XJTLU Entrepreneur College (Taicang) Cover Sheet

Module code and Title	**DTS202TC Foundation of Parallel Computing**
School Title	**School of AI and Advanced Computing**
Assignment Title	**Individual Assessment 2**
Submission Deadline	**Saturday Dec. 23rd, 2023 @ 11:59pm**
Final Word Count	**N/A**
If you agree to let the university use your work anonymously for teaching and learning purposes, please type **"yes"** here.	**yes**

I certify that I have read and understood the University's Policy for dealing with Plagiarism, Collusion and the Fabrication of Data (available on Learning Mall Online). With reference to this policy I certify that:

- My work does not contain any instances of plagiarism and/or collusion.
- My work does not contain any fabricated data.

By uploading my assignment onto Learning Mall Online, I formally declare that all of the above information is true to the best of my knowledge and belief.

Scoring – For Tutor Use	
Student ID	**2142457**

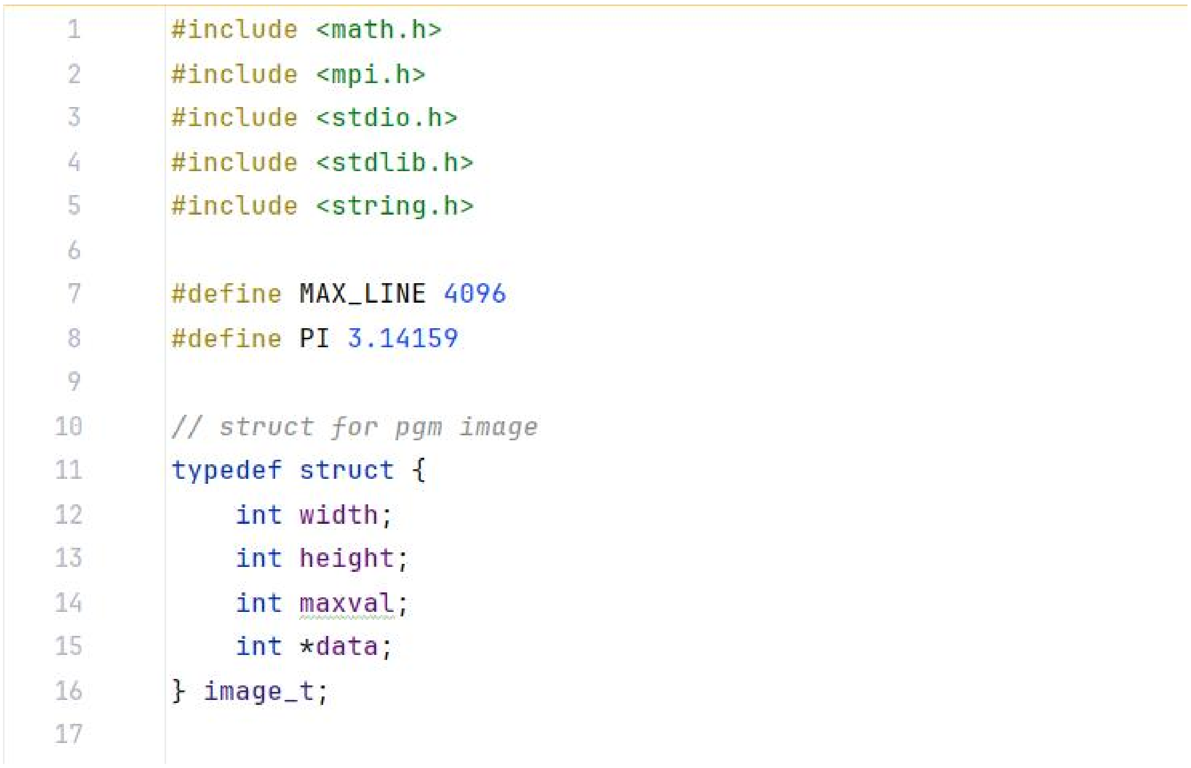
Stage of Marking	**Marker Code**	**Learning Outcomes Achieved (F/P/M/D) (please modify as appropriate)**			**Final Score**
		A	**B**	**C**	
1st Marker – red pen					
Moderation – green pen	**IM Initials**	The original mark has been accepted by the moderator (please circle as appropriate):			Y / N
		Data entry and score calculation have been checked by another tutor (please circle):			Y
2nd Marker if needed – green pen					

For Academic Office Use			**Possible Academic Infringement (please tick as appropriate)**	
Date Received	**Days late**	**Late Penalty**	☐ **Category A**	Total Academic Infringement Penalty (A,B, C, D, E, Please modify where necessary) ________________
			☐ **Category B**	
			☐ **Category C**	
			☐ **Category D**	
			☐ **Category E**	

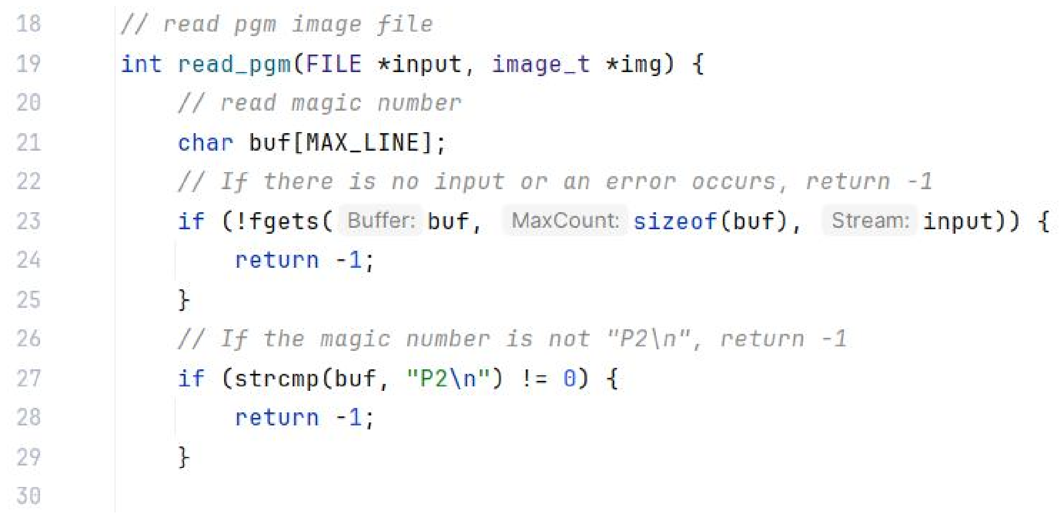
1. Parallel implementations

I implement two difference parallel programs, using MPI and OpenMP.

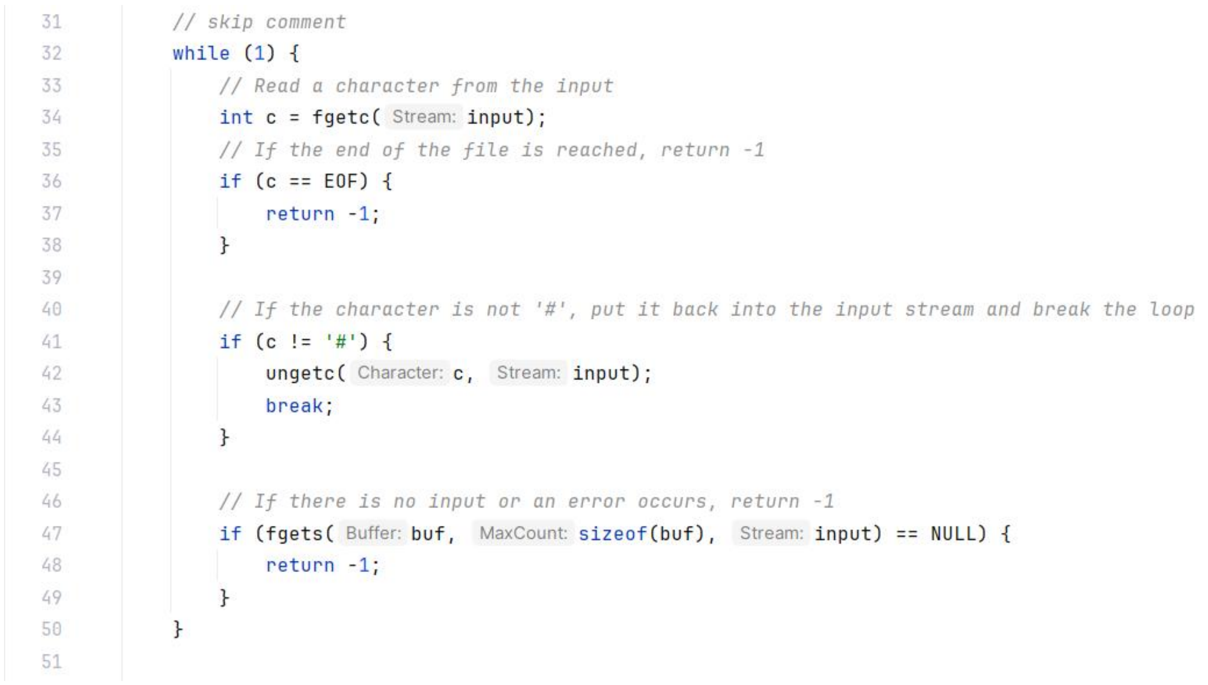
MPI Code



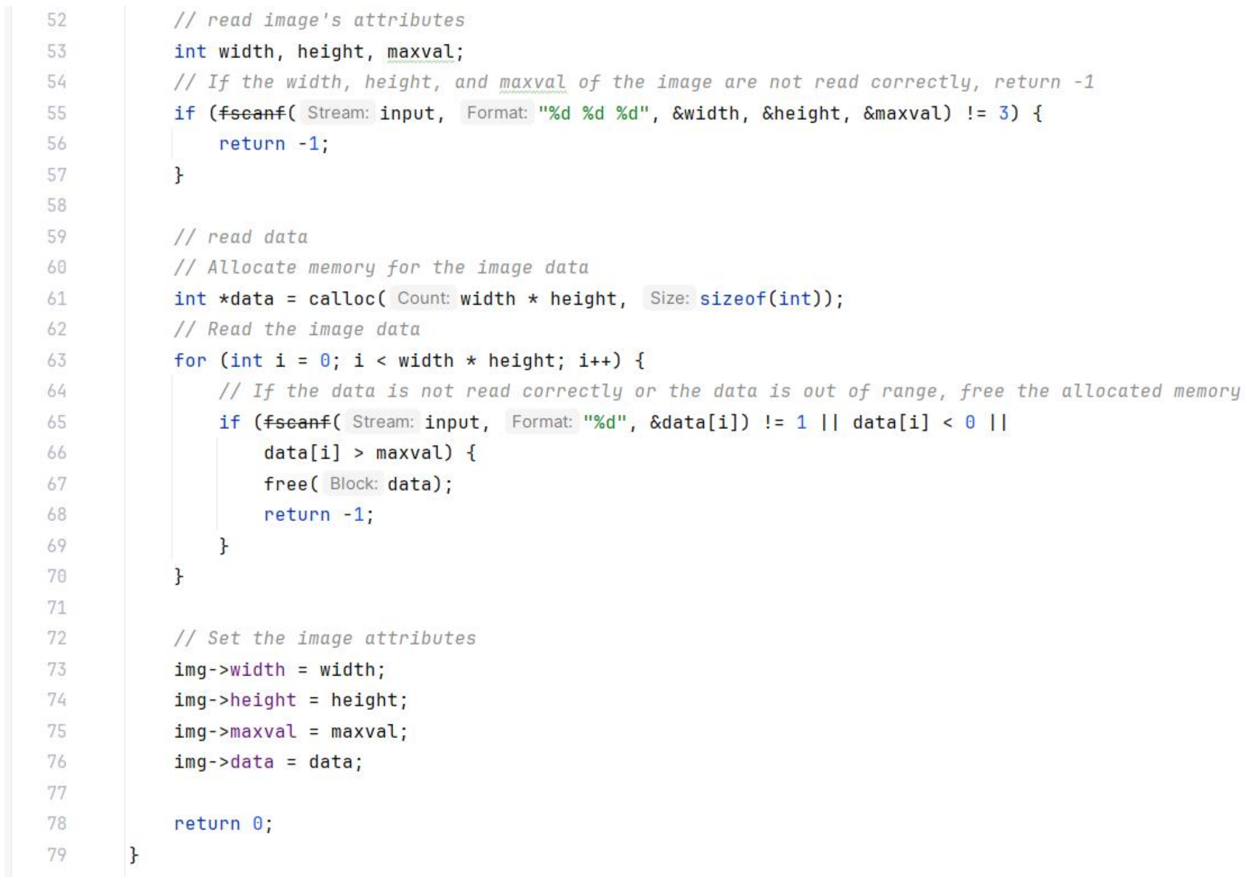
```c
#include <math.h>
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_LINE 4096
#define PI 3.14159

// struct for pgm image
typedef struct {
    int width;
    int height;
    int maxval;
    int *data;
} image_t;

```



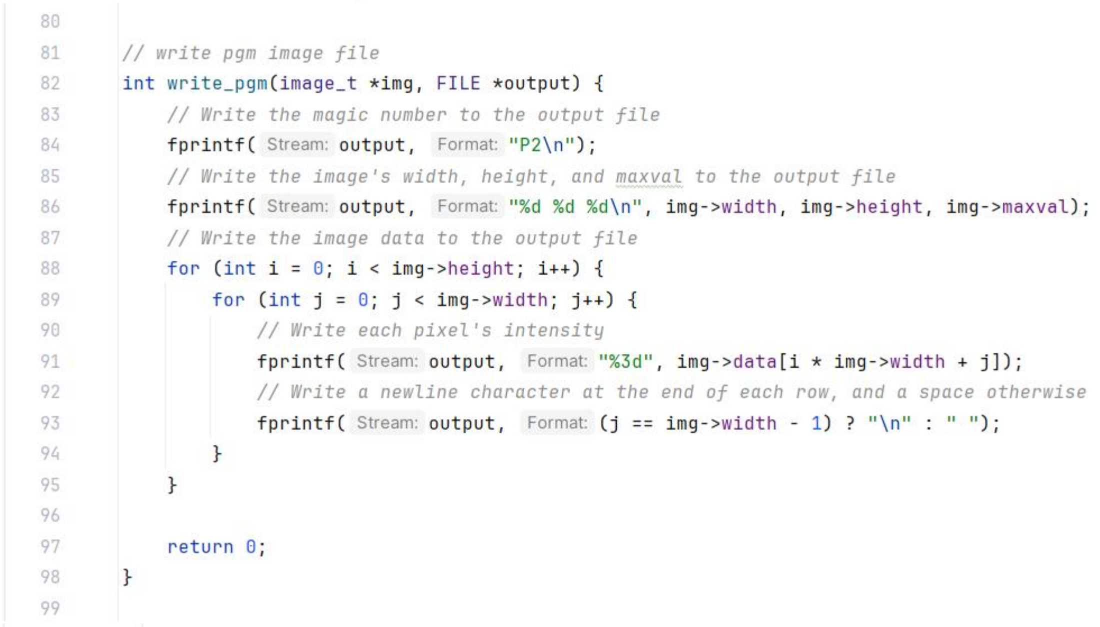
```c
// read pgm image file
int read_pgm(FILE *input, image_t *img) {
    // read magic number
    char buf[MAX_LINE];
    // If there is no input or an error occurs, return -1
    if (!fgets(buf, sizeof(buf), input)) {
        return -1;
    }
    // If the magic number is not "P2\n", return -1
    if (strcmp(buf, "P2\n") != 0) {
        return -1;
    }

```



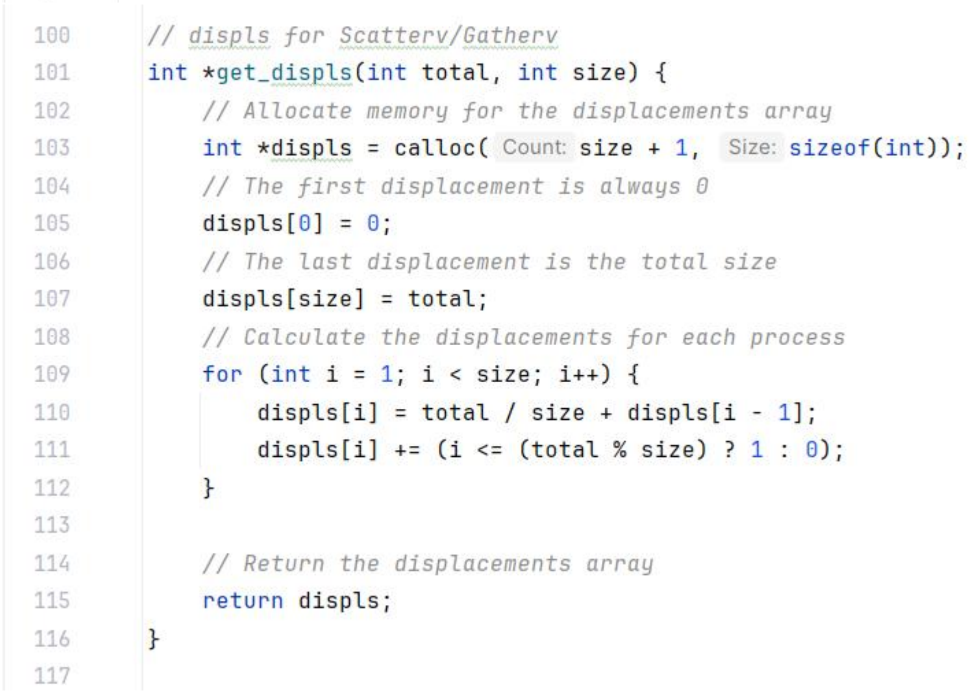
```c
    // skip comment
    while (1) {
        // Read a character from the input
        int c = fgetc( Stream: input);
        // If the end of the file is reached, return -1
        if (c == EOF) {
            return -1;
        }

        // If the character is not '#', put it back into the input stream and break the loop
        if (c != '#') {
            ungetc( Character: c,  Stream: input);
            break;
        }

        // If there is no input or an error occurs, return -1
        if (fgets( Buffer: buf,  MaxCount: sizeof(buf),  Stream: input) == NULL) {
            return -1;
        }
    }

```



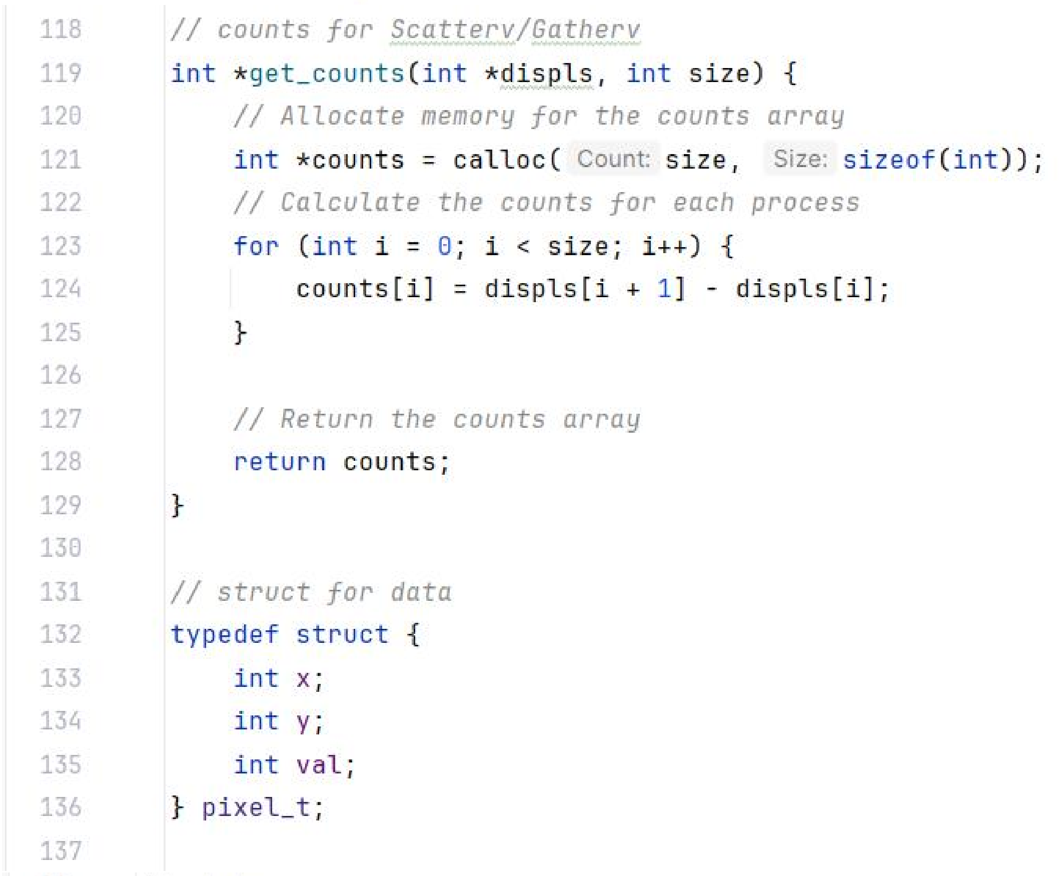
```c
    // read image's attributes
    int width, height, maxval;
    // If the width, height, and maxval of the image are not read correctly, return -1
    if (~~fscanf~~( Stream: input,  Format: "%d %d %d", &width, &height, &maxval) != 3) {
        return -1;
    }

    // read data
    // Allocate memory for the image data
    int *data = calloc( Count: width * height,  Size: sizeof(int));
    // Read the image data
    for (int i = 0; i < width * height; i++) {
        // If the data is not read correctly or the data is out of range, free the allocated memory
        if (~~fscanf~~( Stream: input,  Format: "%d", &data[i]) != 1 || data[i] < 0 ||
            data[i] > maxval) {
            free( Block: data);
            return -1;
        }
    }

    // Set the image attributes
    img->width = width;
    img->height = height;
    img->maxval = maxval;
    img->data = data;

    return 0;
}
```



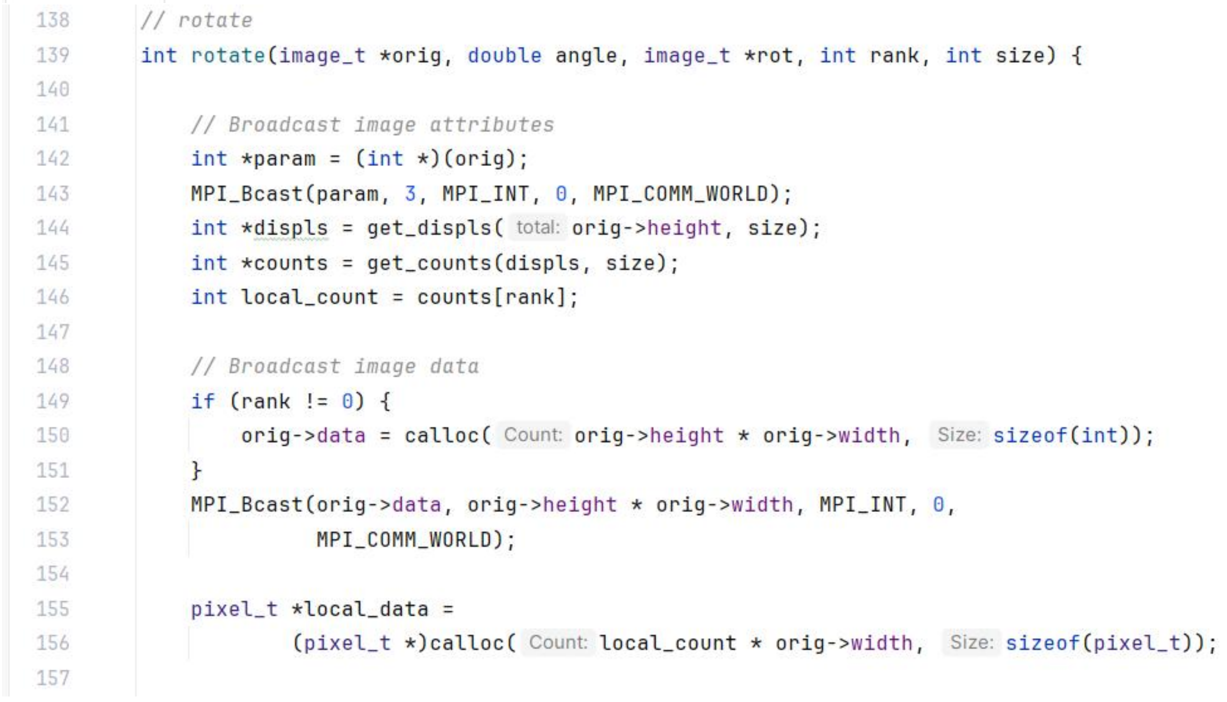
```c
// write pgm image file
int write_pgm(image_t *img, FILE *output) {
    // Write the magic number to the output file
    fprintf( Stream: output,  Format: "P2\n");
    // Write the image's width, height, and maxval to the output file
    fprintf( Stream: output,  Format: "%d %d %d\n", img->width, img->height, img->maxval);
    // Write the image data to the output file
    for (int i = 0; i < img->height; i++) {
        for (int j = 0; j < img->width; j++) {
            // Write each pixel's intensity
            fprintf( Stream: output,  Format: "%3d", img->data[i * img->width + j]);
            // Write a newline character at the end of each row, and a space otherwise
            fprintf( Stream: output,  Format: (j == img->width - 1) ? "\n" : " ");
        }
    }

    return 0;
}
```



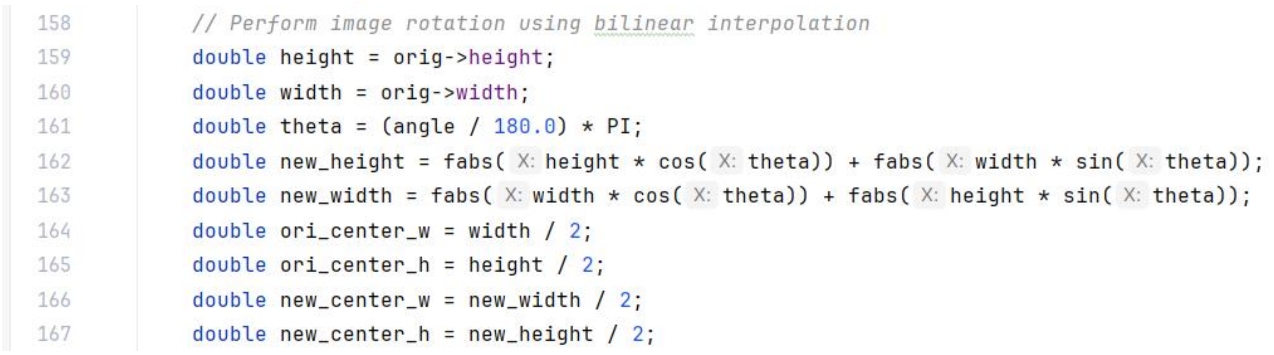
```c
// displs for Scatterv/Gatherv
int *get_displs(int total, int size) {
    // Allocate memory for the displacements array
    int *displs = calloc( Count: size + 1,  Size: sizeof(int));
    // The first displacement is always 0
    displs[0] = 0;
    // The last displacement is the total size
    displs[size] = total;
    // Calculate the displacements for each process
    for (int i = 1; i < size; i++) {
        displs[i] = total / size + displs[i - 1];
        displs[i] += (i <= (total % size) ? 1 : 0);
    }

    // Return the displacements array
    return displs;
}
```



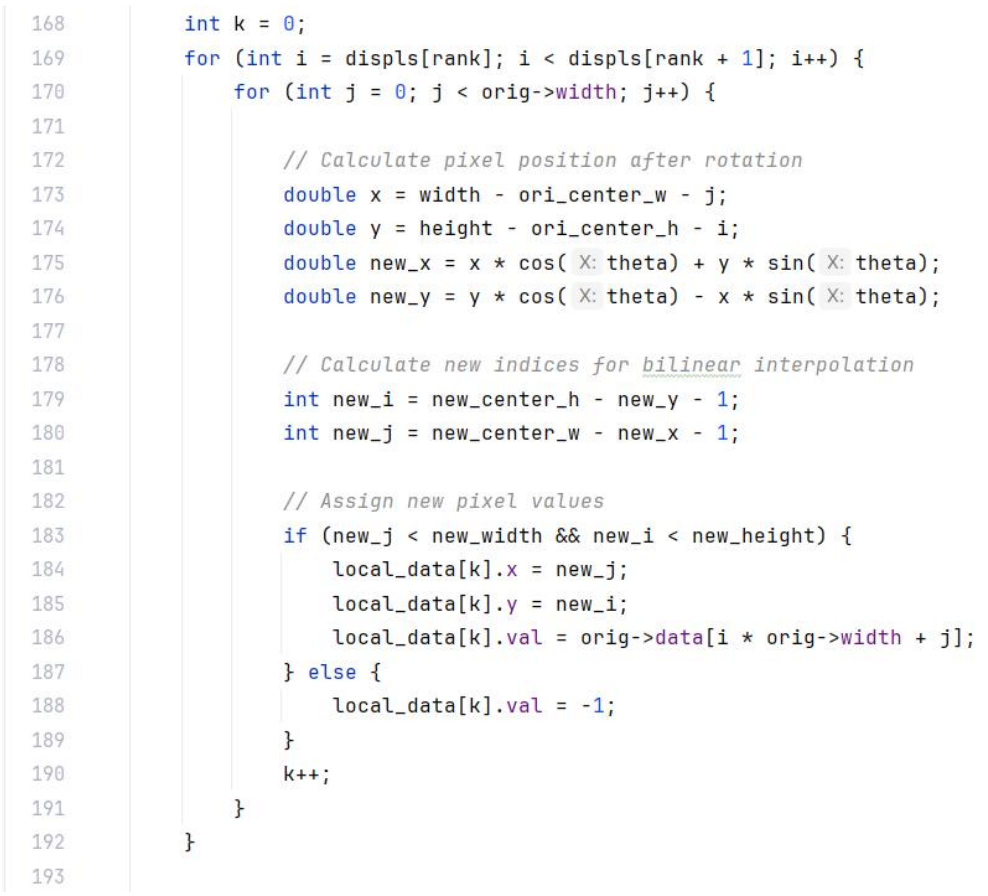
```c
// counts for Scatterv/Gatherv
int *get_counts(int *displs, int size) {
    // Allocate memory for the counts array
    int *counts = calloc( Count: size,  Size: sizeof(int));
    // Calculate the counts for each process
    for (int i = 0; i < size; i++) {
        counts[i] = displs[i + 1] - displs[i];
    }

    // Return the counts array
    return counts;
}

// struct for data
typedef struct {
    int x;
    int y;
    int val;
} pixel_t;

```



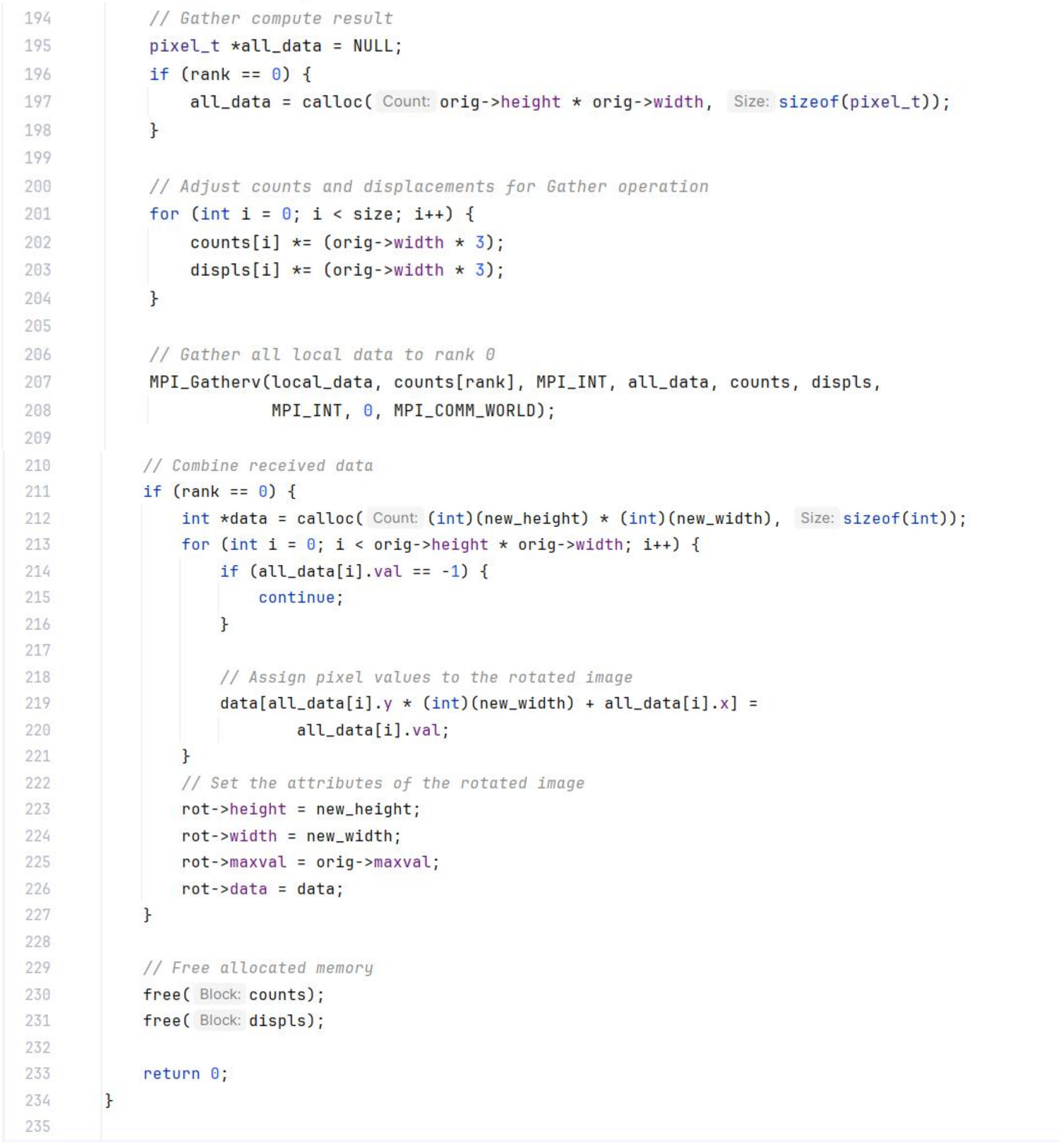
```c
// rotate
int rotate(image_t *orig, double angle, image_t *rot, int rank, int size) {

    // Broadcast image attributes
    int *param = (int *)(orig);
    MPI_Bcast(param, 3, MPI_INT, 0, MPI_COMM_WORLD);
    int *displs = get_displs( total: orig->height, size);
    int *counts = get_counts(displs, size);
    int local_count = counts[rank];

    // Broadcast image data
    if (rank != 0) {
        orig->data = calloc( Count: orig->height * orig->width,  Size: sizeof(int));
    }
    MPI_Bcast(orig->data, orig->height * orig->width, MPI_INT, 0,
              MPI_COMM_WORLD);

    pixel_t *local_data =
            (pixel_t *)calloc( Count: local_count * orig->width,  Size: sizeof(pixel_t));

```



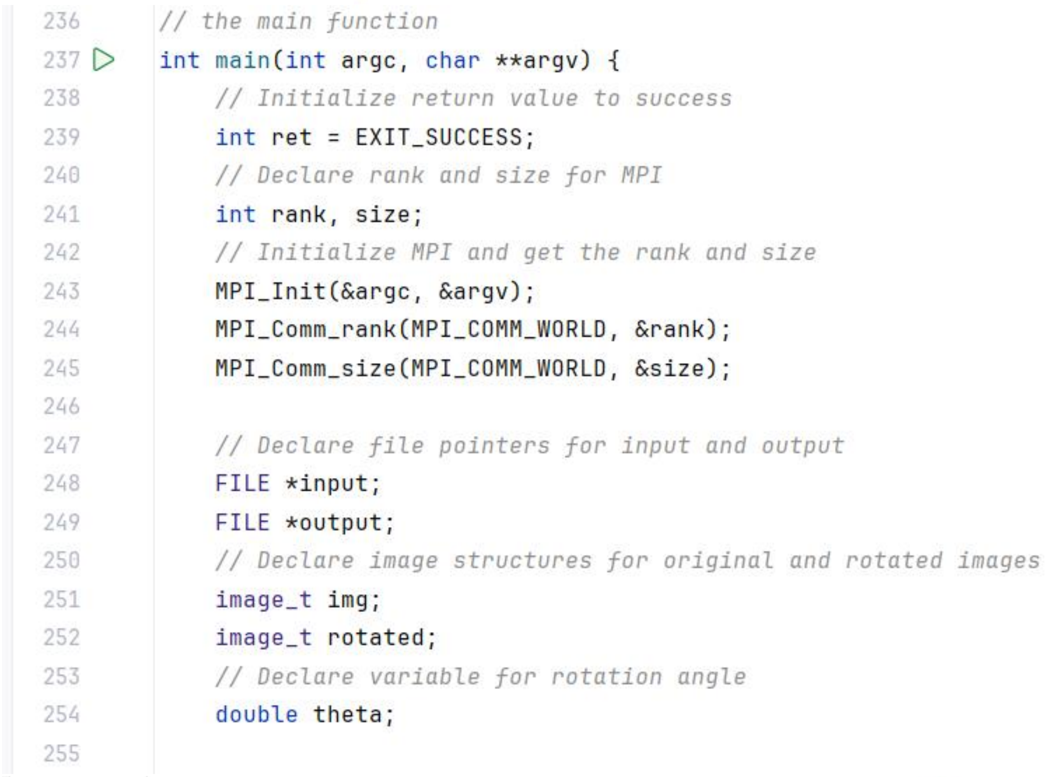
```
// Perform image rotation using bilinear interpolation
double height = orig->height;
double width = orig->width;
double theta = (angle / 180.0) * PI;
double new_height = fabs( x: height * cos( x: theta)) + fabs( x: width * sin( x: theta));
double new_width = fabs( x: width * cos( x: theta)) + fabs( x: height * sin( x: theta));
double ori_center_w = width / 2;
double ori_center_h = height / 2;
double new_center_w = new_width / 2;
double new_center_h = new_height / 2;
```



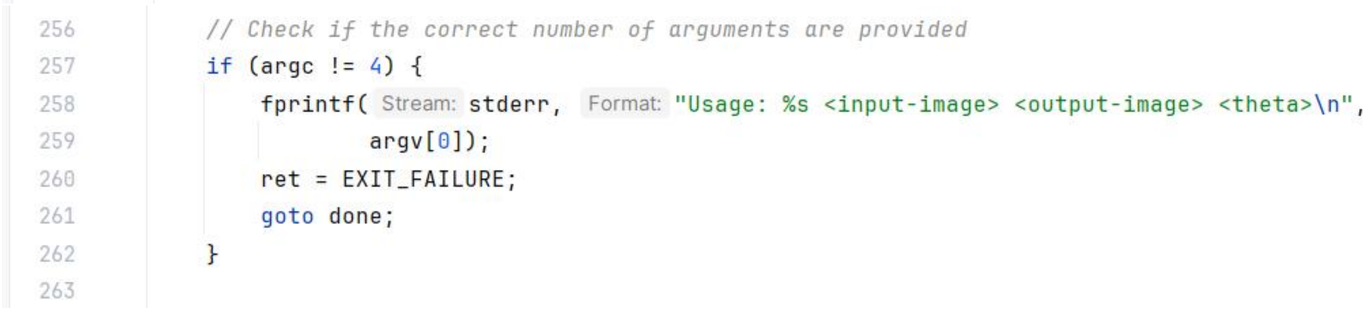
```
int k = 0;
for (int i = displs[rank]; i < displs[rank + 1]; i++) {
    for (int j = 0; j < orig->width; j++) {

        // Calculate pixel position after rotation
        double x = width - ori_center_w - j;
        double y = height - ori_center_h - i;
        double new_x = x * cos( x: theta) + y * sin( x: theta);
        double new_y = y * cos( x: theta) - x * sin( x: theta);

        // Calculate new indices for bilinear interpolation
        int new_i = new_center_h - new_y - 1;
        int new_j = new_center_w - new_x - 1;

        // Assign new pixel values
        if (new_j < new_width && new_i < new_height) {
            local_data[k].x = new_j;
            local_data[k].y = new_i;
            local_data[k].val = orig->data[i * orig->width + j];
        } else {
            local_data[k].val = -1;
        }
        k++;
    }
}

```



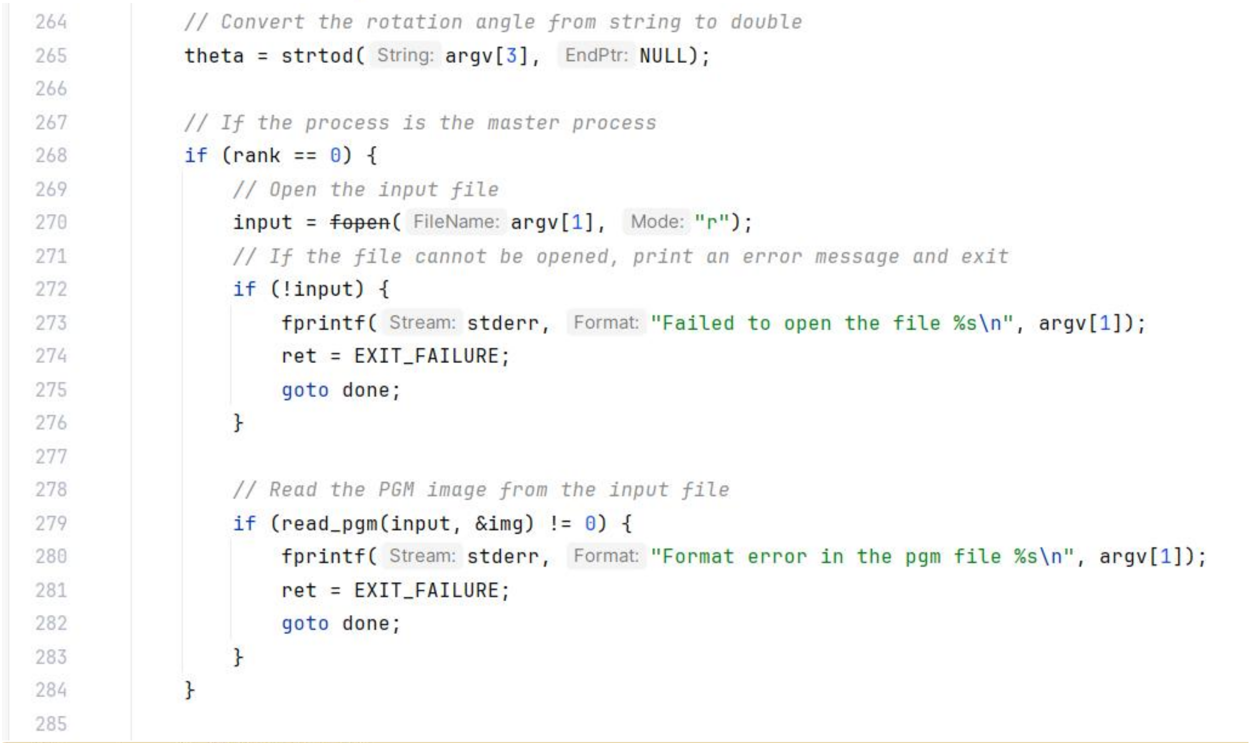
```
    // Gather compute result
    pixel_t *all_data = NULL;
    if (rank == 0) {
        all_data = calloc( Count: orig->height * orig->width,  Size: sizeof(pixel_t));
    }

    // Adjust counts and displacements for Gather operation
    for (int i = 0; i < size; i++) {
        counts[i] *= (orig->width * 3);
        displs[i] *= (orig->width * 3);
    }

    // Gather all local data to rank 0
    MPI_Gatherv(local_data, counts[rank], MPI_INT, all_data, counts, displs,
                MPI_INT, 0, MPI_COMM_WORLD);

    // Combine received data
    if (rank == 0) {
        int *data = calloc( Count: (int)(new_height) * (int)(new_width),  Size: sizeof(int));
        for (int i = 0; i < orig->height * orig->width; i++) {
            if (all_data[i].val == -1) {
                continue;
            }

            // Assign pixel values to the rotated image
            data[all_data[i].y * (int)(new_width) + all_data[i].x] =
                    all_data[i].val;
        }
        // Set the attributes of the rotated image
        rot->height = new_height;
        rot->width = new_width;
        rot->maxval = orig->maxval;
        rot->data = data;
    }

    // Free allocated memory
    free( Block: counts);
    free( Block: displs);

    return 0;
}
```



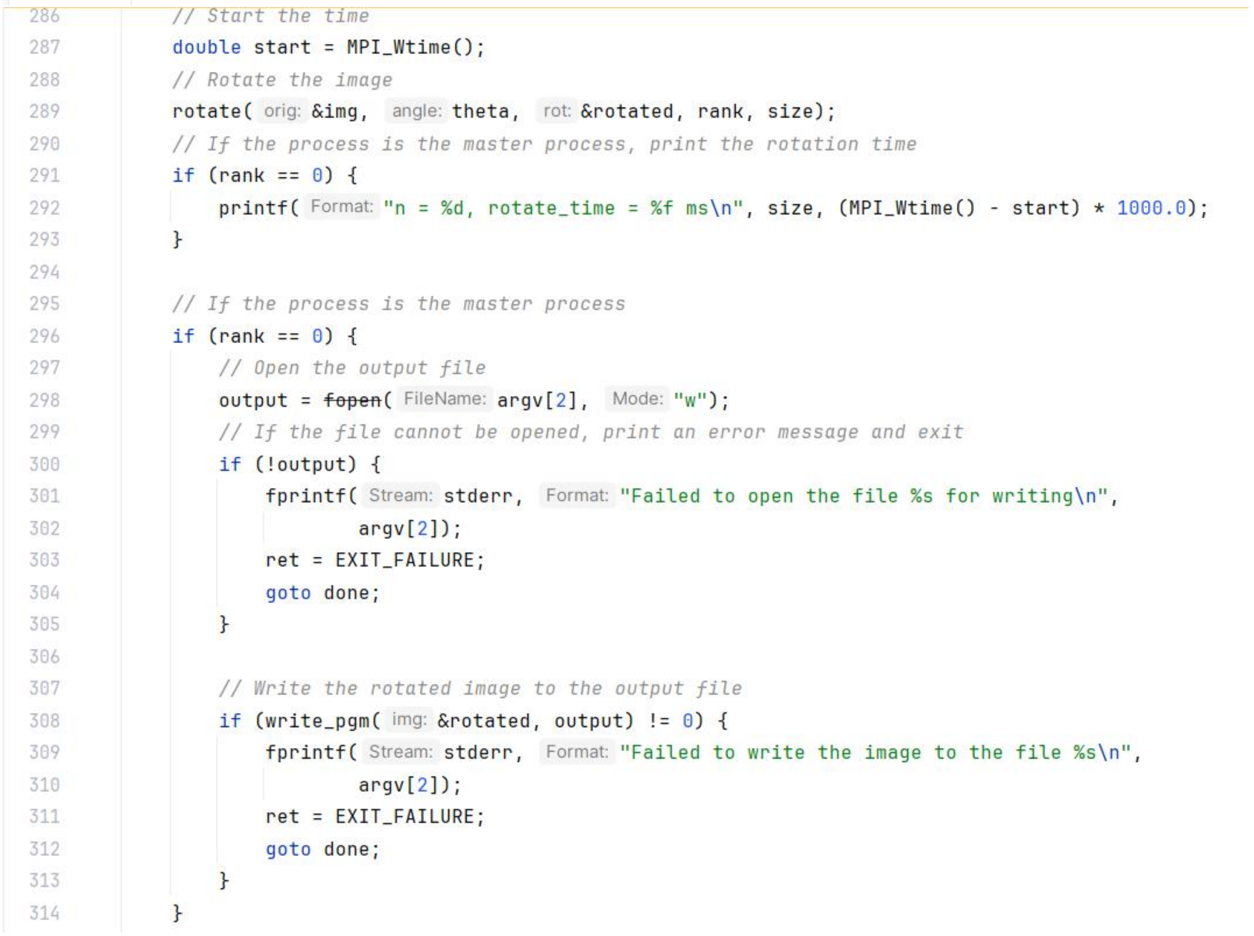
```
// the main function
int main(int argc, char **argv) {
    // Initialize return value to success
    int ret = EXIT_SUCCESS;
    // Declare rank and size for MPI
    int rank, size;
    // Initialize MPI and get the rank and size
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Declare file pointers for input and output
    FILE *input;
    FILE *output;
    // Declare image structures for original and rotated images
    image_t img;
    image_t rotated;
    // Declare variable for rotation angle
    double theta;

```



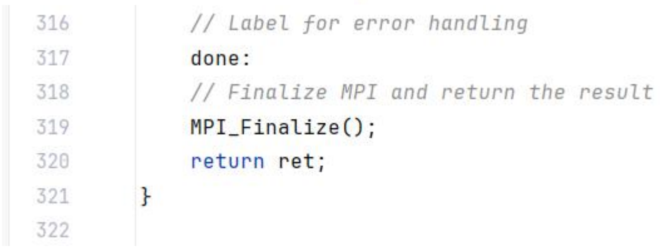
```
    // Check if the correct number of arguments are provided
    if (argc != 4) {
        fprintf( Stream: stderr,  Format: "Usage: %s <input-image> <output-image> <theta>\n",
                argv[0]);
        ret = EXIT_FAILURE;
        goto done;
    }

```



```
    // Convert the rotation angle from string to double
    theta = strtod( String: argv[3],  EndPtr: NULL);

    // If the process is the master process
    if (rank == 0) {
        // Open the input file
        input = ~~fopen~~( FileName: argv[1],  Mode: "r");
        // If the file cannot be opened, print an error message and exit
        if (!input) {
            fprintf( Stream: stderr,  Format: "Failed to open the file %s\n", argv[1]);
            ret = EXIT_FAILURE;
            goto done;
        }

        // Read the PGM image from the input file
        if (read_pgm(input, &img) != 0) {
            fprintf( Stream: stderr,  Format: "Format error in the pgm file %s\n", argv[1]);
            ret = EXIT_FAILURE;
            goto done;
        }
    }

```

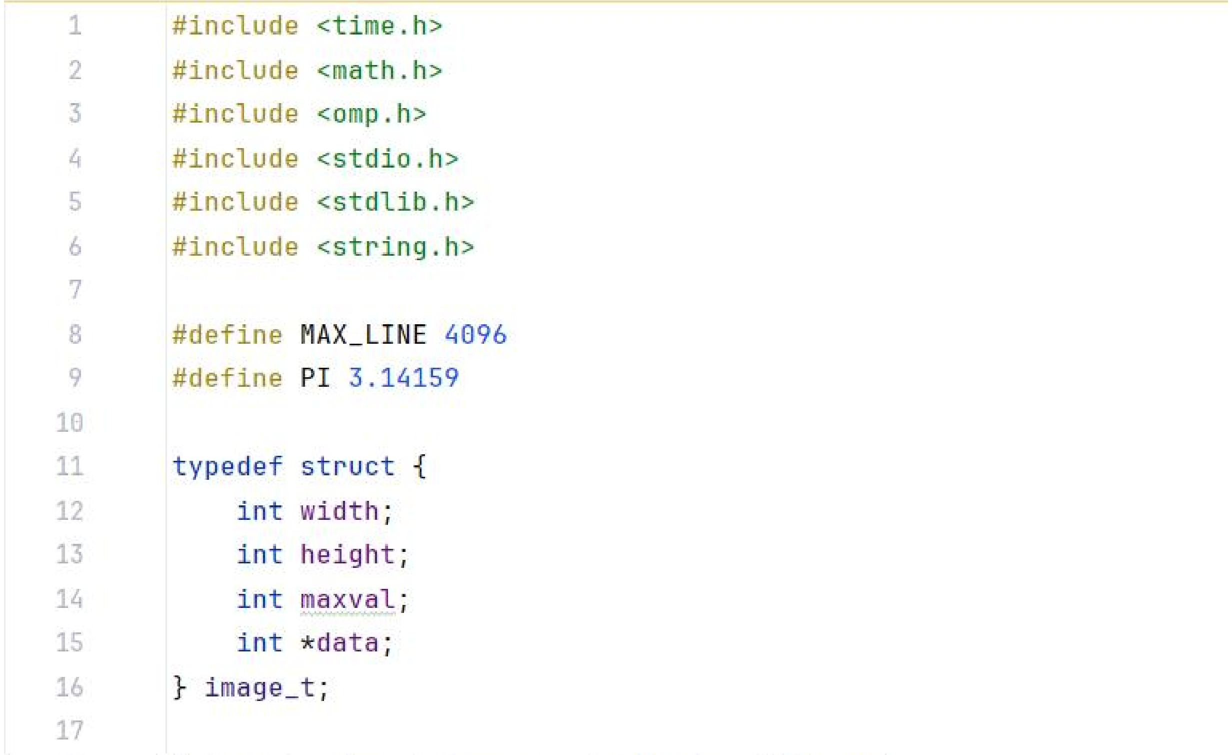


```
    // Start the time
    double start = MPI_Wtime();
    // Rotate the image
    rotate( orig: &img,  angle: theta,  rot: &rotated, rank, size);
    // If the process is the master process, print the rotation time
    if (rank == 0) {
        printf( Format: "n = %d, rotate_time = %f ms\n", size, (MPI_Wtime() - start) * 1000.0);
    }

    // If the process is the master process
    if (rank == 0) {
        // Open the output file
        output = ~~fopen~~( FileName: argv[2],  Mode: "w");
        // If the file cannot be opened, print an error message and exit
        if (!output) {
            fprintf( Stream: stderr,  Format: "Failed to open the file %s for writing\n",
                    argv[2]);
            ret = EXIT_FAILURE;
            goto done;
        }

        // Write the rotated image to the output file
        if (write_pgm( img: &rotated, output) != 0) {
            fprintf( Stream: stderr,  Format: "Failed to write the image to the file %s\n",
                    argv[2]);
            ret = EXIT_FAILURE;
            goto done;
        }
    }
```

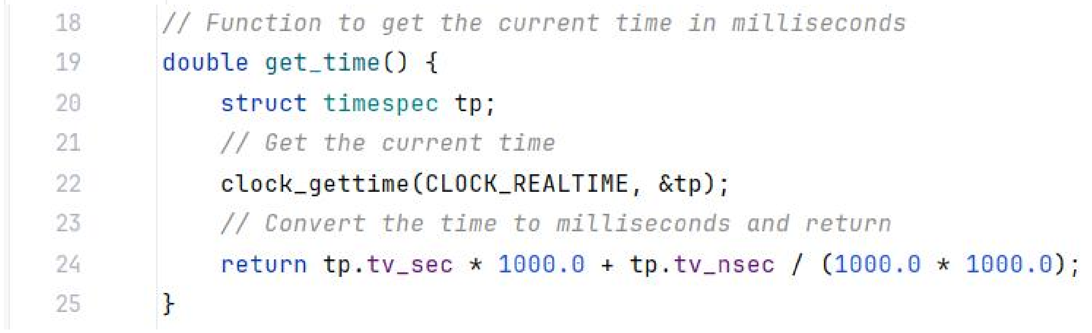


```
    // Label for error handling
    done:
    // Finalize MPI and return the result
    MPI_Finalize();
    return ret;
}

```

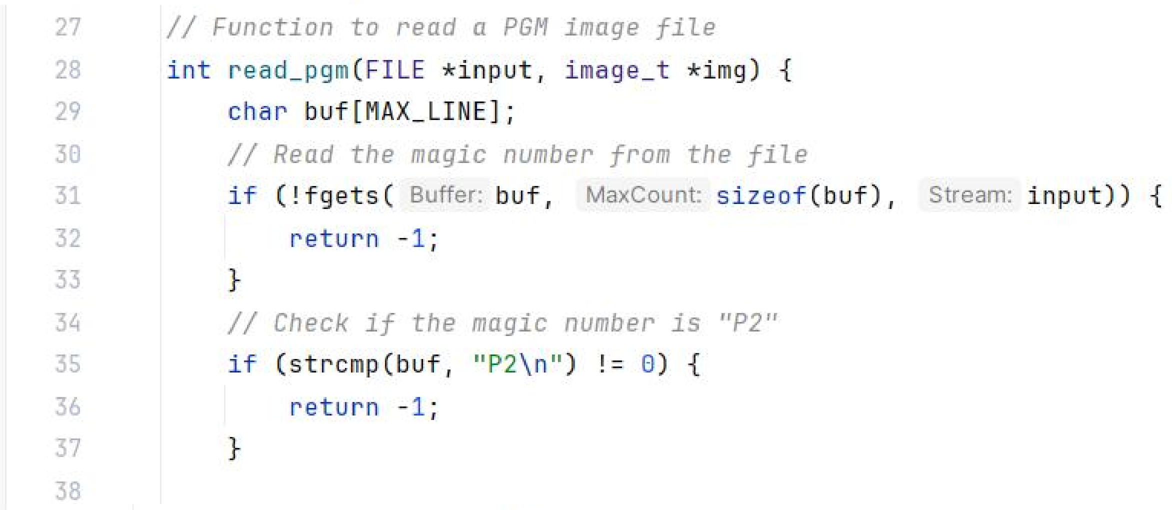
OpenMP Code



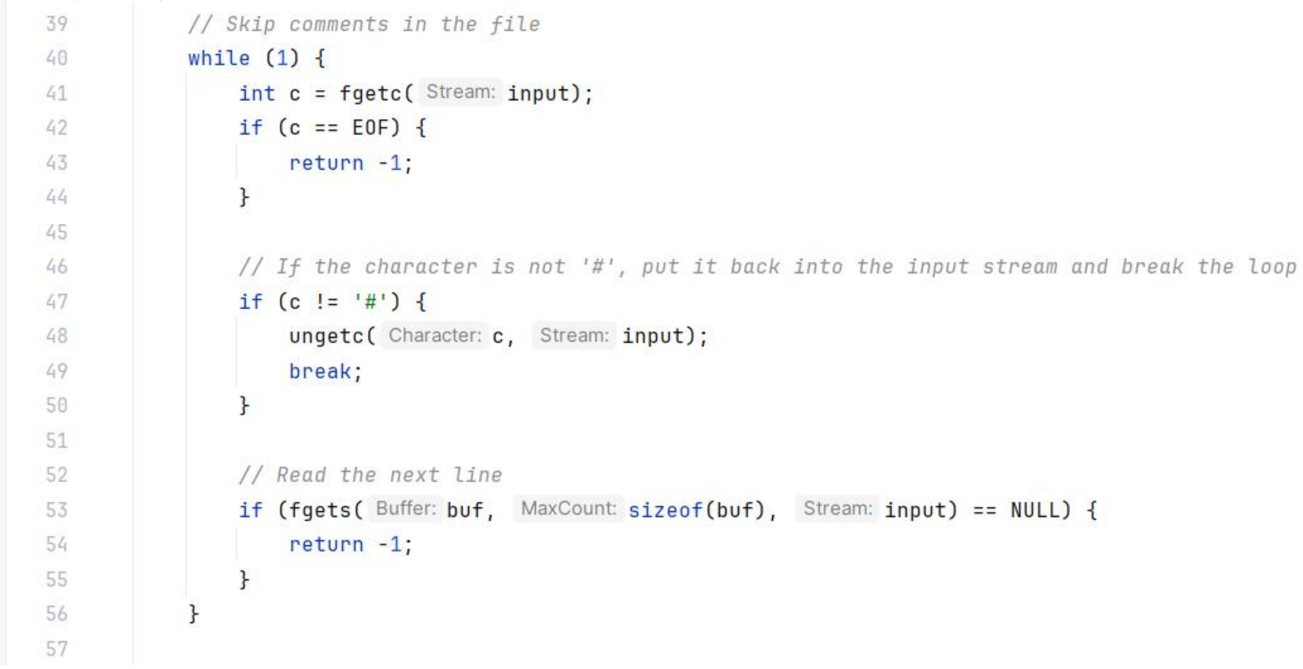
```
#include <time.h>
#include <math.h>
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_LINE 4096
#define PI 3.14159

typedef struct {
    int width;
    int height;
    int maxval;
    int *data;
} image_t;

```



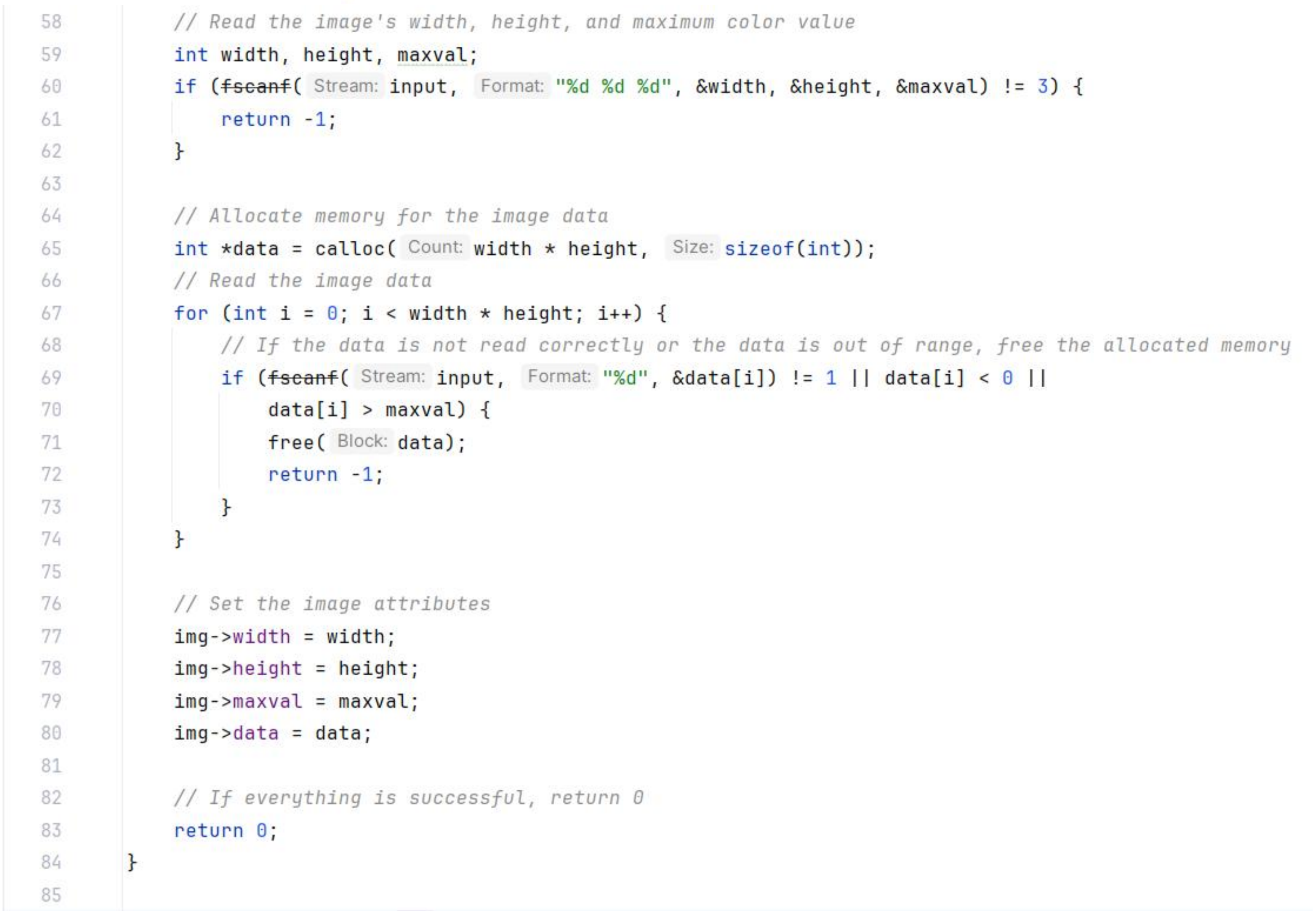
```
// Function to get the current time in milliseconds
double get_time() {
    struct timespec tp;
    // Get the current time
    clock_gettime(CLOCK_REALTIME, &tp);
    // Convert the time to milliseconds and return
    return tp.tv_sec * 1000.0 + tp.tv_nsec / (1000.0 * 1000.0);
}
```



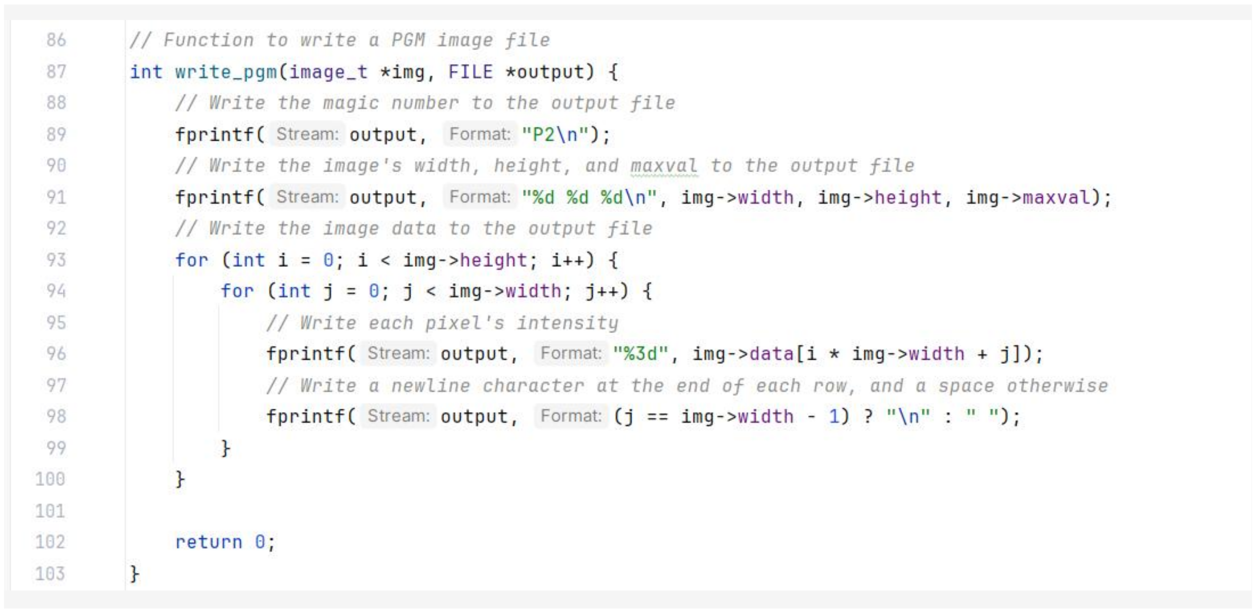
```c
// Function to read a PGM image file
int read_pgm(FILE *input, image_t *img) {
    char buf[MAX_LINE];
    // Read the magic number from the file
    if (!fgets( Buffer: buf,  MaxCount: sizeof(buf),  Stream: input)) {
        return -1;
    }
    // Check if the magic number is "P2"
    if (strcmp(buf, "P2\n") != 0) {
        return -1;
    }

```



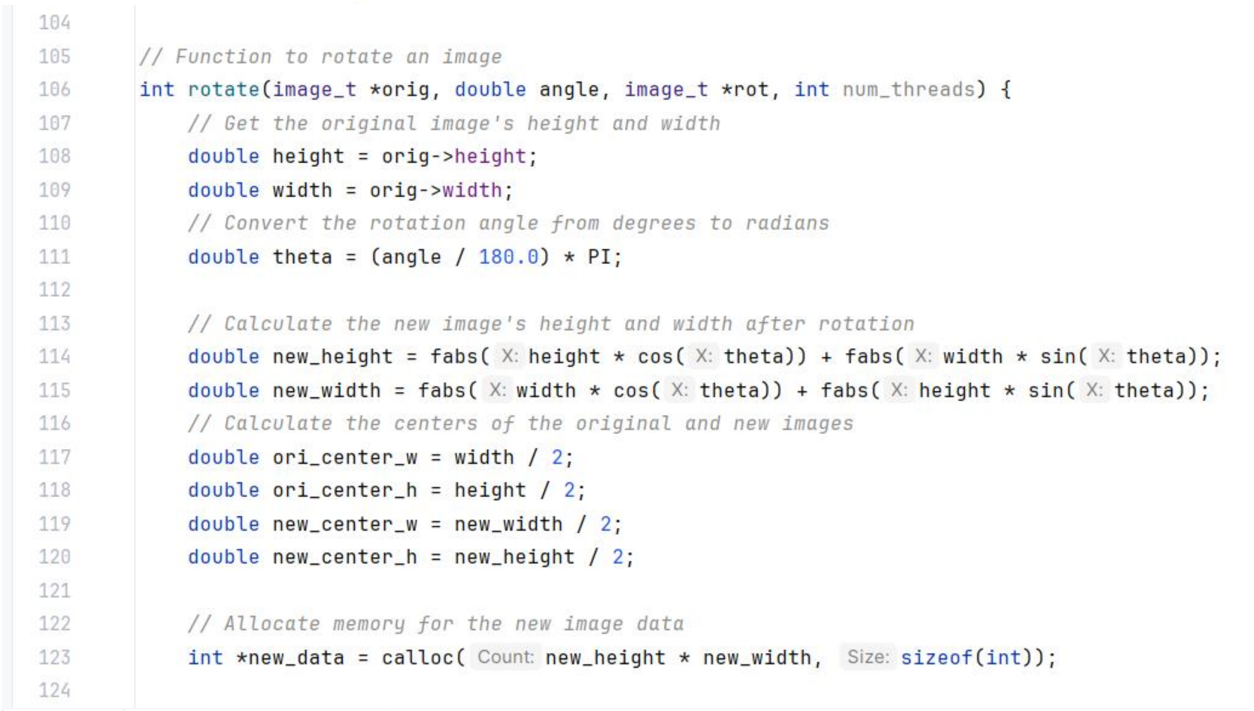
```c
    // Skip comments in the file
    while (1) {
        int c = fgetc( Stream: input);
        if (c == EOF) {
            return -1;
        }

        // If the character is not '#', put it back into the input stream and break the loop
        if (c != '#') {
            ungetc( Character: c,  Stream: input);
            break;
        }

        // Read the next line
        if (fgets( Buffer: buf,  MaxCount: sizeof(buf),  Stream: input) == NULL) {
            return -1;
        }
    }

```



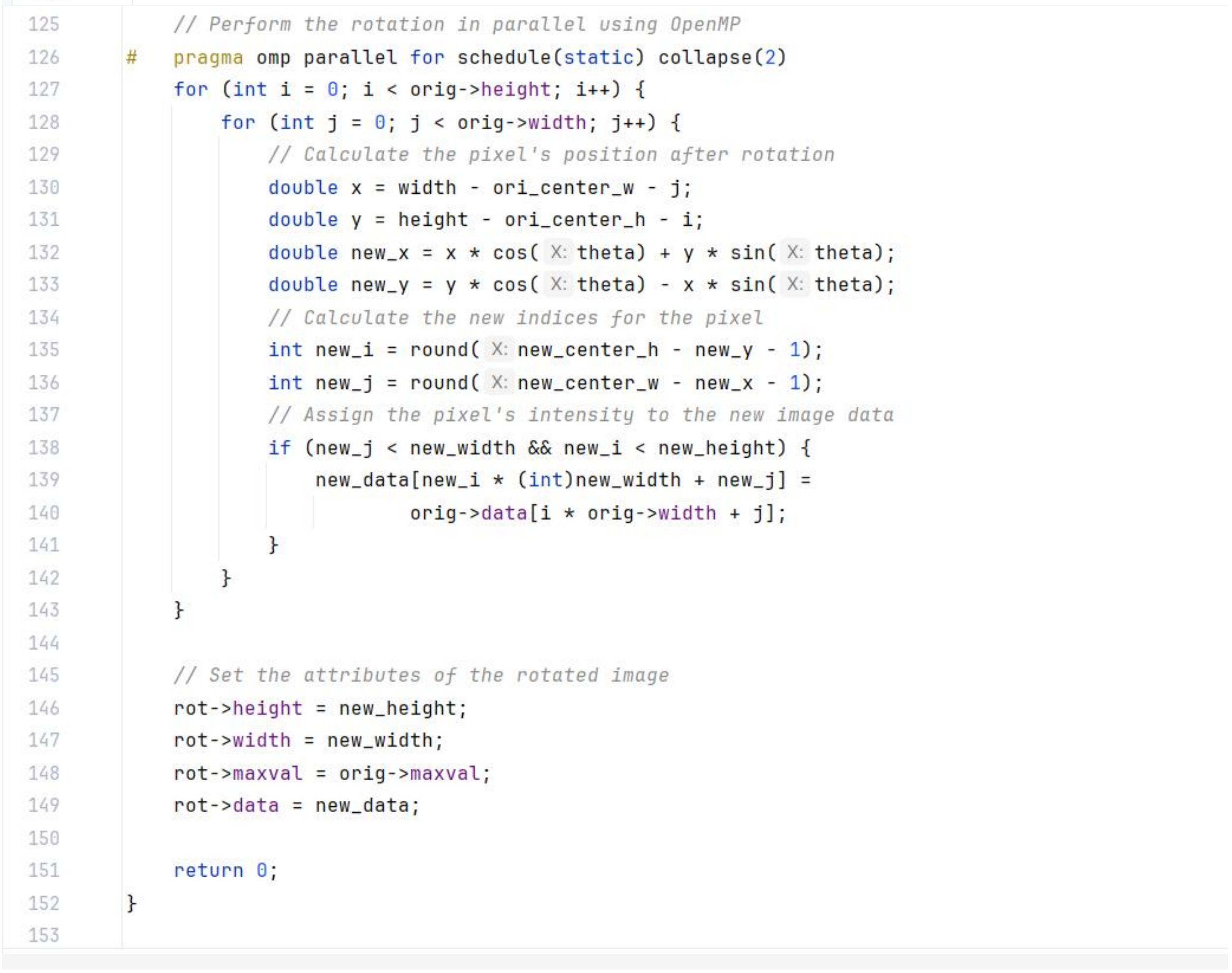
```
    // Read the image's width, height, and maximum color value
    int width, height, maxval;
    if (~~fscanf~~( Stream: input,  Format: "%d %d %d", &width, &height, &maxval) != 3) {
        return -1;
    }

    // Allocate memory for the image data
    int *data = calloc( Count: width * height,  Size: sizeof(int));
    // Read the image data
    for (int i = 0; i < width * height; i++) {
        // If the data is not read correctly or the data is out of range, free the allocated memory
        if (~~fscanf~~( Stream: input,  Format: "%d", &data[i]) != 1 || data[i] < 0 ||
            data[i] > maxval) {
            free( Block: data);
            return -1;
        }
    }

    // Set the image attributes
    img->width = width;
    img->height = height;
    img->maxval = maxval;
    img->data = data;

    // If everything is successful, return 0
    return 0;
}

```



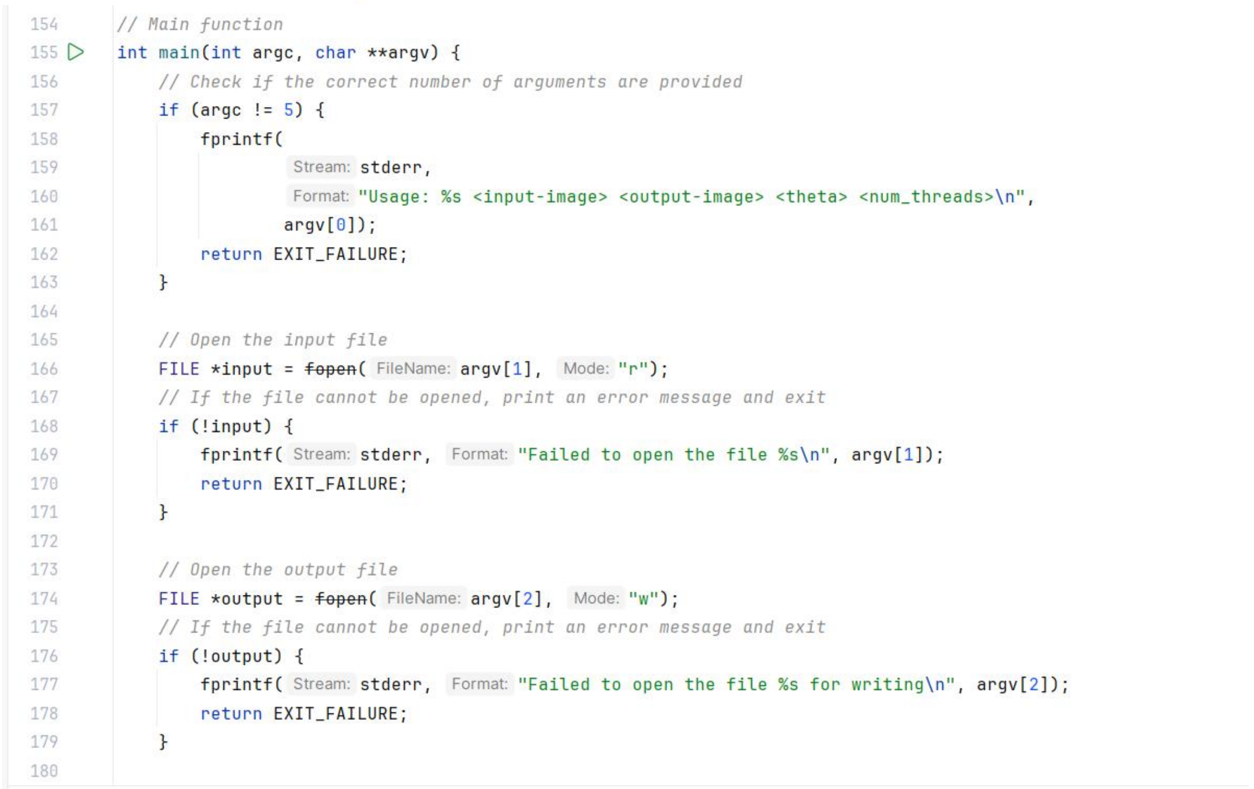
```
// Function to write a PGM image file
int write_pgm(image_t *img, FILE *output) {
    // Write the magic number to the output file
    fprintf( Stream: output,  Format: "P2\n");
    // Write the image's width, height, and maxval to the output file
    fprintf( Stream: output,  Format: "%d %d %d\n", img->width, img->height, img->maxval);
    // Write the image data to the output file
    for (int i = 0; i < img->height; i++) {
        for (int j = 0; j < img->width; j++) {
            // Write each pixel's intensity
            fprintf( Stream: output,  Format: "%3d", img->data[i * img->width + j]);
            // Write a newline character at the end of each row, and a space otherwise
            fprintf( Stream: output,  Format: (j == img->width - 1) ? "\n" : " ");
        }
    }

    return 0;
}
```



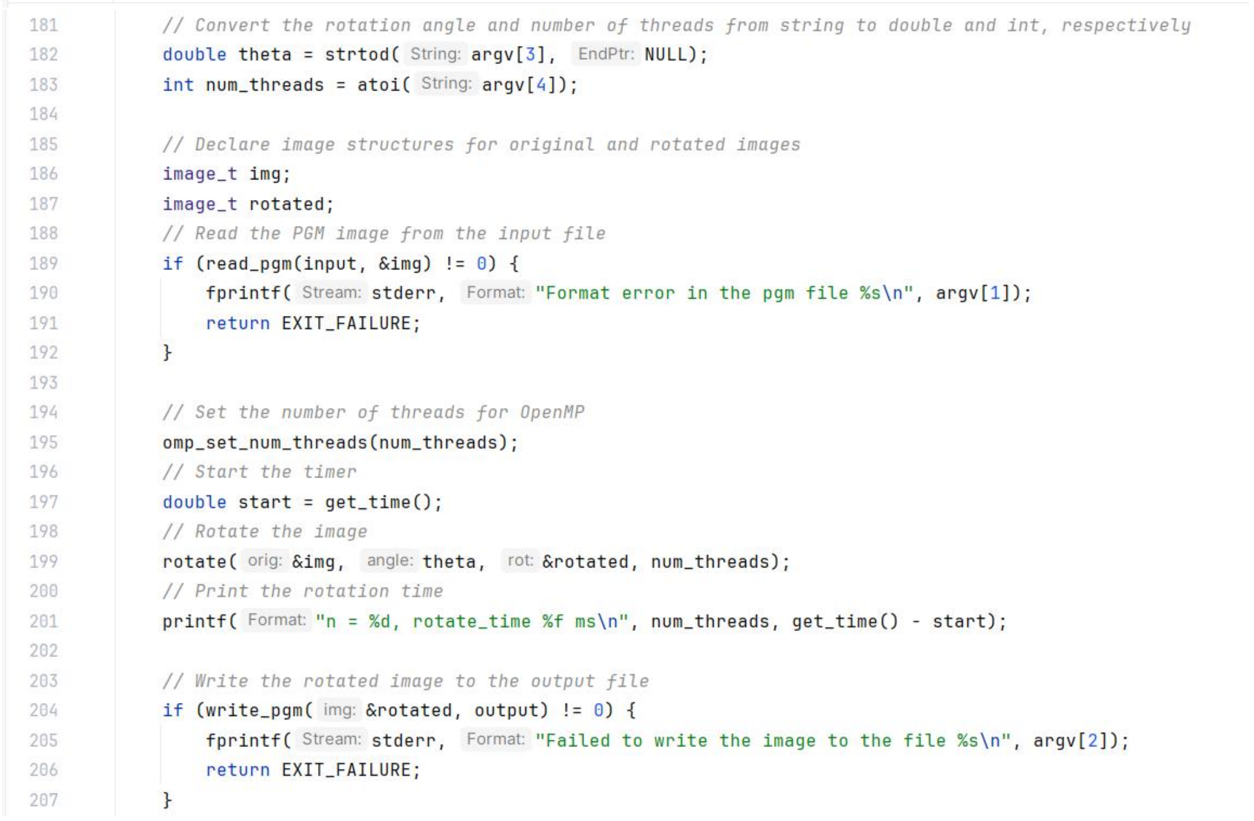
```c

// Function to rotate an image
int rotate(image_t *orig, double angle, image_t *rot, int num_threads) {
    // Get the original image's height and width
    double height = orig->height;
    double width = orig->width;
    // Convert the rotation angle from degrees to radians
    double theta = (angle / 180.0) * PI;

    // Calculate the new image's height and width after rotation
    double new_height = fabs( X: height * cos( X: theta)) + fabs( X: width * sin( X: theta));
    double new_width = fabs( X: width * cos( X: theta)) + fabs( X: height * sin( X: theta));
    // Calculate the centers of the original and new images
    double ori_center_w = width / 2;
    double ori_center_h = height / 2;
    double new_center_w = new_width / 2;
    double new_center_h = new_height / 2;

    // Allocate memory for the new image data
    int *new_data = calloc( Count: new_height * new_width,  Size: sizeof(int));

```



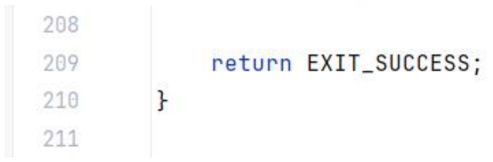
```c
    // Perform the rotation in parallel using OpenMP
#   pragma omp parallel for schedule(static) collapse(2)
    for (int i = 0; i < orig->height; i++) {
        for (int j = 0; j < orig->width; j++) {
            // Calculate the pixel's position after rotation
            double x = width - ori_center_w - j;
            double y = height - ori_center_h - i;
            double new_x = x * cos( X: theta) + y * sin( X: theta);
            double new_y = y * cos( X: theta) - x * sin( X: theta);
            // Calculate the new indices for the pixel
            int new_i = round( X: new_center_h - new_y - 1);
            int new_j = round( X: new_center_w - new_x - 1);
            // Assign the pixel's intensity to the new image data
            if (new_j < new_width && new_i < new_height) {
                new_data[new_i * (int)new_width + new_j] =
                        orig->data[i * orig->width + j];
            }
        }
    }

    // Set the attributes of the rotated image
    rot->height = new_height;
    rot->width = new_width;
    rot->maxval = orig->maxval;
    rot->data = new_data;

    return 0;
}

```



```c
// Main function
int main(int argc, char **argv) {
    // Check if the correct number of arguments are provided
    if (argc != 5) {
        fprintf(
                Stream: stderr,
                Format: "Usage: %s <input-image> <output-image> <theta> <num_threads>\n",
                argv[0]);
        return EXIT_FAILURE;
    }

    // Open the input file
    FILE *input = ~~fopen~~( FileName: argv[1],  Mode: "r");
    // If the file cannot be opened, print an error message and exit
    if (!input) {
        fprintf( Stream: stderr,  Format: "Failed to open the file %s\n", argv[1]);
        return EXIT_FAILURE;
    }

    // Open the output file
    FILE *output = ~~fopen~~( FileName: argv[2],  Mode: "w");
    // If the file cannot be opened, print an error message and exit
    if (!output) {
        fprintf( Stream: stderr,  Format: "Failed to open the file %s for writing\n", argv[2]);
        return EXIT_FAILURE;
    }

```



```c
    // Convert the rotation angle and number of threads from string to double and int, respectively
    double theta = strtod( String: argv[3],  EndPtr: NULL);
    int num_threads = atoi( String: argv[4]);

    // Declare image structures for original and rotated images
    image_t img;
    image_t rotated;
    // Read the PGM image from the input file
    if (read_pgm(input, &img) != 0) {
        fprintf( Stream: stderr,  Format: "Format error in the pgm file %s\n", argv[1]);
        return EXIT_FAILURE;
    }

    // Set the number of threads for OpenMP
    omp_set_num_threads(num_threads);
    // Start the timer
    double start = get_time();
    // Rotate the image
    rotate( orig: &img,  angle: theta,  rot: &rotated, num_threads);
    // Print the rotation time
    printf( Format: "n = %d, rotate_time %f ms\n", num_threads, get_time() - start);

    // Write the rotated image to the output file
    if (write_pgm( img: &rotated, output) != 0) {
        fprintf( Stream: stderr,  Format: "Failed to write the image to the file %s\n", argv[2]);
        return EXIT_FAILURE;
    }
```



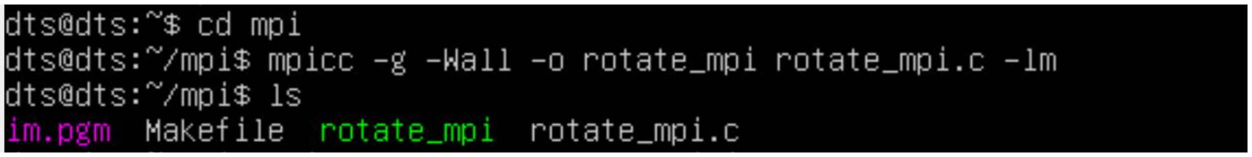
```

    return EXIT_SUCCESS;
}

```

2. Performance Analysis

I implement two difference parallel programs, using MPI and OpenMP.

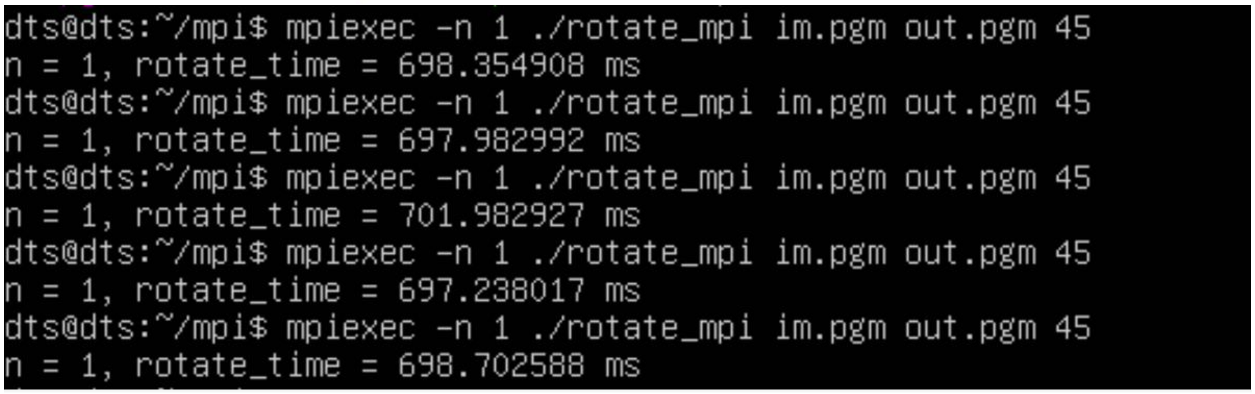
In order to analyze the speedup and efficiencies of the two parallel implementations, I first need to obtain the time of the core algorithm of the two algorithms under different numbers of processes.

The time of the core algorithm of MPI



```
dts@dts:~$ cd mpi
dts@dts:~/mpi$ mpicc -g -Wall -o rotate_mpi rotate_mpi.c -lm
dts@dts:~/mpi$ ls
im.pgm  Makefile  rotate_mpi  rotate_mpi.c
```

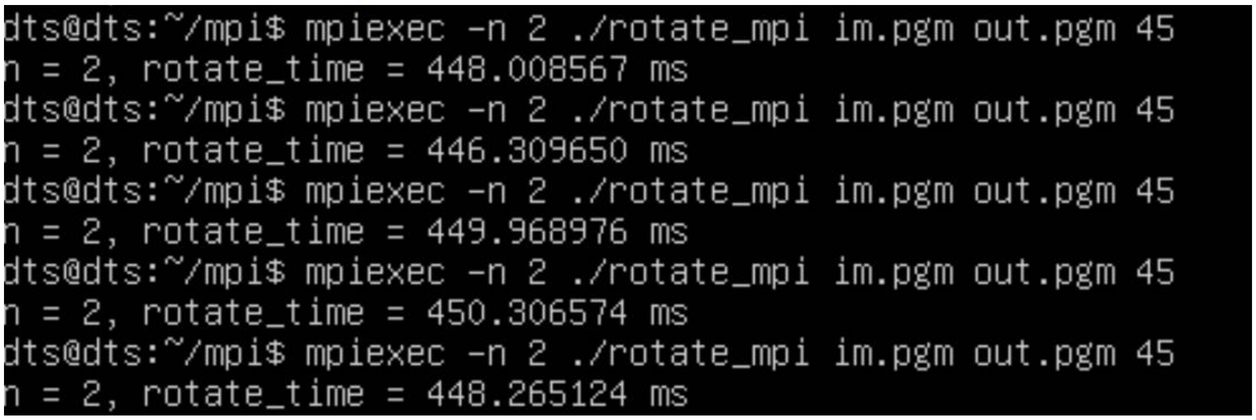
I will perform the following series of commands in the Ubuntu system to compile the MPI program. First, use the cd mpi command to change to the directory named mpi, which may contain the source code files for the MPI program. Next, use the mpicc command to compile the C language source code file named rotate_mpi.c. The option -g means to generate debuggable code, -Wall means to display all warning information, and -o rotate_mpi specifies the name of the generated executable file as rotate_mpi. The final -lm option means linking the math library, which is typically used for math functions included in the code. Finally, use the ls command to list the files and subdirectories in the current working directory to confirm whether the executable file named rotate_mpi was successfully generated. This series of steps is used to prepare the compilation and execution of MPI programs.



```
dts@dts:~/mpi$ mpiexec -n 1 ./rotate_mpi im.pgm out.pgm 45
n = 1, rotate_time = 698.354908 ms
dts@dts:~/mpi$ mpiexec -n 1 ./rotate_mpi im.pgm out.pgm 45
n = 1, rotate_time = 697.982992 ms
dts@dts:~/mpi$ mpiexec -n 1 ./rotate_mpi im.pgm out.pgm 45
n = 1, rotate_time = 701.982927 ms
dts@dts:~/mpi$ mpiexec -n 1 ./rotate_mpi im.pgm out.pgm 45
n = 1, rotate_time = 697.238017 ms
dts@dts:~/mpi$ mpiexec -n 1 ./rotate_mpi im.pgm out.pgm 45
n = 1, rotate_time = 698.702588 ms
```

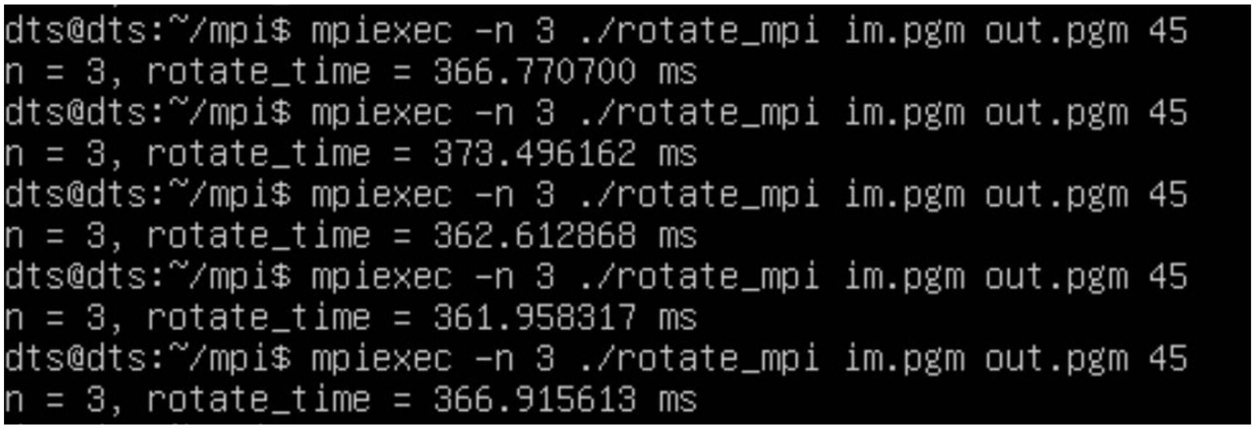
Then I will perform the image rotation operation by executing the MPI program named rotate_mpi. Specifically, the program reads an input image file named im.pgm, rotates it 45 degrees, and writes the result to an output image file named out.pgm. By specifying the number after -n, you can set the corresponding number of processes to run in mode. Since I have added "MPI_Wtime" in the code, an MPI time function that gets the time. Therefore, after executing the program, the corresponding core algorithm time will be printed.

The above picture shows the use of a single MPI process to perform image rotation, that is, executed in a serial manner. In order to make the time obtained relatively accurate, I ran it five times and found the average of the last five results to be **698.86** ms.



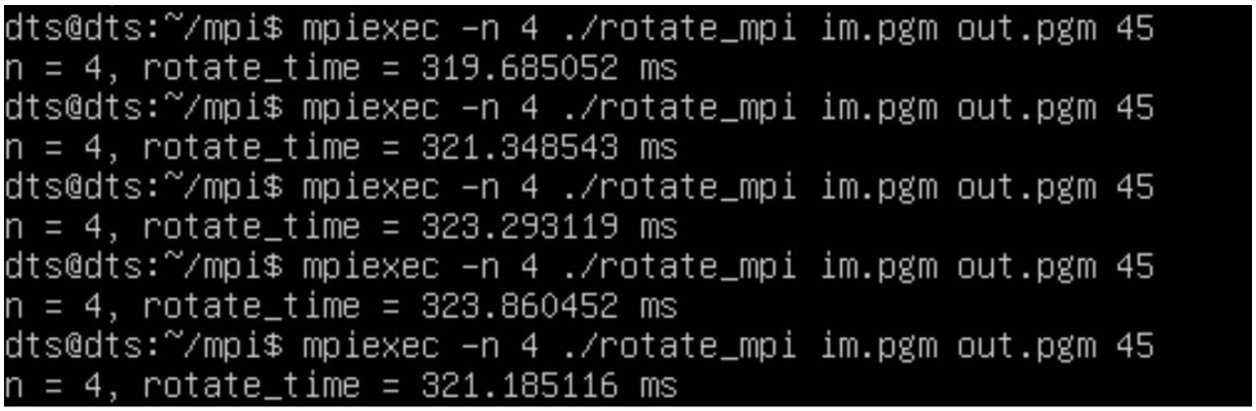
```
dts@dts:~/mpi$ mpiexec -n 2 ./rotate_mpi im.pgm out.pgm 45
n = 2, rotate_time = 448.008567 ms
dts@dts:~/mpi$ mpiexec -n 2 ./rotate_mpi im.pgm out.pgm 45
n = 2, rotate_time = 446.309650 ms
dts@dts:~/mpi$ mpiexec -n 2 ./rotate_mpi im.pgm out.pgm 45
n = 2, rotate_time = 449.968976 ms
dts@dts:~/mpi$ mpiexec -n 2 ./rotate_mpi im.pgm out.pgm 45
n = 2, rotate_time = 450.306574 ms
dts@dts:~/mpi$ mpiexec -n 2 ./rotate_mpi im.pgm out.pgm 45
n = 2, rotate_time = 448.265124 ms
```

The picture above shows enabling two MPI processes to perform image rotation. Similarly, in order to make the time obtained relatively accurate, I ran it five times and found the average of the last five results to be **448.58** ms.



```
dts@dts:~/mpi$ mpiexec -n 3 ./rotate_mpi im.pgm out.pgm 45
n = 3, rotate_time = 366.770700 ms
dts@dts:~/mpi$ mpiexec -n 3 ./rotate_mpi im.pgm out.pgm 45
n = 3, rotate_time = 373.496162 ms
dts@dts:~/mpi$ mpiexec -n 3 ./rotate_mpi im.pgm out.pgm 45
n = 3, rotate_time = 362.612868 ms
dts@dts:~/mpi$ mpiexec -n 3 ./rotate_mpi im.pgm out.pgm 45
n = 3, rotate_time = 361.958317 ms
dts@dts:~/mpi$ mpiexec -n 3 ./rotate_mpi im.pgm out.pgm 45
n = 3, rotate_time = 366.915613 ms
```

The picture above shows enabling three MPI processes to perform image rotation. Similarly, in order to make the time obtained relatively accurate, I ran it five times and found the average of the last five results to be **366.36** ms.


```
dts@dts:~/mpi$ mpiexec -n 4 ./rotate_mpi im.pgm out.pgm 45
n = 4, rotate_time = 319.685052 ms
dts@dts:~/mpi$ mpiexec -n 4 ./rotate_mpi im.pgm out.pgm 45
n = 4, rotate_time = 321.348543 ms
dts@dts:~/mpi$ mpiexec -n 4 ./rotate_mpi im.pgm out.pgm 45
n = 4, rotate_time = 323.293119 ms
dts@dts:~/mpi$ mpiexec -n 4 ./rotate_mpi im.pgm out.pgm 45
n = 4, rotate_time = 323.860452 ms
dts@dts:~/mpi$ mpiexec -n 4 ./rotate_mpi im.pgm out.pgm 45
n = 4, rotate_time = 321.185116 ms
```

The picture above shows enabling four MPI processes to perform image rotation. Similarly, in order to make the time obtained relatively accurate, I ran it five times and found the average of the last five results to be **321.88** ms.

The speedup of a parallel program is defined as the ratio of the execution time in a serial run to the execution time in a parallel run. The running time of a single process on a parallel system is considered as the running time of a serial program.Therefore, I can use the data obtained above to calculate the speedup for process counts of 2, 3, and 4, respectively.

When the number of processes is 2, **S = 698.96 / 448.58 = 1.56**

When the number of processes is 3, **S = 698.96 / 366.36 = 1.91**

When the number of processes is 4, **S = 698.96 / 321.88 = 2.17**

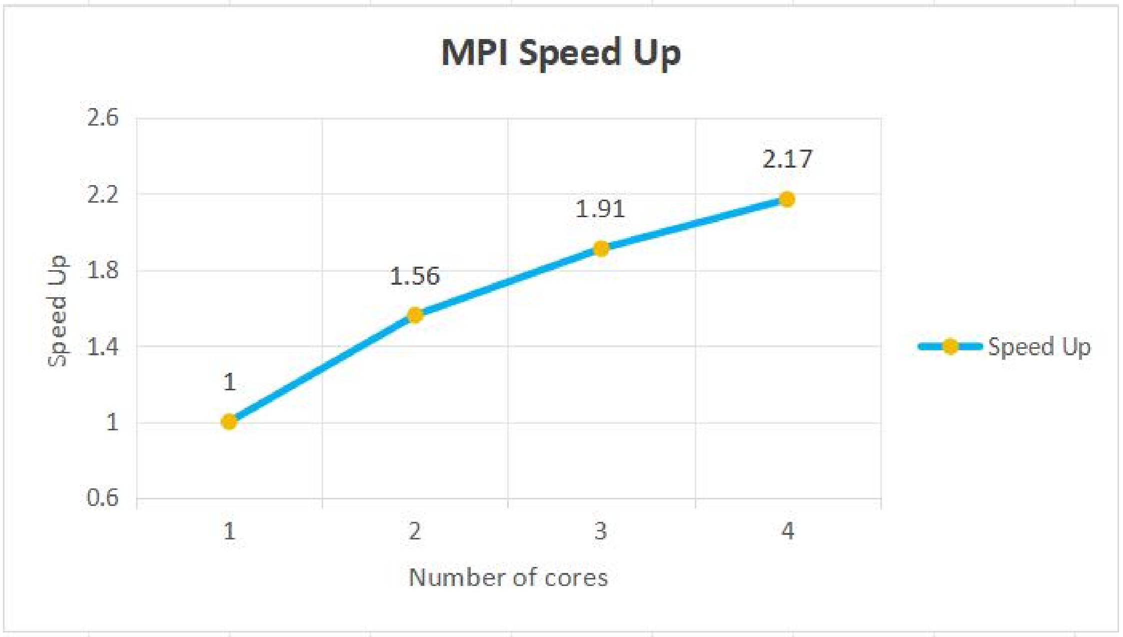
The efficiency of a parallel program is defined as the speedup divided by the corresponding number of processors or cores.

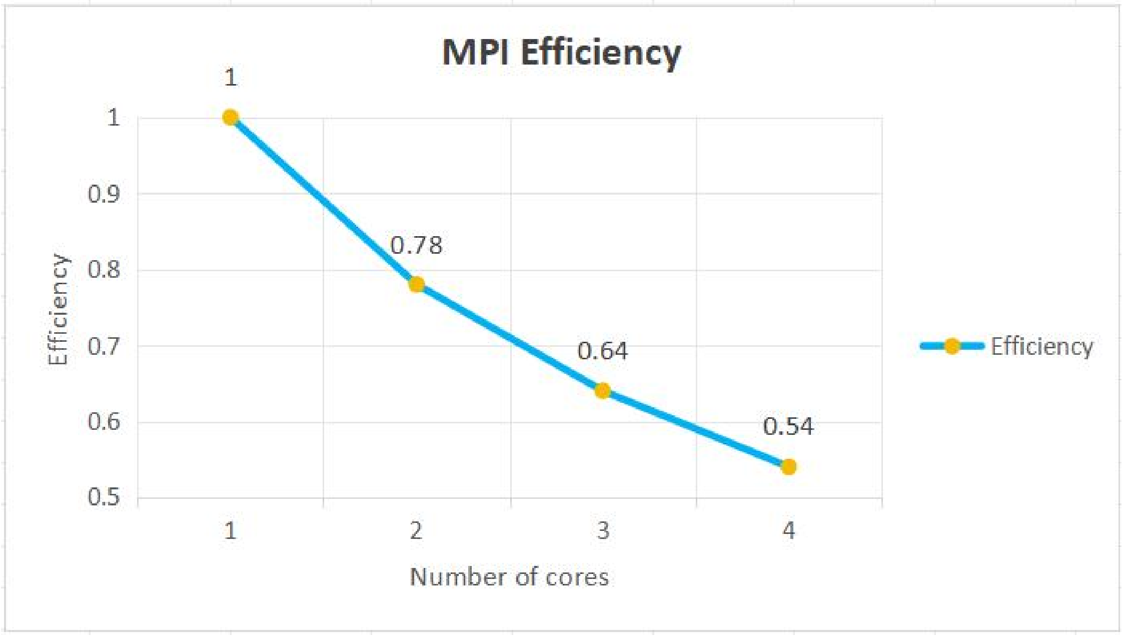
When the number of processes is 2, **E = 1.56 / 2 = 0.78**

When the number of processes is 3, **E = 1.91 / 3 = 0.64**

When the number of processes is 4, **E = 2.17 / 4 = 0.54**

From this, I can draw MPI's Speedup and Efficiency line plots, as shown below





Analyse the performance of MPI

Firstly, let's examine the Speedup. Speedup is a crucial metric for parallel computing performance, representing the improvement in speed of parallel computing compared to serial computing. Ideally, if n processes are used, one would expect the program's runtime to increase n-fold, achieving linear acceleration. However, in practical scenarios, achieving this ideal state is often challenging due to various influencing factors. From the data provided in the chart, when the number of processes increases from 1 to 4, the Speedup rises from 1 to 2.17. This indicates an improvement in parallel computing speed, although it falls short of achieving ideal linear acceleration.

Next, let's consider Efficiency. Efficiency is another significant performance metric, representing the contribution of each process to the overall computational speed. Ideally, if n processes are used, the efficiency of each process should be close to 1/n. However, based on the data from the chart, as the number of processes increases from 1 to 4, the efficiency decreases from 1 to 0.54. This suggests a reduction in the contribution of each process with the increasing number of processes.

This could be attributed to the following reasons:

Communication Overhead: In parallel computing, processes need to communicate through message passing. This communication incurs some time overhead, affecting the overall computational speed. Especially as the number of processes increases, communication overhead also tends to rise. In my code, MPI_Bcast function is used to broadcast image properties and data, and MPI_Gatherv function is used to collect the computation results of each process. These communication operations introduce some overhead, particularly when the number of processes increases, leading to potentially significant communication overhead and influencing parallel acceleration.

Load Imbalance: If the computational load among processes is uneven, some processes may become idle while waiting for others to complete their tasks, resulting in wasted computational resources. In my code, each process is assigned to process a row of the image. If the number of image rows is not divisible by the number of processes, some processes may receive more rows, leading to load imbalance. This can result in some processes being idle while waiting for others to complete their tasks, affecting parallel acceleration.

Optimization suggestions:

Improve load balancing:

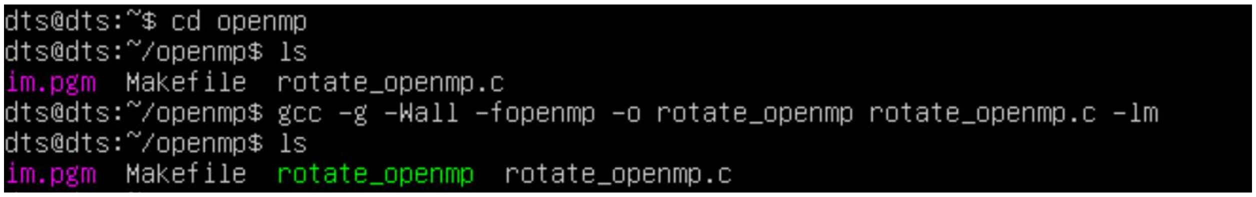
Try using a more dynamic load balancing strategy, dynamically allocating tasks based on each process's computational capabilities and communication latency. Reduce communication overhead:

Optimize communication patterns, such as reducing the use of broadcast and collective operations, or reducing communication costs by reducing the amount of data. Algorithm adjustment:

Optimize the rotation algorithm, using more efficient mathematical methods or data structures to reduce computational and storage requirements.

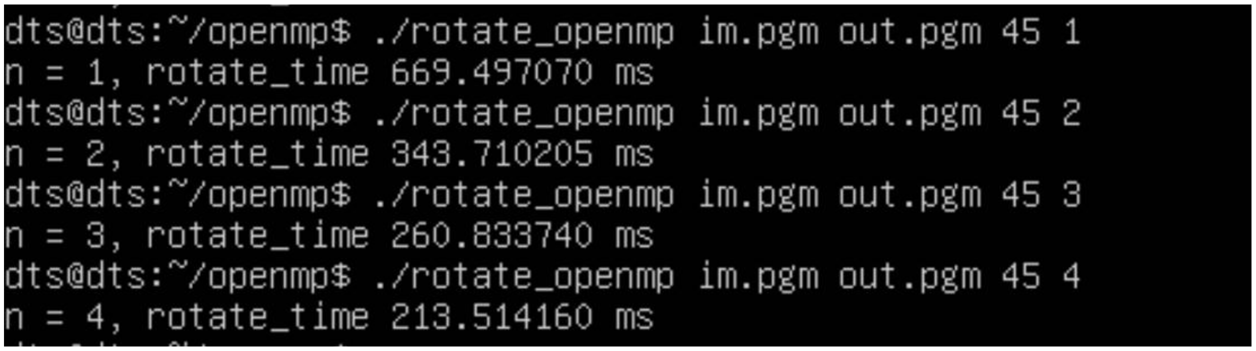
In summary, these data indicate that the MPI parallel implementation is effective in improving computational speed, but attention needs to be paid to issues such as communication overhead and load imbalance when increasing the number of processes. These factors are crucial in influencing parallel computing performance.

The time of the core algorithm of OpenMP



```
dts@dts:~$ cd openmp
dts@dts:~/openmp$ ls
im.pgm  Makefile  rotate_openmp.c
dts@dts:~/openmp$ gcc -g -Wall -fopenmp -o rotate_openmp rotate_openmp.c -lm
dts@dts:~/openmp$ ls
im.pgm  Makefile  rotate_openmp  rotate_openmp.c
```

These commands demonstrates the compilation and execution of an OpenMP-enabled C program for image rotation. The user navigates to a directory named "openmp" using the cd command and lists the contents, revealing files such as im.pgm, Makefile, and rotate_openmp.c. The user then compiles the C program with the GNU Compiler Collection (GCC), enabling OpenMP support for parallelization. The resulting executable, named rotate_openmp, is generated. The ls command is used again to confirm the creation of the executable. The directory appears to contain both source code and necessary files, facilitating the compilation and execution of the OpenMP program for image rotation.



```
dts@dts:~/openmp$ ./rotate_openmp im.pgm out.pgm 45 1
n = 1, rotate_time 669.497070 ms
dts@dts:~/openmp$ ./rotate_openmp im.pgm out.pgm 45 2
n = 2, rotate_time 343.710205 ms
dts@dts:~/openmp$ ./rotate_openmp im.pgm out.pgm 45 3
n = 3, rotate_time 260.833740 ms
dts@dts:~/openmp$ ./rotate_openmp im.pgm out.pgm 45 4
n = 4, rotate_time 213.514160 ms
```

This series of commands and output records the process of executing the OpenMP-enabled C program rotate_openmp in the terminal for image rotation, and observes its performance by changing the number of threads. In this experiment, multiple rotations were performed using different numbers of threads, each with an angle of 45 degrees.

First, run the program with the command **./rotate_openmp im.pgm out.pgm 45 1** to rotate the input image 45 degrees and limit the calculation to only 1 thread. The output shows that this rotation operation took approximately **669.50** milliseconds. Then, increase the number of threads to 2 through the command **./rotate_openmp im.pgm out.pgm 45 2.** The results show that the rotation time is significantly reduced to about **343.71** milliseconds, showing a significant improvement in computing speed by parallel computing. Then, further increasing the number of threads to 3 via the command **./rotate_openmp im.pgm out.pgm 45 3**, the output shows that the rotation time is reduced again to about **260.83** milliseconds. Finally, through the command **./rotate_openmp im.pgm out.pgm 45 4** using 4 threads to perform the rotation operation, the output shows that the rotation time continues to decrease, to about **213.51** milliseconds.

The speedup of a parallel program is defined as the ratio of the execution time in a serial run to the execution time in a parallel run. The running time of a single process on a parallel system is considered as the running time of a serial program.Therefore, I can use the data obtained above to calculate the speedup for process counts of 2, 3, and 4, respectively.

When the number of processes is 2, **S = 669.50 / 343.71 = 1.95**

When the number of processes is 3, **S = 669.50 / 260.83 = 2.57**

When the number of processes is 4, **S = 669.50 / 213.51 = 3.14**

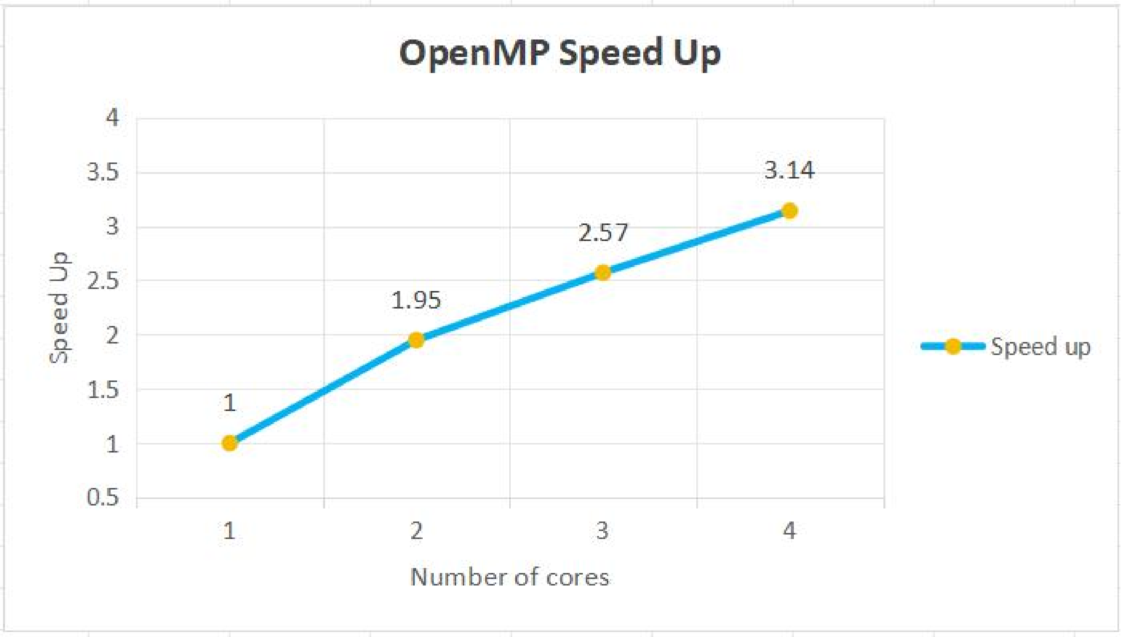
The efficiency of a parallel program is defined as the speedup divided by the corresponding number of processors or cores.

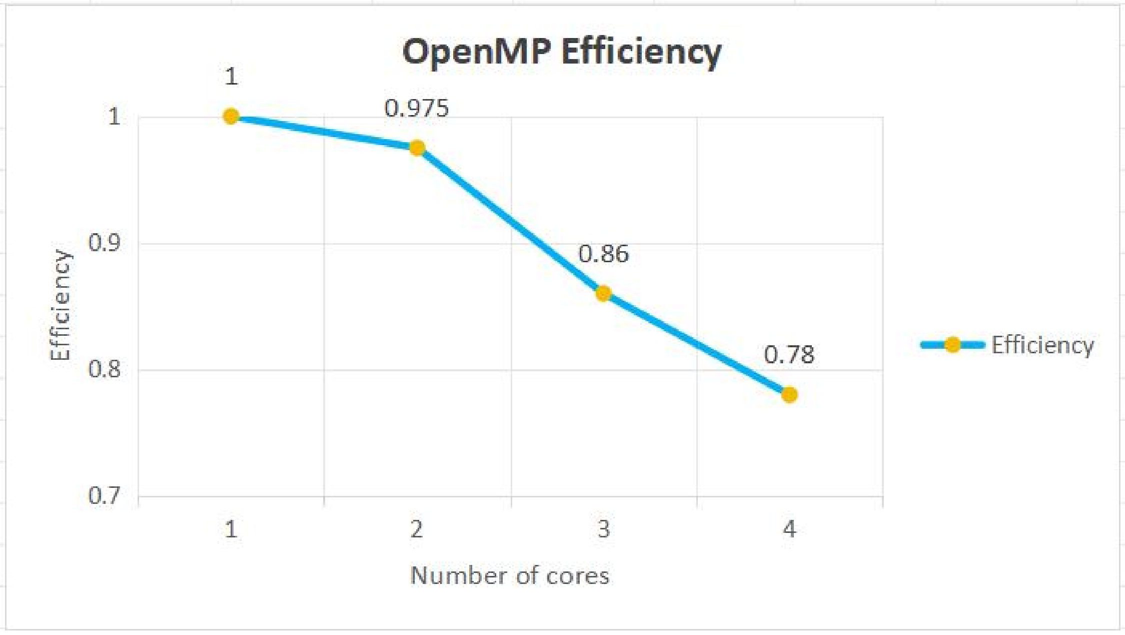
When the number of processes is 2, **E = 1.95 / 2 = 0.975**

When the number of processes is 3, **E = 2.57 / 3 = 0.86**

When the number of processes is 4, **E = 3.13 / 4 = 0.78**

From this, I can draw MPI's Speedup and Efficiency line plots, as shown below





Xi'an Jiaotong-Liverpool University 西交利物浦大學

Analyse the performance of OpenMP

OpenMP is a parallel programming model that allows programs to execute in parallel on multi-core processors to improve performance. Based on the provided code and performance data, the following analysis can be made:

Speedup Analysis:

When the number of cores increases from 1 to 4, the speedup increases from 1 to 3.14, showing that OpenMP can effectively utilize multi-cores when parallelizing image rotation tasks. Ideally, the speedup should be proportional to the number of cores, i.e., the speedup should be 4 in the case of 4 cores. However, the actual speedup is lower than the ideal value, indicating that there are some nonlinear increases, possibly due to parallel overhead or workload imbalance among cores. The increasing trend of speedup shows that OpenMP can utilize additional cores to reduce execution time when parallelizing image processing tasks, but the performance improvement decreases as the number of cores increases. Efficiency Analysis:

Efficiency measures the degree of utilization of each core relative to its maximum potential, with an ideal efficiency of 1. The efficiency is 0.975 with 2 cores, and drops to 0.78 with 4 cores, indicating that the efficiency of each core is decreasing as the number of cores increases. The decrease in efficiency may be due to parallel computing overheads such as thread management, synchronization, and memory access latency. In a multi-core environment, not all work can be perfectly divided, and some inherent serial parts limit the improvement of efficiency. Code Analysis:

The code uses OpenMP's #pragma omp parallel for schedule(static) collapse(2) to parallelize the image rotation operation. The schedule(static) scheduling strategy assumes that the workload of all iterations is evenly distributed, if this assumption does not hold, it may lead to load imbalance. The rotation algorithm includes computationally intensive mathematical operations, especially trigonometric functions, which may become a performance bottleneck.

Memory access is another important factor, the code uses calloc to allocate memory for the rotated image data, and memory access in a multi-threaded environment needs to be effectively managed to avoid performance loss. Performance Optimization Suggestions:

Load Balancing: Consider using schedule(dynamic) or other more flexible scheduling strategies to better handle uneven workloads. Memory Access Optimization: Optimize the algorithm to reduce data dependencies, ensure data locality to reduce cache misses and improve memory access efficiency. Algorithm Optimization: Use fast math libraries to optimize the calculation of trigonometric functions, or use approximation algorithms to speed up the calculation process. Performance Measurement: Use high-precision timing tools to more accurately assess execution time, and adjust the number of threads and optimization strategies based on this.

Overall, the OpenMP implementation has improved processing speed, but has not achieved the ideal full linear speedup. This may be due to the workload not being evenly distributed among processors, or due to memory access patterns and other system overheads. Between 2 cores and 3 cores, the increase in speedup begins to slow down, which may indicate the presence of some parallelism bottlenecks, such as data dependencies, memory bandwidth limitations, or the overhead of thread creation and destruction.

Compare & Contrast

Performance metrics

MPI and OpenMP are two major parallel programming models, each suitable for different hardware and application scenarios. In terms of performance metrics, these two technologies have their own characteristics.

Judging from the speedup and efficiency provided, the speedup and efficiency of MPI decrease as the number of processes increases. This may be due to communication overhead and load imbalance. OpenMP has better speedup and efficiency than MPI for the same number of processes. This may be due to the reduced communication overhead of shared memory access.

Performance characteristics of MPI:

Scalability: MPI is designed for distributed computing, suitable for large-scale parallel processing, especially when it is necessary to span multiple nodes. Its scalability is limited by network bandwidth and latency, but in a high-performance computing environment, it can scale to tens of thousands of processors.

Speedup: Due to the involvement of network communication between nodes, the speedup of MPI may be affected by network overhead, especially in communication-intensive applications. This means that the speedup may increase as the number of processors increases, but it is not linear.

Efficiency: In MPI programs, the balance between communication and computation is crucial for achieving high efficiency. Efficiency may be reduced due to network communication, especially in applications where data is frequently exchanged.

Communication Overhead: MPI needs to explicitly manage data transmission and message passing, which is especially important in distributed memory systems. Therefore, optimizing communication patterns and reducing data transmission volume are key to improving performance.

Performance characteristics of OpenMP:

Scalability: OpenMP is mainly used for shared memory systems, such as multi-core processors and multiprocessor systems. It can efficiently scale on multiple cores of a single node, but it may be limited by the memory bandwidth and the number of cores of a single system.

Speedup: OpenMP can usually achieve high speedup on shared memory architectures, because the communication overhead between threads is low. However, when the number of cores increases, memory bandwidth and cache contention may become limiting factors.

Efficiency: In shared memory systems, OpenMP can usually achieve higher efficiency because all threads can access shared memory, thereby reducing communication overhead. But synchronization mechanisms, such as barriers and locks, may introduce additional overhead when there are many threads.

Communication Overhead: Compared with MPI, the communication overhead of OpenMP is usually smaller, but synchronization between threads can still cause significant overhead.

In summary, MPI is suitable for scenarios where computation and data can be distributed across multiple nodes, while OpenMP is suitable for applications where data is concentrated in a single shared memory space. MPI provides fine control over distributed computing, but programming and debugging may be more complex. OpenMP is simpler to program in the shared memory model, but may not be suitable for applications that need to be parallel processed across multiple nodes.

Hardware limitation

From the perspective of hardware limitations, MPI and OpenMP have different requirements and constraints on hardware. These differences are mainly determined by their different design goals and operating environments.

Hardware limitations of MPI:

Distributed Memory Architecture: MPI is designed to run on different computing nodes with distributed memory. It does not rely on shared memory architecture, so it can be used in a wide range of distributed systems, including local clusters, supercomputers, and cloud computing environments.

Network Dependence: The performance of MPI largely depends on the quality of network communication between nodes. High-performance networks (such as InfiniBand) can significantly improve the performance of MPI programs, while standard Ethernet may become a performance bottleneck.

Scalability: MPI programs can scale across multiple nodes, making them not limited by the memory and processing power of a single node. However, hardware heterogeneity may lead to load imbalance and optimization difficulties.

Node Resources: MPI programs require enough memory on each node to store a copy of the process data. As the number of nodes increases, the resource utilization of each node may become unbalanced.

Hardware limitations of OpenMP:

Shared Memory Architecture: OpenMP is designed for shared memory systems, usually a single multi-core or multiprocessor machine. This means that OpenMP is limited by the physical memory and the number of processors of a single system.

Memory Bandwidth and Latency: In shared memory systems, memory bandwidth and latency are the main factors limiting the performance of OpenMP. As the number of parallel threads increases, memory contention may increase, leading to performance degradation.

Core Number: The scalability of OpenMP programs is limited by the number of cores available on the machine. This means that after the cores are saturated, adding more threads will not bring performance improvement.

Cache Consistency: Multithreaded parallel execution may cause cache consistency problems, especially when multiple threads modify the same data. To maintain consistency, expensive synchronization operations may be required, which will affect performance.

In summary, the main differences between MPI and OpenMP in terms of hardware limitations lie in their design goals and applicable hardware environments. MPI is suitable for large-scale distributed environments that are not limited by the resources of a single machine, while OpenMP is more suitable for shared memory systems, limited by the number of cores and memory resources of a single system.

Ease of Debugging & Testing

In terms of ease of debugging and testing, MPI and OpenMP show significant differences, mainly due to their parallel programming models and operating environments.

Debugging and Testing of MPI:

Complexity: Since MPI programs typically run in multiple independent processes, possibly distributed across different physical machines, debugging is relatively complex. To effectively debug MPI programs, developers need to be able to track and manage communication between different processes. Debugging sessions must be launched separately for each MPI process, as each process runs independently using its own memory space.

Concurrency Errors: In MPI programs, common issues include deadlocks, race conditions, and communication errors. These problems are difficult to detect and reproduce, as they may only occur under specific parallel states.

Environment Dependence: Testing of MPI programs may need to be conducted in actual multi-node environments, which increases the complexity of testing, especially in resource-constrained or inconsistently configured environments.

Debugging and Testing of OpenMP:

Simplicity: OpenMP programs run multiple threads in a single shared memory space, making the debugging process more intuitive. Developers can use standard debugging tools for debugging.

Thread Issues: Although OpenMP simplifies the parallelization process, debugging can still be complex, as it needs to handle issues such as thread synchronization and race conditions. However, these issues are usually easier to track than communication problems between distributed processes.

Test Environment: Testing OpenMP programs usually does not require complex setup or a dedicated test environment, as they run on a single machine. This makes building and maintaining the test environment relatively easy.

In summary, due to its distributed nature, MPI is usually more challenging than OpenMP in debugging and testing. MPI needs to handle communication issues across multiple processes and computing nodes, while OpenMP mainly deals with multi-thread synchronization issues within a single system.

Communication & Synchronization

In parallel programming, communication and synchronization are core issues, as they determine how different computational tasks exchange information and how to maintain the consistency and correctness of operations. MPI and OpenMP have fundamental differences in these two aspects.

Communication and Synchronization in MPI:

Communication Mechanism: MPI is a message-passing parallel programming model that requires explicit inter-process communication. It provides a rich set of communication functions, including point-to-point and collective communication operations, allowing programmers to finely control how data is transmitted between different processes.

Synchronization Operations: Synchronization in MPI is usually achieved through communication itself, for example, in send and receive operations. MPI also provides explicit synchronization mechanisms, such as MPI_Barrier, to implement synchronization between different processes.

Distributed Memory Model: In MPI, each process has its own memory space and needs to share data through message passing, which is necessary in distributed memory systems. Therefore, communication overhead is often a key determinant of the performance of MPI programs.

Communication and Synchronization in OpenMP:

Shared Memory Communication: OpenMP is based on a shared memory model, and communication between threads is implicit, mainly implemented by accessing shared variables. This reduces programming complexity but also increases the risk of data access contention.

Synchronization Constructs: OpenMP provides several synchronization directives, such as #pragma omp barrier, #pragma omp critical, #pragma omp atomic, etc., to control the execution order and data access between threads, making thread synchronization operations simpler.

Cache Consistency: Since all threads share the same memory, OpenMP programs may face cache consistency issues, especially when multiple threads update the same memory location. OpenMP's synchronization directives help manage this situation, but may introduce performance overhead.

Overall, MPI’s communication and synchronization are explicit, giving programmers high control, and are suitable for distributed memory models, where inter-process communication is the main performance bottleneck. On the other hand, OpenMP provides implicit communication and concise synchronization mechanisms in the shared memory model, suitable for quickly developing and maintaining multi-threaded programs, especially on a single node. However, these advantages of OpenMP may also be affected by competition and synchronization overhead between threads.

Choose preferred parallel technique

Considering performance, scalability, ease of debugging and testing, as well as communication and synchronization strategies, I lean towards choosing OpenMP as the preferred parallel computing method. In terms of performance, OpenMP exhibits significantly higher speedup and efficiency compared to MPI. The thread-based parallelization in OpenMP, operating within a shared memory model, proves effective in reducing communication overhead, while MPI introduces additional overhead due to explicit inter-process communication.

For scalability, OpenMP is limited to the available local cores but is relatively straightforward to use on a single machine. On the other hand, MPI demonstrates notable flexibility by efficiently utilizing computation resources across a network, making it more suitable for large-scale distributed computing tasks that extend beyond the boundaries of a single machine.

Regarding ease of debugging and testing, OpenMP is relatively simple to operate when running within a single process, resembling procedures for a serial version. However, debugging MPI code introduces complexities, requiring separate debugging sessions for each independent MPI process, thus demanding careful synchronization during debugging.

In terms of communication and synchronization, OpenMP's implicit shared memory model simplifies communication and synchronization through structures like barriers and critical regions. In contrast, MPI's explicit communication mechanism necessitates explicit coding for data transfer between processes, which is achieved through blocking communication calls for synchronization. OpenMP's thread-based approach allows threads to have both local and shared data in memory, while MPI maintains all data as process-local, requiring inter-process communication for information exchange.

While OpenMP has limitations in scalability, especially in large-scale distributed computing scenarios, its simplicity and convenience make it an effective choice for medium-sized parallel computing tasks. On the other hand, MPI's distributed nature and flexible scalability make it more suitable for larger-scale and high-performance computing requirements.

In conclusion, OpenMP appears to be a simpler and more effective choice, especially for tasks of medium complexity.

3. Reflection

One challenge for me is in the MPI implementation, how to combine the result together?

In parallel computing, implementing with MPI is a challenge, especially in the phase of result merging. This is because the rotation operation will change the order of pixels in memory. In many other MPI programs, the process ranked 0 can simply collect the rotated pixel parts from other MPI processes and combine them to get the final picture. However, in this case, it is not feasible to do so.

To solve this problem, I proposed a solution, which is to bundle the position information with pixel values. In this way, the process ranked 0 can finally compose the resulting picture based on these data. Although this method may require more computing and memory resources, it ensures the correct correspondence and sorting of pixels during the rotation and merging process.

A potential advantage of this method is its universality. It can be applied not only to image processing but also to any scenario that needs to process and reorder data in a parallel environment. For example, it can be used to handle large-scale scientific simulation data, where the spatial position information of the data is crucial for the correctness of the results.

Another challenge for me is that when programming image rotation using OpenMP, a key challenge is how to effectively divide the image rotation task into subtasks that can be processed in parallel by multiple threads, ensuring that each thread contributes equally and efficiently to the computing power. The image rotation task involves transforming the coordinates of each pixel, which is a computationally intensive task that can be parallelized.

The solution involves using OpenMP's parallel loop. By applying #pragma omp parallel for in the code, the compiler is instructed to automatically distribute the iterations of the loop to multiple threads. This method simplifies the task allocation process, avoiding the manual management of thread creation and destruction. Additionally, by using the collapse(2) directive, the outer and inner loops can be combined into a larger iteration space, allowing OpenMP to distribute work more flexibly among threads. Loop collapsing helps reduce the imbalance in work distribution between threads, especially when the inner loop is small and the outer loop is large.

Furthermore, choosing schedule(static) as the scheduling mechanism allows for the static allocation of iterations to threads before runtime. Static scheduling assumes that the workload of each iteration is roughly the same, so each thread receives a roughly equal number of iterations. This is very suitable in the context of image rotation, as the computational requirement for processing each pixel is essentially uniform.

Adopting this parallel design and data partitioning method, the image rotation task can be effectively divided and allocated to multiple threads, achieving efficient parallel computation.

Each thread processes a similar number of pixels, reducing workload disparities between threads and improving the overall efficiency of parallel execution.