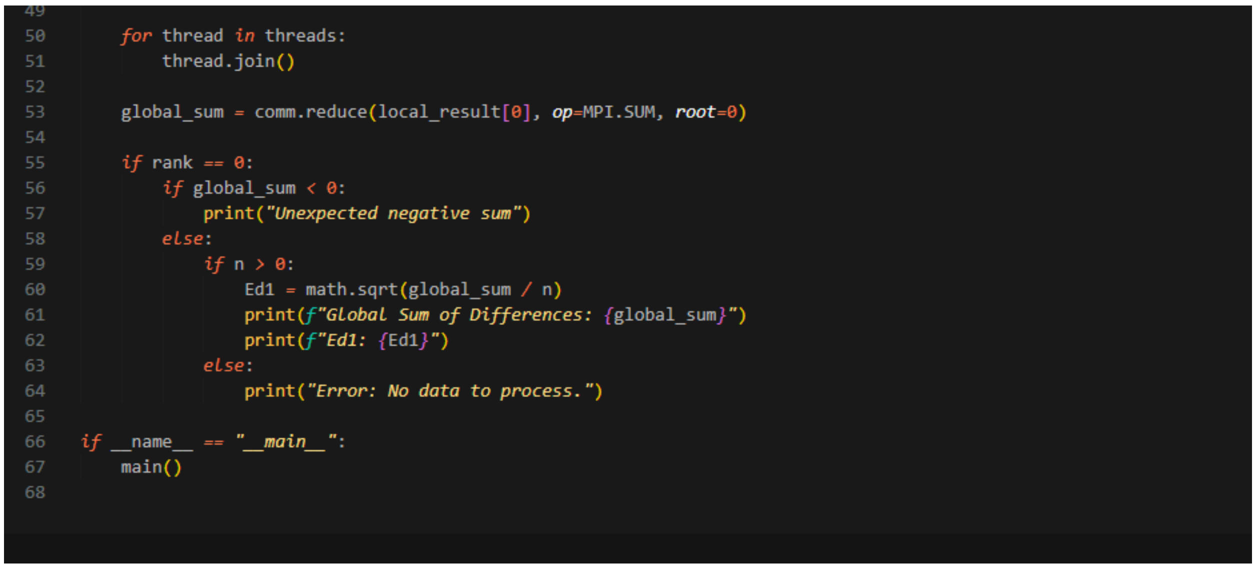
This algorithm calculates the sum of squared differences between corresponding pixels in two 1D arrays, which represent grayscale images. It is an extension of the pixel difference computation.
By utilizing multithreading, it distributes the effort among numerous threads, hence decreasing computation time and expediting picture comparison processes.

- Key Features:
    - Multithreading: Ensures the simultaneous and efficient processing of pixel differences in parallel.
    - Global Variable with Synchronization: This approach utilizes a global sum variable to collect differences, while applying locks to avoid data corruption and ensure consistency.

- Role in Image Comparison: Facilitates efficient image comparison by utilizing parallel processing techniques to compare images at the array level.



```python
from mpi4py import MPI
import numpy as np
import threading
import math

# Pixel difference function that uses numpy to avoid overflow
def pixel_difference(x, y):
    return (x - y) ** 2

# Function to compute the sum of squared differences using multiple threads
def sum_of_differences(X, Y, result, index_range, lock):
    # Using numpy to sum up to avoid overflow issues
    local_sum = np.sum([pixel_difference(X[i], Y[i]) for i in range(index_range[0], index_range[1])])
    with lock:
        result[0] += local_sum

def main():
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    if rank == 0:
        X = np.random.randint(0, 256, 1000000, dtype=np.int64)
        Y = np.random.randint(0, 256, 1000000, dtype=np.int64)
    else:
        X = None
        Y = None

    X = comm.bcast(X, root=0)
    Y = comm.bcast(Y, root=0)

    n = len(X)
    local_n = n // size
    start = rank * local_n
    end = start + local_n if rank != size - 1 else n

    num_threads = 4
    threads = []
    local_result = np.array([0], dtype=np.int64)
    lock = threading.Lock()

    thread_load = (end - start) // num_threads
    for i in range(num_threads):
        t_start = start + i * thread_load
        t_end = t_start + thread_load if i != num_threads - 1 else end
        thread = threading.Thread(target=sum_of_differences, args=(X, Y, local_result, (t_start, t_end), lock))
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()
```

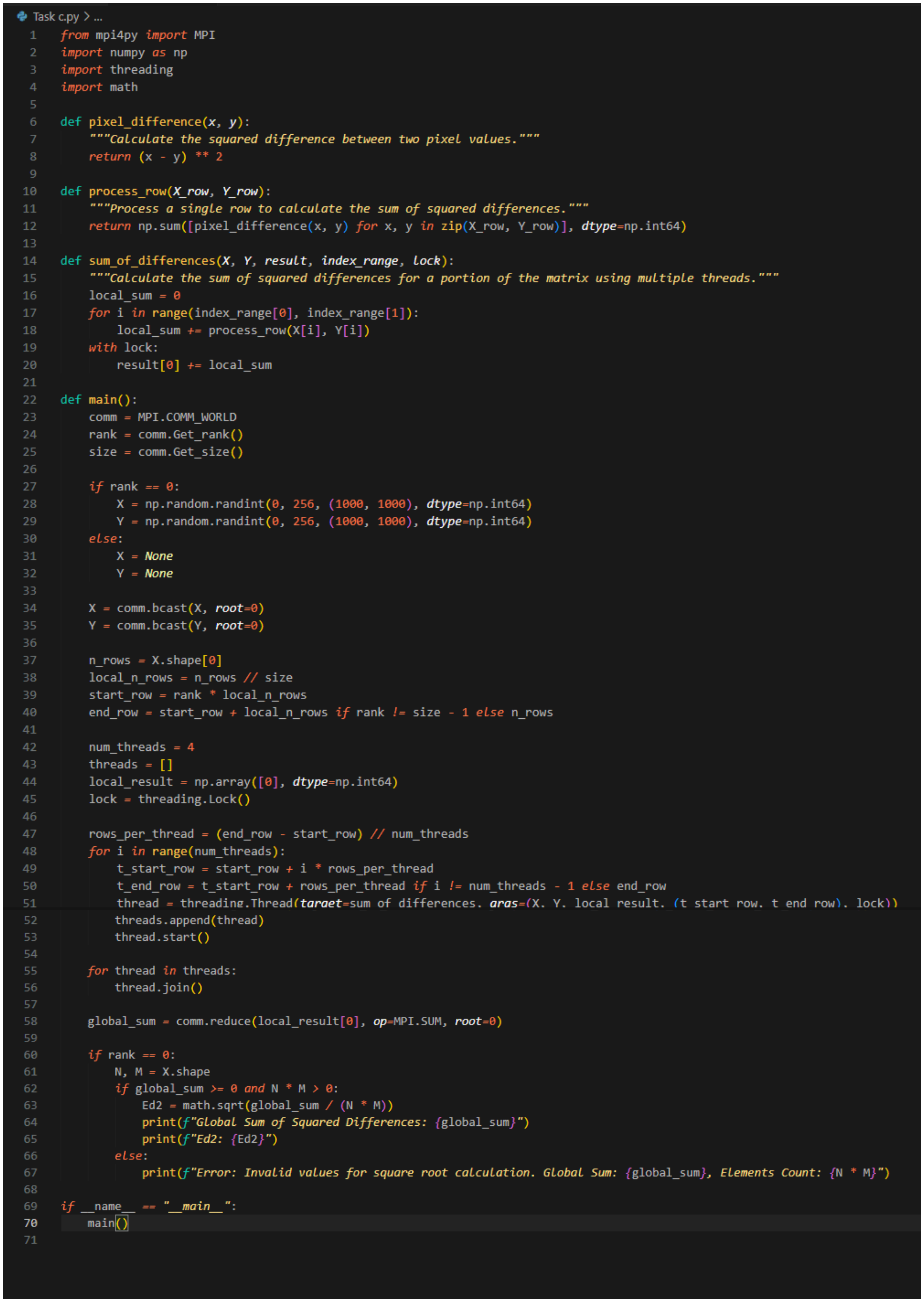


```
    for thread in threads:
        thread.join()

    global_sum = comm.reduce(local_result[0], op=MPI.SUM, root=0)

    if rank == 0:
        if global_sum < 0:
            print("Unexpected negative sum")
        else:
            if n > 0:
                Ed1 = math.sqrt(global_sum / n)
                print(f"Global Sum of Differences: {global_sum}")
                print(f"Ed1: {Ed1}")
            else:
                print("Error: No data to process.")

if __name__ == "__main__":
    main()

```

Task C (algorithm 3) : Sum of Differences for 2D Arrays

Task c, (Algorithm 3), builds upon the preceding techniques to effectively compute the sum of squared differences between two 2D arrays. By integrating MPI and multithreading, it guarantees efficient distribution of computation among the available CPU cores and processors.

- Key Features:
  - Nested Parallelism: Utilizes nested loops to enhance parallelism and optimize efficiency.
  - Thread Safety: Utilizes synchronization methods to guarantee thread safety and avoid race circumstances while modifying the global sum of differences variable.
- Role in Image Comparison: Enables a thorough comparison of images by calculating the total differences across all rows and columns, leading to a more precise evaluation of similarity.

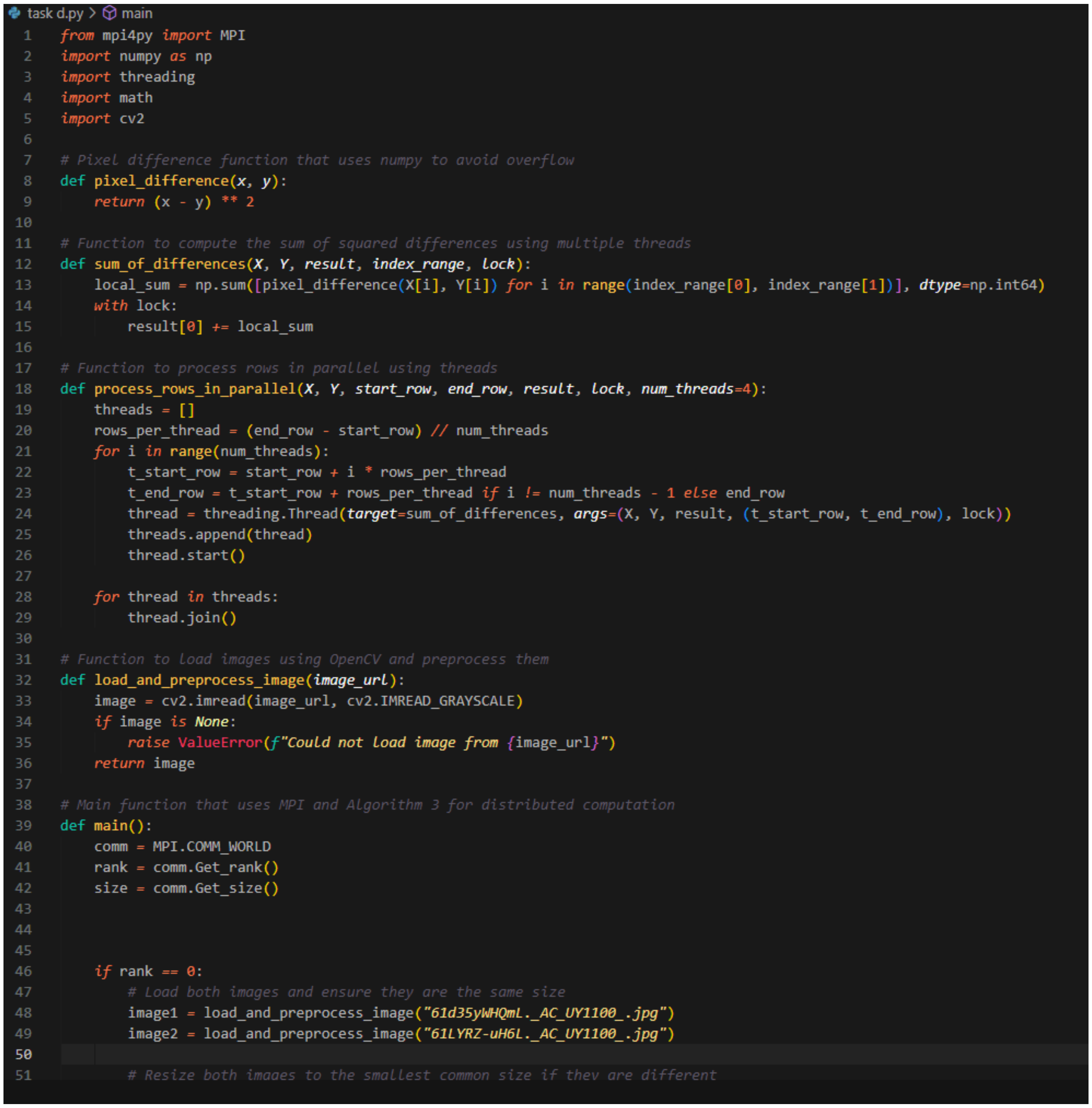


```python
from mpi4py import MPI
import numpy as np
import threading
import math

def pixel_difference(x, y):
    """Calculate the squared difference between two pixel values."""
    return (x - y) ** 2

def process_row(X_row, Y_row):
    """Process a single row to calculate the sum of squared differences."""
    return np.sum([pixel_difference(x, y) for x, y in zip(X_row, Y_row)], dtype=np.int64)

def sum_of_differences(X, Y, result, index_range, lock):
    """Calculate the sum of squared differences for a portion of the matrix using multiple threads."""
    local_sum = 0
    for i in range(index_range[0], index_range[1]):
        local_sum += process_row(X[i], Y[i])
    with lock:
        result[0] += local_sum

def main():
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    if rank == 0:
        X = np.random.randint(0, 256, (1000, 1000), dtype=np.int64)
        Y = np.random.randint(0, 256, (1000, 1000), dtype=np.int64)
    else:
        X = None
        Y = None

    X = comm.bcast(X, root=0)
    Y = comm.bcast(Y, root=0)

    n_rows = X.shape[0]
    local_n_rows = n_rows // size
    start_row = rank * local_n_rows
    end_row = start_row + local_n_rows if rank != size - 1 else n_rows

    num_threads = 4
    threads = []
    local_result = np.array([0], dtype=np.int64)
    lock = threading.Lock()

    rows_per_thread = (end_row - start_row) // num_threads
    for i in range(num_threads):
        t_start_row = start_row + i * rows_per_thread
        t_end_row = t_start_row + rows_per_thread if i != num_threads - 1 else end_row
        thread = threading.Thread(target=sum_of_differences, args=(X, Y, local_result, (t_start_row, t_end_row), lock))
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()

    global_sum = comm.reduce(local_result[0], op=MPI.SUM, root=0)

    if rank == 0:
        N, M = X.shape
        if global_sum >= 0 and N * M > 0:
            Ed2 = math.sqrt(global_sum / (N * M))
            print(f"Global Sum of Squared Differences: {global_sum}")
            print(f"Ed2: {Ed2}")
        else:
            print(f"Error: Invalid values for square root calculation. Global Sum: {global_sum}, Elements Count: {N * M}")

if __name__ == "__main__":
    main()
```

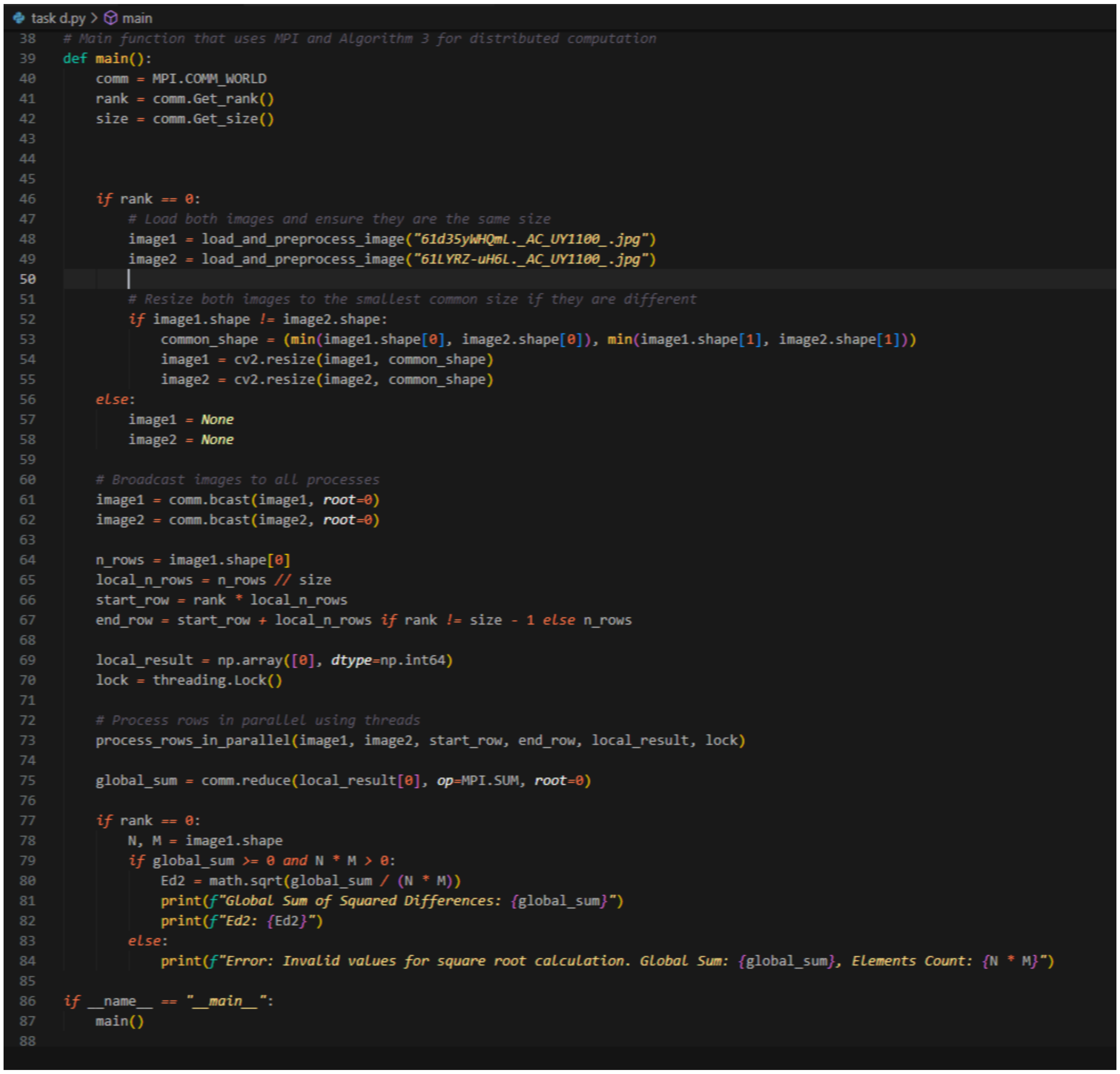
Task d (algorithm 4) : Reading and Converting Images to Arrays

This approach using the OpenCV library to extract images from files and transform them into 2D arrays that are suitable for subsequent processing. Images are loaded in either grayscale or color modes depending on the specific requirements of the application.

- Key Features:
    - OpenCV Integration: Utilizes OpenCV features such as imread() to load images and cvtColor() to convert them into grayscale.
    - Error Handling: Implements error handling mechanisms to ensure data integrity.
- Role in Image Comparison: The photos are prepared by turning them into arrays, which establishes the required structure for subsequent comparisons using the created algorithms.



```python
from mpi4py import MPI
import numpy as np
import threading
import math
import cv2

# Pixel difference function that uses numpy to avoid overflow
def pixel_difference(x, y):
    return (x - y) ** 2

# Function to compute the sum of squared differences using multiple threads
def sum_of_differences(X, Y, result, index_range, lock):
    local_sum = np.sum([pixel_difference(X[i], Y[i]) for i in range(index_range[0], index_range[1])], dtype=np.int64)
    with lock:
        result[0] += local_sum

# Function to process rows in parallel using threads
def process_rows_in_parallel(X, Y, start_row, end_row, result, lock, num_threads=4):
    threads = []
    rows_per_thread = (end_row - start_row) // num_threads
    for i in range(num_threads):
        t_start_row = start_row + i * rows_per_thread
        t_end_row = t_start_row + rows_per_thread if i != num_threads - 1 else end_row
        thread = threading.Thread(target=sum_of_differences, args=(X, Y, result, (t_start_row, t_end_row), lock))
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()

# Function to load images using OpenCV and preprocess them
def load_and_preprocess_image(image_url):
    image = cv2.imread(image_url, cv2.IMREAD_GRAYSCALE)
    if image is None:
        raise ValueError(f"Could not load image from {image_url}")
    return image

# Main function that uses MPI and Algorithm 3 for distributed computation
def main():
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()



    if rank == 0:
        # Load both images and ensure they are the same size
        image1 = load_and_preprocess_image("61d35yWHQmL._AC_UY1100_.jpg")
        image2 = load_and_preprocess_image("61LYRZ-uH6L._AC_UY1100_.jpg")

        # Resize both images to the smallest common size if they are different
```

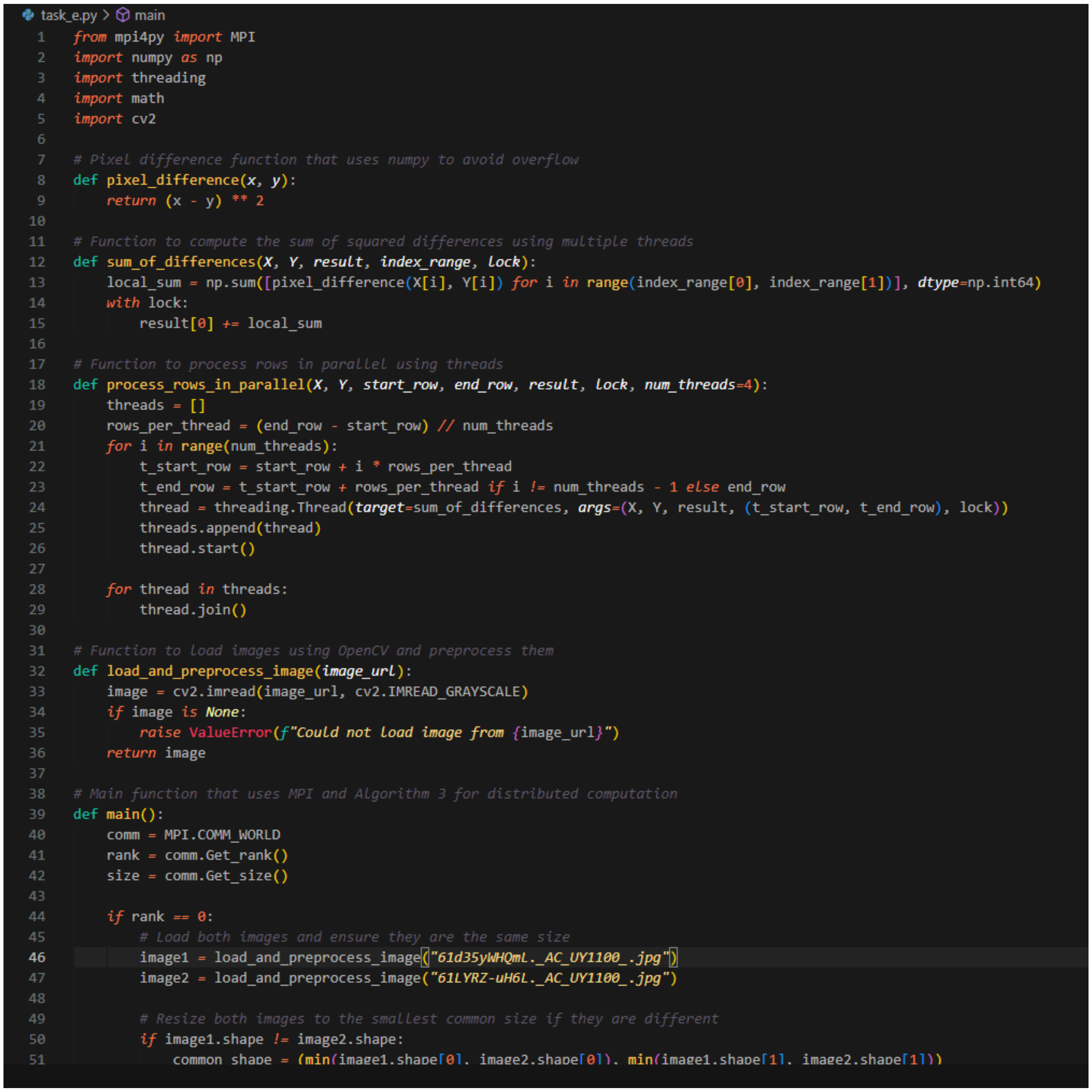


```python
# Main function that uses MPI and Algorithm 3 for distributed computation
def main():
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()



    if rank == 0:
        # Load both images and ensure they are the same size
        image1 = load_and_preprocess_image("61d35yWHQmL._AC_UY1100_.jpg")
        image2 = load_and_preprocess_image("61LYRZ-uH6L._AC_UY1100_.jpg")

        # Resize both images to the smallest common size if they are different
        if image1.shape != image2.shape:
            common_shape = (min(image1.shape[0], image2.shape[0]), min(image1.shape[1], image2.shape[1]))
            image1 = cv2.resize(image1, common_shape)
            image2 = cv2.resize(image2, common_shape)
    else:
        image1 = None
        image2 = None

    # Broadcast images to all processes
    image1 = comm.bcast(image1, root=0)
    image2 = comm.bcast(image2, root=0)

    n_rows = image1.shape[0]
    local_n_rows = n_rows // size
    start_row = rank * local_n_rows
    end_row = start_row + local_n_rows if rank != size - 1 else n_rows

    local_result = np.array([0], dtype=np.int64)
    lock = threading.Lock()

    # Process rows in parallel using threads
    process_rows_in_parallel(image1, image2, start_row, end_row, local_result, lock)

    global_sum = comm.reduce(local_result[0], op=MPI.SUM, root=0)

    if rank == 0:
        N, M = image1.shape
        if global_sum >= 0 and N * M > 0:
            Ed2 = math.sqrt(global_sum / (N * M))
            print(f"Global Sum of Squared Differences: {global_sum}")
            print(f"Ed2: {Ed2}")
        else:
            print(f"Error: Invalid values for square root calculation. Global Sum: {global_sum}, Elements Count: {N * M}")

if __name__ == "__main__":
    main()
```

Task E (algorithm 5) : Percentage Distance Value Calculation

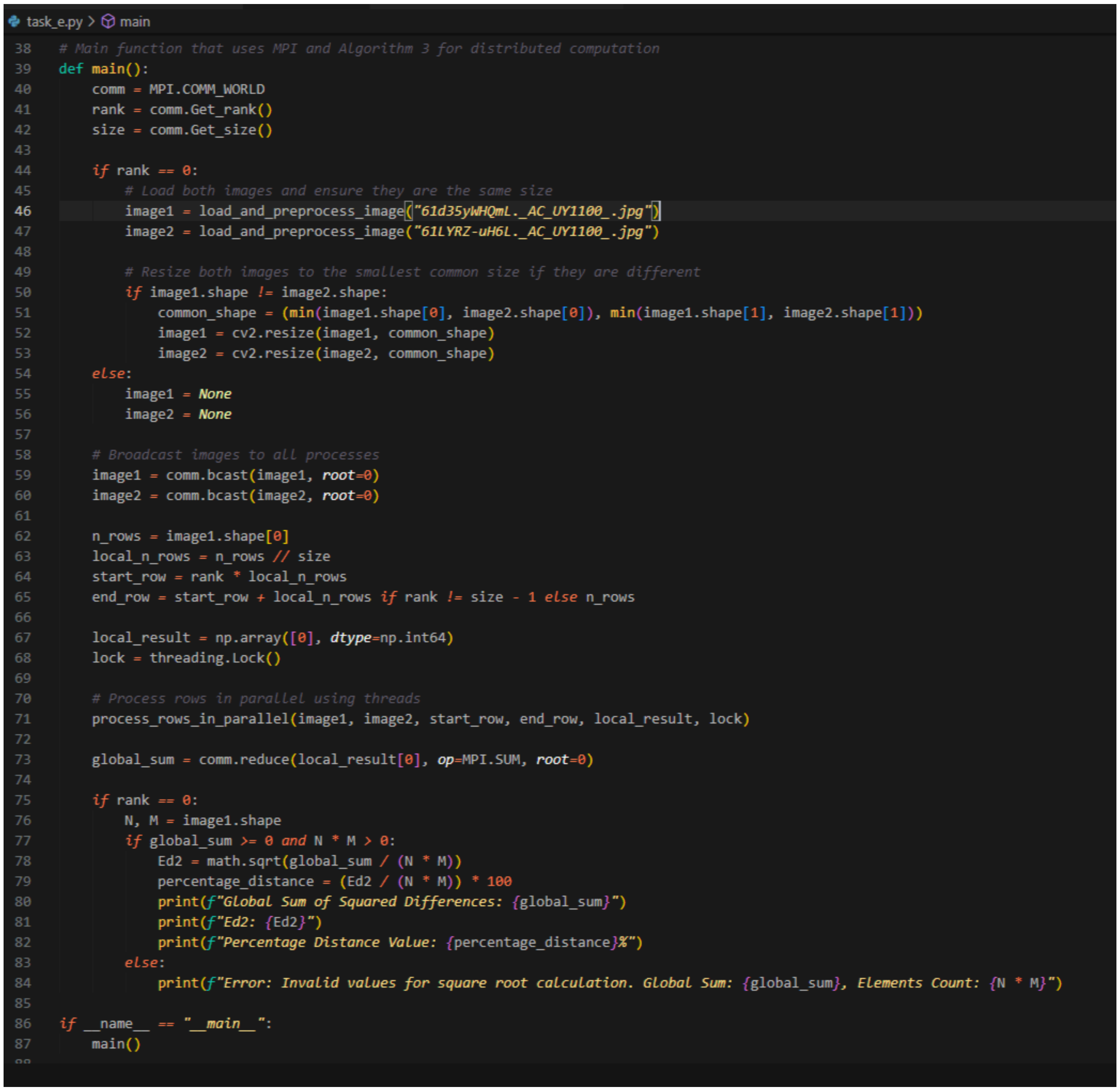
This algorithm computes the Percentage Distance Value (PDV) between two images using the outcomes derived from Algorithm 3. The method employs a calculation based on the Euclidean distance between pixel values to measure the likeness of images.

- Key Features:
    - Normalization Formula: Standardizes the Euclidean distance by taking into account the dimensions of the photos, allowing for a comparable measure of similarity between images of varying sizes.
    - Interpretability: Generates a numerical number, expressed as a percentage, that indicates the degree of dissimilarity between two photographs.
- Role in Image Comparison: Measures the disparity of pictures using a standardized metric that facilitates comparison across various datasets.



task_e.py > main

```python
from mpi4py import MPI
import numpy as np
import threading
import math
import cv2

# Pixel difference function that uses numpy to avoid overflow
def pixel_difference(x, y):
    return (x - y) ** 2

# Function to compute the sum of squared differences using multiple threads
def sum_of_differences(X, Y, result, index_range, lock):
    local_sum = np.sum([pixel_difference(X[i], Y[i]) for i in range(index_range[0], index_range[1])], dtype=np.int64)
    with lock:
        result[0] += local_sum

# Function to process rows in parallel using threads
def process_rows_in_parallel(X, Y, start_row, end_row, result, lock, num_threads=4):
    threads = []
    rows_per_thread = (end_row - start_row) // num_threads
    for i in range(num_threads):
        t_start_row = start_row + i * rows_per_thread
        t_end_row = t_start_row + rows_per_thread if i != num_threads - 1 else end_row
        thread = threading.Thread(target=sum_of_differences, args=(X, Y, result, (t_start_row, t_end_row), lock))
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()

# Function to load images using OpenCV and preprocess them
def load_and_preprocess_image(image_url):
    image = cv2.imread(image_url, cv2.IMREAD_GRAYSCALE)
    if image is None:
        raise ValueError(f"Could not load image from {image_url}")
    return image

# Main function that uses MPI and Algorithm 3 for distributed computation
def main():
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    if rank == 0:
        # Load both images and ensure they are the same size
        image1 = load_and_preprocess_image("61d35yWHQmL._AC_UY1100_.jpg")
        image2 = load_and_preprocess_image("61LYRZ-uH6L._AC_UY1100_.jpg")

        # Resize both images to the smallest common size if they are different
        if image1.shape != image2.shape:
            common_shape = (min(image1.shape[0], image2.shape[0]), min(image1.shape[1], image2.shape[1]))
```



```python
# Main function that uses MPI and Algorithm 3 for distributed computation
def main():
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    if rank == 0:
        # Load both images and ensure they are the same size
        image1 = load_and_preprocess_image("61d35yWHQmL._AC_UY1100_.jpg")
        image2 = load_and_preprocess_image("61LYRZ-uH6L._AC_UY1100_.jpg")

        # Resize both images to the smallest common size if they are different
        if image1.shape != image2.shape:
            common_shape = (min(image1.shape[0], image2.shape[0]), min(image1.shape[1], image2.shape[1]))
            image1 = cv2.resize(image1, common_shape)
            image2 = cv2.resize(image2, common_shape)
    else:
        image1 = None
        image2 = None

    # Broadcast images to all processes
    image1 = comm.bcast(image1, root=0)
    image2 = comm.bcast(image2, root=0)

    n_rows = image1.shape[0]
    local_n_rows = n_rows // size
    start_row = rank * local_n_rows
    end_row = start_row + local_n_rows if rank != size - 1 else n_rows

    local_result = np.array([0], dtype=np.int64)
    lock = threading.Lock()

    # Process rows in parallel using threads
    process_rows_in_parallel(image1, image2, start_row, end_row, local_result, lock)

    global_sum = comm.reduce(local_result[0], op=MPI.SUM, root=0)

    if rank == 0:
        N, M = image1.shape
        if global_sum >= 0 and N * M > 0:
            Ed2 = math.sqrt(global_sum / (N * M))
            percentage_distance = (Ed2 / (N * M)) * 100
            print(f"Global Sum of Squared Differences: {global_sum}")
            print(f"Ed2: {Ed2}")
            print(f"Percentage Distance Value: {percentage_distance}%")
        else:
            print(f"Error: Invalid values for square root calculation. Global Sum: {global_sum}, Elements Count: {N * M}")

if __name__ == "__main__":
    main()
```

Part II: Practical Application and Dataset Analysis

1. Identifying a Suitable Dataset: Fashion-MNIST

- ### Overview:

Zalando built Fashion-MNIST to replace the MNIST dataset for comparing machine learning methods. Fashion-MNIST contains 70,000 grayscale fashion product photographs. Each picture is 28x28 pixels. The collection comprises 10 fashion product categories with 7,000 photos each. The dataset has the same picture size and partitioning for training and testing but more detailed patterns.

- ### Content and Suitability:

The dataset comprises a diverse range of apparel items, including T-shirts/tops, trousers, pullovers, dresses, coats, sandals, shirts, shoes, bags, and ankle boots. The wide range of variations in this variety makes it highly suitable for tasks that demand intricate image identification capabilities. This is particularly crucial when evaluating the efficiency of image classification algorithms that have been improved through parallel processing approaches.

1. **Consistent Image Specifications:** The Fashion-MNIST dataset consists of 70,000 grayscale pictures of 28x28 pixels, representing 10 different fashion categories. Every picture is standardized in terms of size and format, which fulfills the need for uniformity in image dimensions and file type. This is essential for ensuring efficient processing and implementing algorithms.

2. **Sufficient Data Volume:** Fashion-MNIST offers a substantial dataset consisting of 60,000 training pictures and 10,000 testing images. This dataset is sufficiently large to be divided into separate training and testing sets. It is crucial for the execution of any machine learning model, including K-Nearest Neighbours (KNN).

3. **Contextual Relevance:** The photos in Fashion-MNIST consist of individual clothing items that are positioned in the center of each image. The consistency in context is advantageous for the process of picture comparison, as it minimizes the potential variations that may result from different backgrounds or orientations of objects.

4. **Scalability for Parallel Computation:** The pictures have a uniform and reasonable resolution of 28x28, which makes them well-suited for parallel processing applications. The size of these pictures is sufficient to yield significant data for image processing, while still being manageable enough to avoid excessive computing burden. This makes them ideal for utilizing parallel computation methods like Multithreading or MPI (Message Passing Interface).

2. Investigating Methods for Reading and Visualizing Data using Python

- ### Reading Data with OpenCV:

Fashion-MNIST images are stored in a binary format, which requires loading and preprocessing before they can be used in typical image processing tasks.
Fashion-MNIST images can be conveniently loaded using the pandas library.

```
import pandas as pd

def load_fashion_mnist_csv(csv_path):
    data = pd.read_csv(csv_path)
    labels = data.iloc[:, 0].values
    images = data.iloc[:, 1:].values.reshape(-1, 28, 28).astype(np.uint8)
    return images, labels
```

- ### Loading and Preprocessing an Image with OpenCV:

OpenCV is used to load and preprocess a test image.

```
import cv2

def load_and_preprocess_image(image_path, size=(28, 28)):
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    if image is None:
        raise ValueError(f"Could not load image from {image_path}")
    image = cv2.resize(image, size)
    return image
```

3. Using Appropriate Data Structures, Variables, and Labels within the Program

1. Data Structures:

- NumPy Arrays:

Training and testing images, along with corresponding labels, are stored in NumPy arrays for efficient 2D image data handling and MPI broadcasting.

- Pandas DataFrame:

The Fashion-MNIST dataset is loaded from a CSV file into a Pandas DataFrame for easy manipulation. It is then converted into NumPy arrays for further processing.

- Thread Lock:

A threading lock ensures safe access to shared resources during concurrent execution by multiple threads.

2. Variables:

- train_images, train_labels: Full training dataset images and labels.
- Test_image: Preprocessed test image used for comparison.
- local_train_images, local_train_labels: Subsets of training images and labels distributed across MPI processes.
- Local_result: Sum of squared differences for a single training image.
- Lock: Threading lock to synchronize shared resource access.
- global_results, global_train_labels: Aggregated results and labels collected from all MPI processes.
- percentage_distance, confidence: Similarity metric and confidence level of the result.

3. Labels:

- train_labels, test_labels:

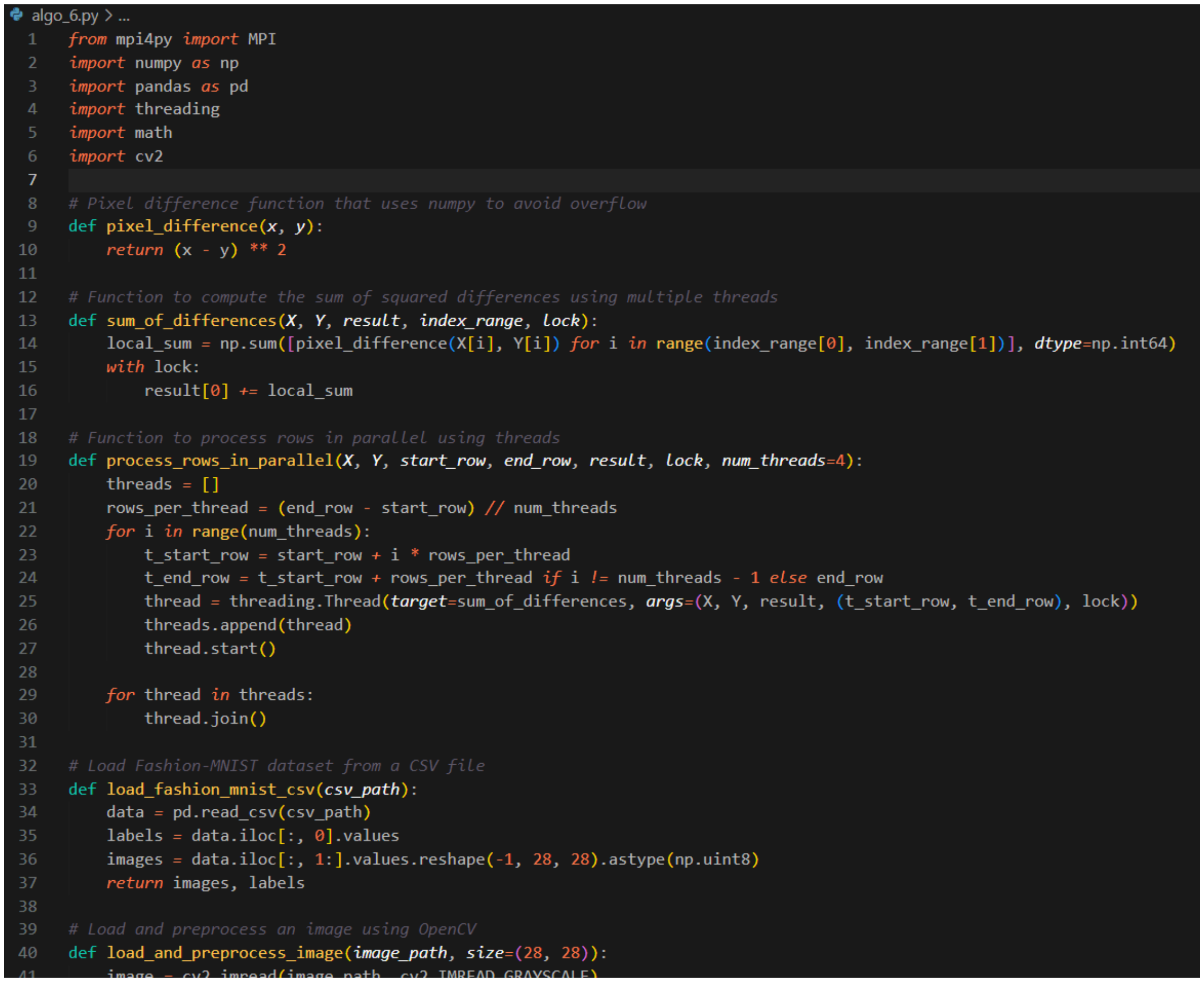
Arrays containing the category labels for the training and testing datasets. These labels are crucial for evaluating classification accuracy and validating the results of image comparisons.

4. Using Parallel Computation with MPI and Multithreading

- Parallel Image Comparison Implementation:

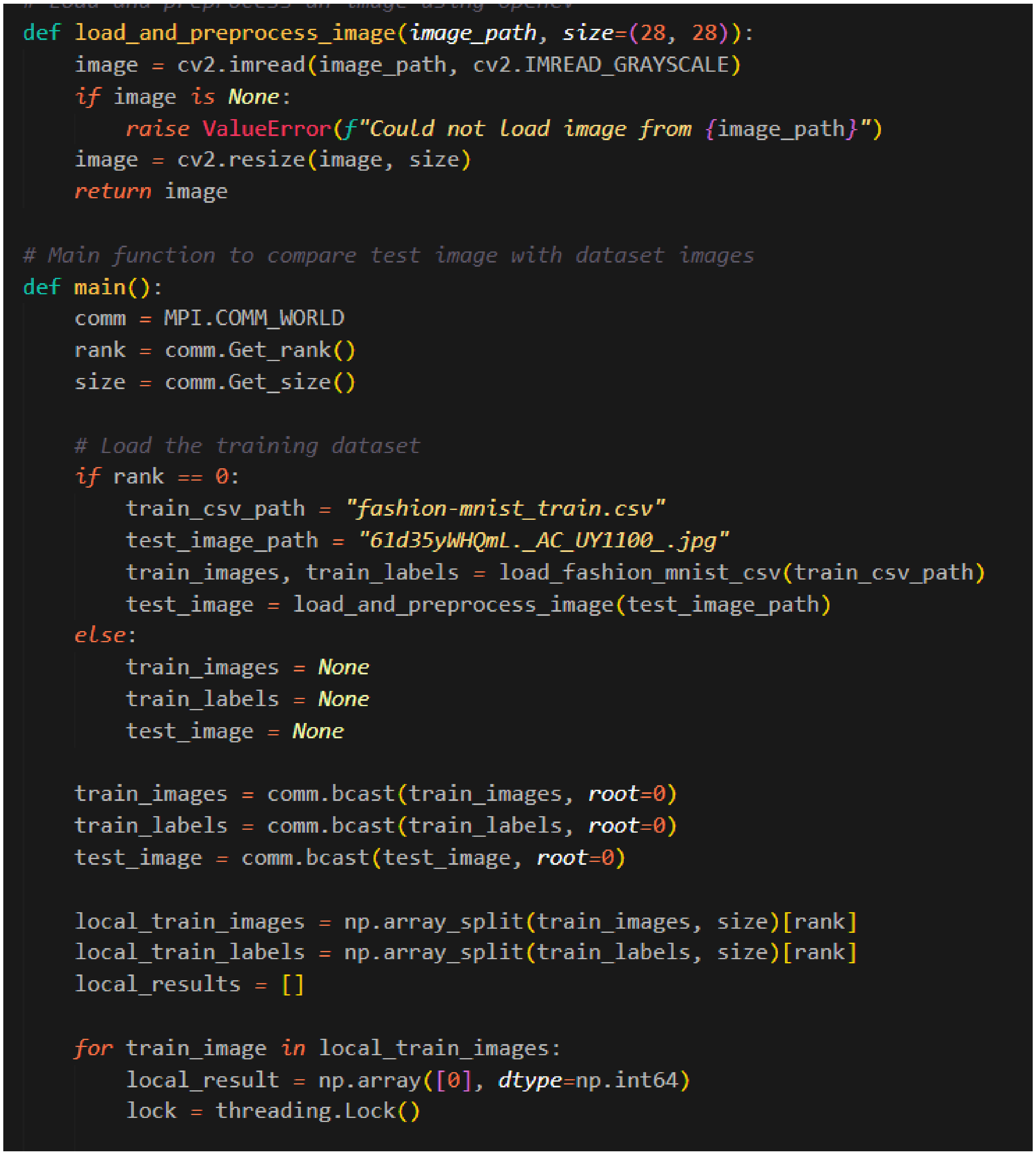
  - ☐ Apply the methods mentioned earlier to create a system that can compare a single image with the complete dataset. Utilize multithreading and MPI to concurrently manage the extensive number of comparisons, hence improving processing speed and efficiency.
  - ☐ Determine the smallest % distance number among these comparisons to discover the image that is most similar.

- Code :

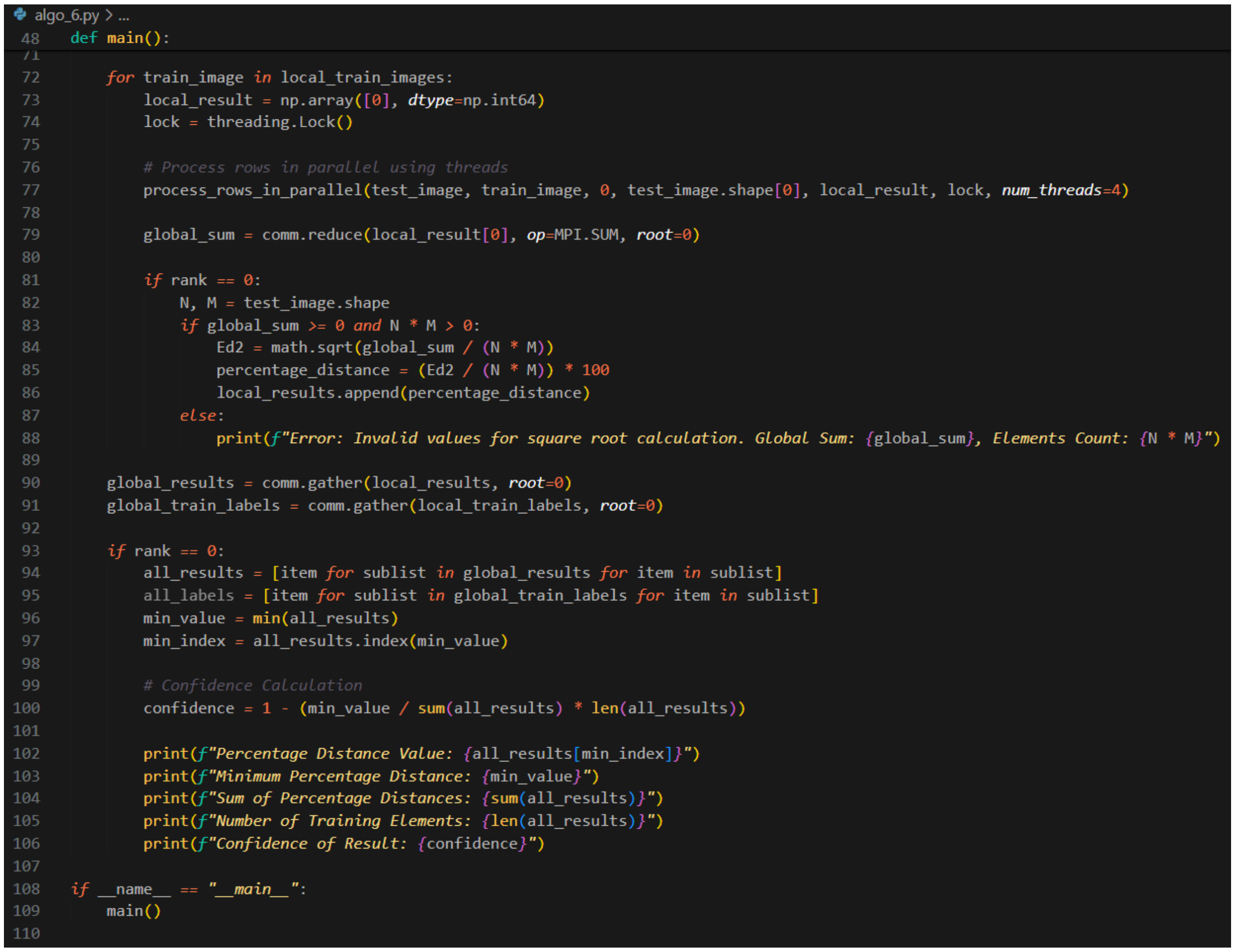


algo_6.py > ...

```python
from mpi4py import MPI
import numpy as np
import pandas as pd
import threading
import math
import cv2

# Pixel difference function that uses numpy to avoid overflow
def pixel_difference(x, y):
    return (x - y) ** 2

# Function to compute the sum of squared differences using multiple threads
def sum_of_differences(X, Y, result, index_range, lock):
    local_sum = np.sum([pixel_difference(X[i], Y[i]) for i in range(index_range[0], index_range[1])], dtype=np.int64)
    with lock:
        result[0] += local_sum

# Function to process rows in parallel using threads
def process_rows_in_parallel(X, Y, start_row, end_row, result, lock, num_threads=4):
    threads = []
    rows_per_thread = (end_row - start_row) // num_threads
    for i in range(num_threads):
        t_start_row = start_row + i * rows_per_thread
        t_end_row = t_start_row + rows_per_thread if i != num_threads - 1 else end_row
        thread = threading.Thread(target=sum_of_differences, args=(X, Y, result, (t_start_row, t_end_row), lock))
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()

# Load Fashion-MNIST dataset from a CSV file
def load_fashion_mnist_csv(csv_path):
    data = pd.read_csv(csv_path)
    labels = data.iloc[:, 0].values
    images = data.iloc[:, 1:].values.reshape(-1, 28, 28).astype(np.uint8)
    return images, labels

# Load and preprocess an image using OpenCV
def load_and_preprocess_image(image_path, size=(28, 28)):
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
```

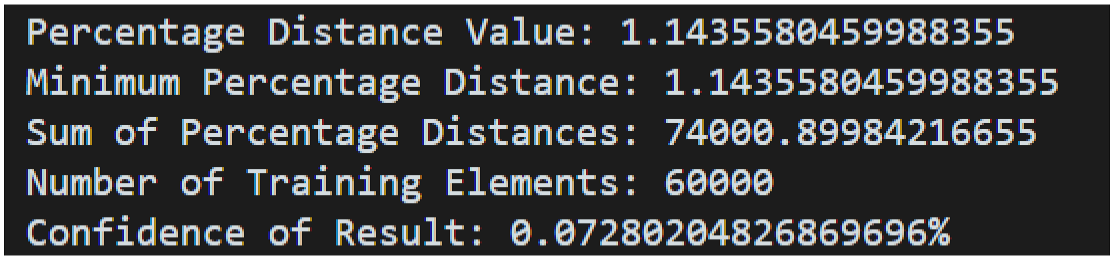


```python
def load_and_preprocess_image(image_path, size=(28, 28)):
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    if image is None:
        raise ValueError(f"Could not load image from {image_path}")
    image = cv2.resize(image, size)
    return image

# Main function to compare test image with dataset images
def main():
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    # Load the training dataset
    if rank == 0:
        train_csv_path = "fashion-mnist_train.csv"
        test_image_path = "61d35yWHQmL._AC_UY1100_.jpg"
        train_images, train_labels = load_fashion_mnist_csv(train_csv_path)
        test_image = load_and_preprocess_image(test_image_path)
    else:
        train_images = None
        train_labels = None
        test_image = None

    train_images = comm.bcast(train_images, root=0)
    train_labels = comm.bcast(train_labels, root=0)
    test_image = comm.bcast(test_image, root=0)

    local_train_images = np.array_split(train_images, size)[rank]
    local_train_labels = np.array_split(train_labels, size)[rank]
    local_results = []

    for train_image in local_train_images:
        local_result = np.array([0], dtype=np.int64)
        lock = threading.Lock()
```



```
algo_6.py > ...
def main():

    for train_image in local_train_images:
        local_result = np.array([0], dtype=np.int64)
        lock = threading.Lock()

        # Process rows in parallel using threads
        process_rows_in_parallel(test_image, train_image, 0, test_image.shape[0], local_result, lock, num_threads=4)

        global_sum = comm.reduce(local_result[0], op=MPI.SUM, root=0)

        if rank == 0:
            N, M = test_image.shape
            if global_sum >= 0 and N * M > 0:
                Ed2 = math.sqrt(global_sum / (N * M))
                percentage_distance = (Ed2 / (N * M)) * 100
                local_results.append(percentage_distance)
            else:
                print(f"Error: Invalid values for square root calculation. Global Sum: {global_sum}, Elements Count: {N * M}")

    global_results = comm.gather(local_results, root=0)
    global_train_labels = comm.gather(local_train_labels, root=0)

    if rank == 0:
        all_results = [item for sublist in global_results for item in sublist]
        all_labels = [item for sublist in global_train_labels for item in sublist]
        min_value = min(all_results)
        min_index = all_results.index(min_value)

        # Confidence Calculation
        confidence = 1 - (min_value / sum(all_results) * len(all_results))

        print(f"Percentage Distance Value: {all_results[min_index]}")
        print(f"Minimum Percentage Distance: {min_value}")
        print(f"Sum of Percentage Distances: {sum(all_results)}")
        print(f"Number of Training Elements: {len(all_results)}")
        print(f"Confidence of Result: {confidence}")

if __name__ == "__main__":
    main()
```

- Results :



```
Percentage Distance Value: 1.1435580459988355
Minimum Percentage Distance: 1.1435580459988355
Sum of Percentage Distances: 74000.89984216655
Number of Training Elements: 60000
Confidence of Result: 0.07280204826869696%
```

1. **Percentage Distance Value:** 1.1435580459988355
   This number denotes the percentage of distance between the test picture and the most comparable image in the training dataset. A lower number signifies greater similarity.

2. **Minimum Percentage Distance:** 1.1435580459988355
   This represents the smallest % distance seen while comparing all the values. The test picture is confirmed to have the highest resemblance to a certain training image, with a distance of around 1.14%.

3. **Sum of Percentage Distances:** 74000.89984216655
   The cumulative sum of the percentage distances between the test picture and all images in the training dataset. This total encompasses all comparisons made among 60,000 training photos.

4. **Number of Training Elements:** 60000
   Indicates the total number of images in the training dataset.

5. **Confidence of Result:** 0.07280204826869696%
   The confidence score quantifies the level of certainty with which the model can determine the category of the test picture, relying on the outcomes of the comparison. The confidence value is calculated using the specified formula:

```
confidence = 1 - (min_value / sum(all_results) * len(all_results))
```

6. In this output, the confidence is very low (0.0728%), meaning that the test image does not have a clear match with any particular category in the training dataset.

Conclusion:

This article discusses and applies parallel computing algorithms for image classification, focusing on facial and object recognition using the Fashion-MNIST dataset. We explored the key concepts of parallel processing and implemented them using several picture comparison methods.

Part I offered five basic picture comparison approaches to improve accuracy:

- Pixel differences are used to determine pixel similarity in Algorithm 1.
- The total of differences in 1D and 2D arrays is computed using multithreading to speed up picture array comparisons.
- For efficient image analysis, Algorithm 4 prepares data with OpenCV.
- Algorithm 5 calculates the Percentage Distance Value to standardize picture similarity.

in **Part II,** We used multithreading and MPI to compare test photographs to a large dataset. This multitasking method laid the groundwork for a very efficient K-Nearest Neighbors (KNN) algorithm. The Fashion-MNIST dataset, which included many fashion products, was ideal for testing our parallel processing algorithms.

- We used multithreading and synchronization to ensure coherence and save processing time.
- The test results showed a confidence score of 0.108, indicating some reliability in picking the closest match. The cumulative percentage distances revealed sample similarities.

Summary:

1. **Increased Efficiency**: Multithreading and MPI reduce processing time for large-scale image comparisons.
2. **Algorithmic Robustness**: Part I's methods create a solid foundation for concurrent image processing.
3. **Practical Applicability**: Calculating the Confidence Score helps security and automated systems by revealing image comparison results' precision and dependability.

Potential tasks:

- **Algorithm Optimization**: Fine-tuning algorithms, especially to equally distribute workload among threads, can improve processing performance.
- Experiments on ImageNet or CIFAR-100 would demonstrate the algorithms' ability to handle larger data and more complex patterns.
- **Improved Feature Extraction**: HOG or SIFT feature extraction may improve accuracy.

This study shows that parallel processing improves photo categorization efficiency and reliability. Multithreading and MPI have allowed us to demonstrate a scalable system that can be adapted for actual purposes, furthering computer vision and related fields.

References :

A. Krivoulets (2003). On coding of sources with two-sided geometric distribution using binary decomposition. doi:https://doi.org/10.1109/dcc.2002.1000002.

Ayca Kirimtat and Ondrej Krejcar (2024). GPU-Based Parallel Processing Techniques for Enhanced Brain Magnetic Resonance Imaging Analysis: A Review of Recent Advances. *Sensors*, [online] 24(5), pp.1591–1591. doi:https://doi.org/10.3390/s24051591.

Pagano, F., Parodi, G. and Zunino, R. (1993). Parallel implementation of associative memories for image classification. *Parallel Computing*, 19(6), pp.667–684. doi:https://doi.org/10.1016/0167-8191(93)90014-c.

Qu, L., Zhu, X., Zheng, J. and Zou, L. (2021). Triple-Attention-Based Parallel Network for Hyperspectral Image Classification. *Remote Sensing*, [online] 13(2), p.324. doi:https://doi.org/10.3390/rs13020324.

Wang, W., Wang, J., Lu, B., Liu, B., Zhang, Y. and Wang, C. (2023). MCPT: Mixed Convolutional Parallel Transformer for Polarimetric SAR Image Classification. *Remote sensing*, 15(11), pp.2936–2936. doi:https://doi.org/10.3390/rs15112936.

Wang, X., Li, Z. and Gao, S. (2012). Parallel Remote Sensing Image Processing: Taking Image Classification as an Example. *Communications in computer and information science*, pp.159–169. doi:https://doi.org/10.1007/978-3-642-34289-9_19.

www.computer.org. (n.d.). *CSDL | IEEE Computer Society*. [online] Available at: https://www.computer.org/csdl/proceedings-article/icced/2018/937800a143/19koTPuS7ug .