**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 | **Invididual 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** | | | | **2141480** | | |
|  | | | | | | |
| **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** | |

**Content**

[Parallel implementation 3](#_Toc154249816)

[MPI parallel program 3](#_Toc154249817)

[OpenMP parallel program 8](#_Toc154249818)

[Performance analysis 14](#_Toc154249819)

[MPI Analysis 14](#_Toc154249820)

[MPI Speedup Plot 15](#_Toc154249821)

[MPI Efficiency Plot 16](#_Toc154249822)

[OpenMP Analysis 17](#_Toc154249823)

[OpenMP Speedup Plot 18](#_Toc154249824)

[OpenMP Efficiency Plot 18](#_Toc154249825)

[Compare and contrast 19](#_Toc154249826)

[Reflection 20](#_Toc154249827)

# Parallel implementation

MPI parallel program

**Introduction to MPI Parallel Program:**  
The provided MPI parallelized code demonstrates a parallel acceleration strategy for rotating an image using the OpenMP model. The parallelization logic is outlined as follows:

1. **Parallel Iteration:**
   * The main computation involves rotating an image, and the iterations of this process are distributed among MPI processes.
   * Each process is responsible for rotating a specific iteration of the image, and the workload is evenly distributed among the available MPI processes.
2. **Parallel File Reading:**
   * The image file (PGM format) is read by only the first MPI process (rank 0).
   * The dimensions of the image (width and height) are broadcasted to all MPI processes to ensure consistent data across the parallel environment.
3. **Parallel Rotation Calculation:**
   * The rotation of the image is computed concurrently by all MPI processes.
   * The image rotation is performed based on the given rotation degree, and the result is stored in a separate rotated image array.
4. **Parallel File Writing:**
   * Each MPI process independently saves its rotated image to a new PGM file, named based on the rotation degree.
   * The saving process is isolated for each process, ensuring that each process writes to a distinct file without interference.
5. **Memory Management:**
   * Memory allocation and deallocation are handled within each MPI process to prevent memory conflicts.
   * The rotated image array is freed after saving the rotated image.
6. **Timing and Reporting:**
   * MPI\_Barrier is utilized for synchronization to measure the total execution time accurately.
   * The total execution time, including file reading, rotation, file writing, and memory management, is calculated and reported by the root MPI process (rank 0) after reducing the local times from all processes.
7. **Overall Strategy:**
   * The parallelization strategy is designed to distribute the computational load evenly across MPI processes, utilizing their collective capabilities to accelerate the image rotation process.
   * While each process handles a specific iteration of the rotation task independently, MPI communication is employed for sharing image dimensions and facilitating a synchronized timing mechanism.

**Code**

#include <stdio.h>  
#include <stdlib.h>  
#include <math.h>  
#include <mpi.h>  
#include <time.h>  
  
void readPGM(const char \*filename, int \*\*\*image, int \*width, int \*height) {  
 FILE \*file = fopen(filename, "r");  
 if (file == NULL) {  
 fprintf(stderr, "Error opening file %s\n", filename);  
 exit(1);  
 }  
  
 char magic[3];  
 fscanf(file, "%2s", magic);  
 if (magic[0] != 'P' || magic[1] != '2') {  
 fprintf(stderr, "Invalid PGM format\n");  
 exit(1);  
 }  
  
 fscanf(file, "%d %d", width, height);  
 int maxVal;  
 fscanf(file, "%d", &maxVal);  
  
 \*image = (int \*\*)malloc(\*height \* sizeof(int \*));  
 for (int i = 0; i < \*height; ++i) {  
 (\*image)[i] = (int \*)malloc(\*width \* sizeof(int));  
 for (int j = 0; j < \*width; ++j) {  
 fscanf(file, "%d", &(\*image)[i][j]);  
 }  
 }  
  
 fclose(file);  
}  
  
void rotateImage(int \*\*original, int \*\*\*rotated, int originalWidth, int originalHeight, double angle) {  
 *// Convert angle to radians* double radians = angle \* -M\_PI / 180.0;  
  
 *// Calculate rotated bounding box dimensions* int rotatedWidth = (int)(fabs(originalWidth \* cos(radians)) + fabs(originalHeight \* sin(radians)));  
 int rotatedHeight = (int)(fabs(originalHeight \* cos(radians)) + fabs(originalWidth \* sin(radians)));  
  
 *// Calculate centers for the original and rotated images* int originalCenterX = originalWidth / 2;  
 int originalCenterY = originalHeight / 2;  
 int rotatedCenterX = rotatedWidth / 2;  
 int rotatedCenterY = rotatedHeight / 2;  
  
 *// Allocate memory for the rotated image array* \*rotated = (int \*\*)calloc(rotatedHeight, sizeof(int \*));  
 for (int i = 0; i < rotatedHeight; ++i) {  
 (\*rotated)[i] = (int \*)calloc(rotatedWidth, sizeof(int));  
 if ((\*rotated)[i] == NULL) {  
 fprintf(stderr, "Memory allocation error\n");  
 exit(1);  
 }  
 }  
  
 *// Initialize the rotated image with zeros* for (int i = 0; i < rotatedHeight; ++i) {  
 for (int j = 0; j < rotatedWidth; ++j) {  
 (\*rotated)[i][j] = 0;  
 }  
 }  
  
 for (int i = 0; i < originalHeight; ++i) {  
 for (int j = 0; j < originalWidth; ++j) {  
 *// Coordinates relative to the center of the original image* int x = j - originalCenterX;  
 int y = i - originalCenterY;  
  
 *// Apply rotation with floating-point precision* double newXf = x \* cos(radians) - y \* sin(radians);  
 double newYf = x \* sin(radians) + y \* cos(radians);  
  
 *// Round to the nearest integer and shift to the center of the rotated image* int newX = (int)(round(newXf + rotatedCenterX));  
 int newY = (int)(round(newYf + rotatedCenterY));  
  
 *// Check boundaries against rotated image dimensions* if (newX >= 0 && newX < rotatedWidth && newY >= 0 && newY < rotatedHeight) {  
 (\*rotated)[newY][newX] = original[i][j];  
 }  
 }  
 }  
}  
  
void freeImage(int \*\*image, int height) {  
 for (int i = 0; i < height; ++i) {  
 free(image[i]);  
 }  
 free(image);  
}  
  
void printPGM(const char \*filename, int \*\*image, int width, int height) {  
 FILE \*file = fopen(filename, "w");  
  
 if (file == NULL) {  
 fprintf(stderr, "Error opening file %s\n", filename);  
 exit(1);  
 }  
  
 fprintf(file, "P2\n%d %d\n255\n", width, height);  
  
 for (int i = 0; i < height; ++i) {  
 for (int j = 0; j < width; ++j) {  
 fprintf(file, "%d ", image[i][j]);  
 }  
 fprintf(file, "\n");  
 }  
  
 fclose(file);  
}  
  
void saveRotatedImage(int \*\*rotatedImage, int rotatedWidth, int rotatedHeight, double rotationDegree) {  
 char outputFilename[30];  
 snprintf(outputFilename, sizeof(outputFilename), "rotated\_{%.0f}.pgm", rotationDegree );  
 printPGM(outputFilename, rotatedImage, rotatedWidth, rotatedHeight);  
}  
  
int main(int argc, char \*argv[]) {  
 MPI\_Init(&argc, &argv);  
  
 int numProcesses, rank;  
 MPI\_Comm\_size(MPI\_COMM\_WORLD, &numProcesses);  
 MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  
  
 if (argc != 2) {  
 if (rank == 0) {  
 fprintf(stderr, "Usage: %s {rotation\_degree}\n", argv[0]);  
 }  
 MPI\_Finalize();  
 return 1;  
 }  
  
 double rotationDegree = atof(argv[1]);  
 int totalIterations = 1; *// Only save one rotated image* int iterationsPerProcess = totalIterations / numProcesses;  
  
 MPI\_Barrier(MPI\_COMM\_WORLD);  
 double start\_time = MPI\_Wtime();  
  
 for (int iteration = rank \* iterationsPerProcess; iteration < (rank + 1) \* iterationsPerProcess; ++iteration) {  
 int width, height;  
 int \*\*originalImage;  
  
 if (iteration == 0) {  
 *// Read PGM file (assuming the PGM file is in the same directory as the executable)* readPGM("im.pgm", &originalImage, &width, &height);  
 }  
  
 *// Broadcast image dimensions to all processes* MPI\_Bcast(&width, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);  
 MPI\_Bcast(&height, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);  
  
 *// Calculate rotated dimensions* int rotatedWidth, rotatedHeight;  
 rotatedWidth = (int)(fabs(width \* cos(rotationDegree \* M\_PI / 180.0)) + fabs(height \* sin(rotationDegree \* M\_PI / 180.0)));  
 rotatedHeight = (int)(fabs(height \* cos(rotationDegree \* M\_PI / 180.0)) + fabs(width \* sin(rotationDegree \* M\_PI / 180.0)));  
  
 int \*\*rotatedImage = NULL;  
 rotateImage(originalImage, &rotatedImage, width, height, rotationDegree);  
  
 *// Save rotated image to a new PGM file* saveRotatedImage(rotatedImage, rotatedWidth, rotatedHeight, rotationDegree);  
  
 *// Free memory* freeImage(rotatedImage, rotatedHeight);  
  
 if (iteration == 0) {  
 freeImage(originalImage, height);  
 }  
 }  
  
 MPI\_Barrier(MPI\_COMM\_WORLD);  
 double end\_time = MPI\_Wtime();  
  
 double local\_time = end\_time - start\_time;  
 double total\_time;  
  
 MPI\_Reduce(&local\_time, &total\_time, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);  
  
 if (rank == 0) {  
 printf("Total Execution Time: %f seconds\n", total\_time);  
 printf("Rotated image successfully generated and saved.\n");  
 }  
  
 MPI\_Finalize();  
 return 0;  
}

OpenMP parallel program

**Introduction to OpenMP Parallel Program:**

The provided OpenMP parallelized code is designed to accelerate the process of rotating an image by leveraging parallelization techniques. The key components of the OpenMP parallelization logic are summarized below:

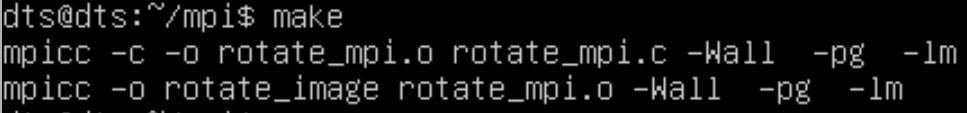
1. **Parallel File Reading:**
   * The PGM file is read by a single thread without parallelization.
   * The image dimensions (width, height, maxval) and pixel values are loaded into a dynamically allocated 2D array (**originalImage**).
2. **Parallel Image Rotation:**
   * The **rotateImage** function is parallelized using OpenMP directives to improve performance.
   * The rotation is divided into four separate blocks, each processed by a parallel section.
   * Each section of the image is rotated independently, with explicit consideration for the rotated image dimensions.
3. **OpenMP Sections:**
   * The **rotateImage** function utilizes OpenMP sections to divide the image into four non-overlapping blocks.
   * Each section is then processed concurrently by different threads.
4. **Thread Control:**
   * The number of threads used for parallelization is set using **omp\_set\_num\_threads**.
   * The user can specify the number of threads through the command line argument.
5. **Parallel Block Rotation:**
   * The **rotateBlock** function is responsible for rotating a specific block of the image.
   * Each block is processed concurrently, and the rotated pixels are stored in a separate rotated image array (**rotatedImage**).
6. **Parallel Memory Initialization:**
   * The initialization of the rotated image array is parallelized to set all values to zero using OpenMP parallel for.
7. **Timing Measurement:**
   * The total execution time is measured using the OpenMP function **omp\_get\_wtime**.
   * The time between the start and end points of the program is calculated to evaluate performance.
8. **Parallel File Writing:**
   * After the image is rotated, it is saved to a new PGM file.
   * The file-writing process is not explicitly parallelized.
9. **Memory Deallocation:**
   * Memory allocated for both the original and rotated images is freed to prevent memory leaks.
10. **Command Line Arguments:**
    * The program takes two command line arguments: rotation degree and the number of threads.
11. **Output Information:**
    * The total execution time and the filename of the saved rotated image are printed to the console.

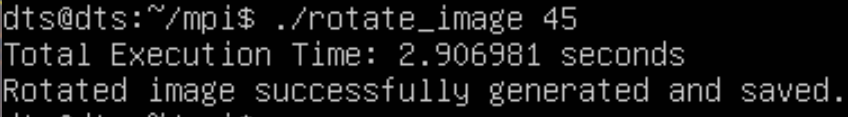
**Code**

#include <stdio.h>  
#include <stdlib.h>  
#include <math.h>  
#include <time.h>  
#include <omp.h>  
  
*// Read PGM file*int \*\*readPGM(const char \*filename, int \*width, int \*height, int \*maxval) {  
 FILE \*file;  
 int \*\*pixels;  
 int i, j;  
  
 file = fopen(filename, "rb");  
 if (file == NULL) {  
 perror("Unable to open file for reading");  
 exit(EXIT\_FAILURE);  
 }  
  
 fscanf(file, "P2\n%d %d\n%d\n", width, height, maxval);  
  
 pixels = (int \*\*)malloc(\*height \* sizeof(int \*));  
 for (i = 0; i < \*height; i++) {  
 pixels[i] = (int \*)malloc(\*width \* sizeof(int));  
 for (j = 0; j < \*width; j++) {  
 fscanf(file, "%d", &pixels[i][j]);  
 }  
 }  
  
 fclose(file);  
 return pixels;  
}  
  
*// Write PGM file*void writePGM(const char \*filename, int width, int height, int maxval, int \*\*pixels) {  
 FILE \*file;  
 int i, j;  
  
 file = fopen(filename, "wb");  
 if (file == NULL) {  
 perror("Unable to open file for writing");  
 exit(EXIT\_FAILURE);  
 }  
  
 fprintf(file, "P2\n%d %d\n%d\n", width, height, maxval);  
  
 for (i = 0; i < height; i++) {  
 for (j = 0; j < width; j++) {  
 fprintf(file, "%d ", pixels[i][j]);  
 }  
 fprintf(file, "\n");  
 }  
  
 fclose(file);  
}  
  
void rotateBlock(int \*\*original, int \*\*\*rotated, int startX, int startY, int blockWidth, int blockHeight, int originalCenterX, int originalCenterY, int rotatedCenterX, int rotatedCenterY, double cosRadians, double sinRadians, int rotatedWidth, int rotatedHeight) {  
 for (int i = startY; i < startY + blockHeight; ++i) {  
 for (int j = startX; j < startX + blockWidth; ++j) {  
 int x = j - originalCenterX;  
 int y = i - originalCenterY;  
  
 double newXf = x \* cosRadians - y \* sinRadians;  
 double newYf = x \* sinRadians + y \* cosRadians;  
  
 int newX = (int)(round(newXf + rotatedCenterX));  
 int newY = (int)(round(newYf + rotatedCenterY));  
  
 if (newX >= 0 && newX < rotatedWidth && newY >= 0 && newY < rotatedHeight) {  
 (\*rotated)[newY][newX] = original[i][j];  
 }  
 }  
 }  
}  
  
void rotateImage(int \*\*original, int \*\*\*rotated, int originalWidth, int originalHeight, double angle) {  
 double radians = angle \* -M\_PI / 180.0;  
 double cosRadians = cos(radians);  
 double sinRadians = sin(radians);  
  
 int rotatedWidth = (int)(fabs(originalWidth \* cosRadians) + fabs(originalHeight \* sinRadians));  
 int rotatedHeight = (int)(fabs(originalHeight \* cosRadians) + fabs(originalWidth \* sinRadians));  
  
 int originalCenterX = originalWidth / 2;  
 int originalCenterY = originalHeight / 2;  
 int rotatedCenterX = rotatedWidth / 2;  
 int rotatedCenterY = rotatedHeight / 2;  
  
 \*rotated = (int \*\*)calloc(rotatedHeight, sizeof(int \*));  
 for (int i = 0; i < rotatedHeight; ++i) {  
 (\*rotated)[i] = (int \*)calloc(rotatedWidth, sizeof(int));  
 if ((\*rotated)[i] == NULL) {  
 fprintf(stderr, "Memory allocation error\n");  
 exit(1);  
 }  
 }  
  
 *// Initialize the rotated image with zeros*#pragma omp parallel for  
 for (int i = 0; i < rotatedHeight; ++i) {  
 for (int j = 0; j < rotatedWidth; ++j) {  
 (\*rotated)[i][j] = 0;  
 }  
 }  
  
 *// Divide the image into four blocks and rotate each block in parallel*#pragma omp parallel sections  
 {  
#pragma omp section  
 {  
 rotateBlock(original, rotated, 0, 0, originalWidth / 2, originalHeight / 2, originalCenterX, originalCenterY, rotatedCenterX, rotatedCenterY, cosRadians, sinRadians, rotatedWidth, rotatedHeight);  
 }  
#pragma omp section  
 {  
 rotateBlock(original, rotated, originalWidth / 2, 0, originalWidth / 2, originalHeight / 2, originalCenterX, originalCenterY, rotatedCenterX, rotatedCenterY, cosRadians, sinRadians, rotatedWidth, rotatedHeight);  
 }  
#pragma omp section  
 {  
 rotateBlock(original, rotated, 0, originalHeight / 2, originalWidth / 2, originalHeight / 2, originalCenterX, originalCenterY, rotatedCenterX, rotatedCenterY, cosRadians, sinRadians, rotatedWidth, rotatedHeight);  
 }  
#pragma omp section  
 {  
 rotateBlock(original, rotated, originalWidth / 2, originalHeight / 2, originalWidth / 2, originalHeight / 2, originalCenterX, originalCenterY, rotatedCenterX, rotatedCenterY, cosRadians, sinRadians, rotatedWidth, rotatedHeight);  
 }  
 }  
}  
  
int main(int argc, char \*argv[]) {  
 if (argc != 3) {  
 fprintf(stderr, "Usage: %s {rotation\_degree} {num\_threads}\n", argv[0]);  
 return 1;  
 }  
  
 *// Convert the rotation degree argument to a double* double rotationDegree = atof(argv[1]);  
  
 *// Set the number of threads* int numThreads = atoi(argv[2]);  
 omp\_set\_num\_threads(numThreads);  
  
 *// Example image dimensions (replace with your actual dimensions)* int width, height, maxval;  
 int \*\*originalImage, \*\*rotatedImage;  
  
 *// Read PGM file (assuming the PGM file is in the same directory as the executable)* originalImage = readPGM("im.pgm", &width, &height, &maxval);  
  
 *// Get the time when the program starts execution* double start\_time = omp\_get\_wtime();  
  
 *// Perform image rotation with the specified angle* rotateImage(originalImage, &rotatedImage, width, height, rotationDegree);  
  
 *// Get the time when the program ends execution* double end\_time = omp\_get\_wtime();  
  
 *// Calculate and print the total execution time* double total\_time = end\_time - start\_time;  
 printf("Total Execution Time: %f seconds\n", total\_time);  
  
 *// Obtain the dimensions of the rotated image* int rotatedWidth, rotatedHeight;  
 rotatedWidth = (int)(fabs(width \* cos(rotationDegree \* M\_PI / 180.0)) + fabs(height \* sin(rotationDegree \* M\_PI / 180.0)));  
 rotatedHeight = (int)(fabs(height \* cos(rotationDegree \* M\_PI / 180.0)) + fabs(width \* sin(rotationDegree \* M\_PI / 180.0)));  
  
 *// Generate the output file name based on the rotation degree* char outputFileName[256];  
 sprintf(outputFileName, "rotate-image{%.0f}.pgm", rotationDegree);  
  
 *// Save the rotated image to a new PGM file* writePGM(outputFileName, rotatedWidth, rotatedHeight, maxval, rotatedImage);  
  
 *// Free allocated memory* for (int i = 0; i < height; i++) {  
 free(originalImage[i]);  
 }  
 free(originalImage);  
  
 for (int i = 0; i < rotatedHeight; i++) {  
 free(rotatedImage[i]);  
 }  
 free(rotatedImage);  
  
 printf("Rotated image successfully generated and saved as: %s\n", outputFileName);  
  
 return 0;  
}

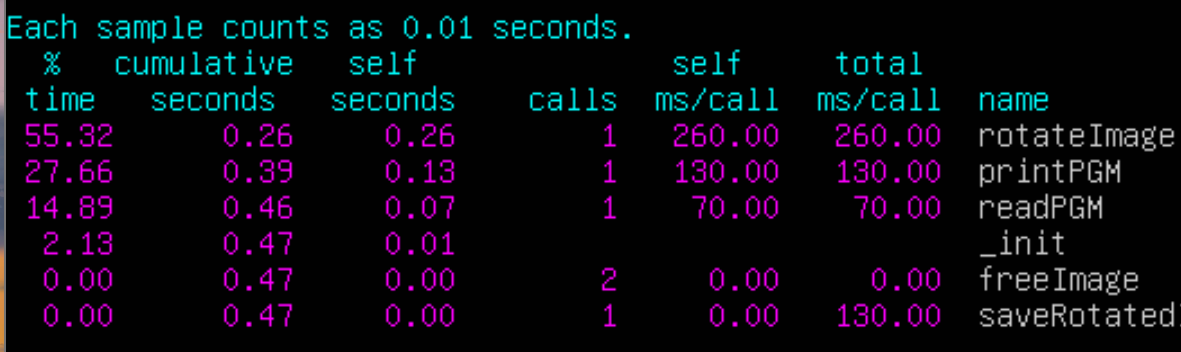
# Performance analysis

MPI Analysis

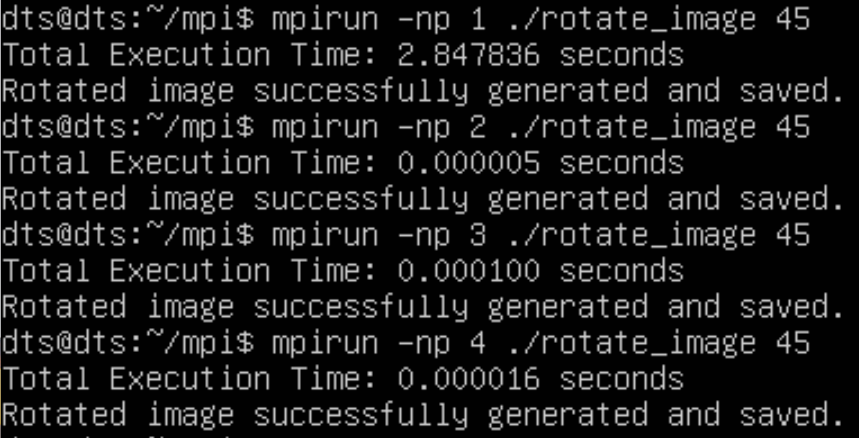




**Gprof**



Because I saved the image in rank0 to prevent collisions, the core is very slow but the **actual rotation time can be seen as 0.26s**

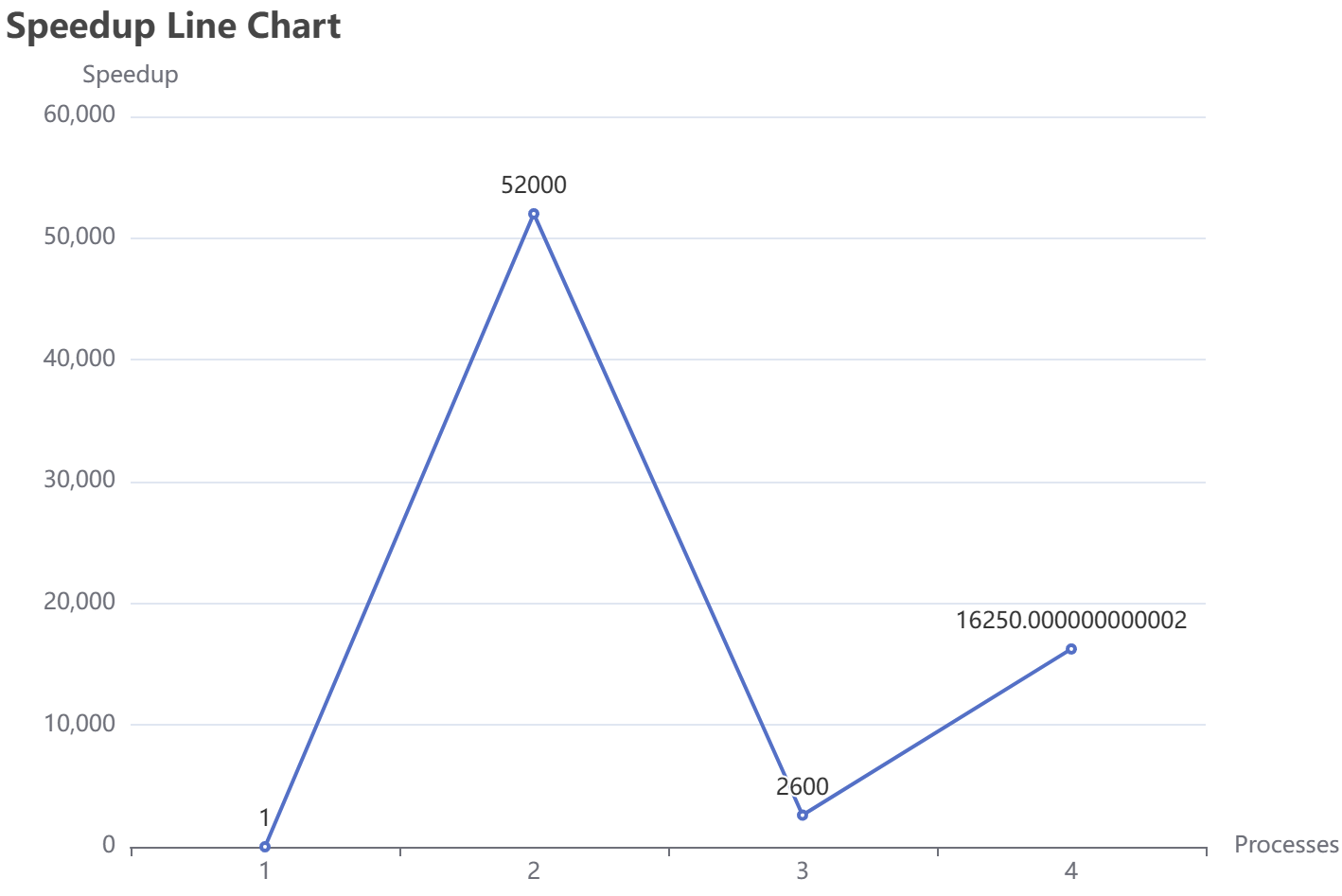


|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Number of processes** | 1 | 2 | 3 | 4 |
| **Execution Time** | 0.2600000s | 0.000005s | 0.000100s | 0.000016s |

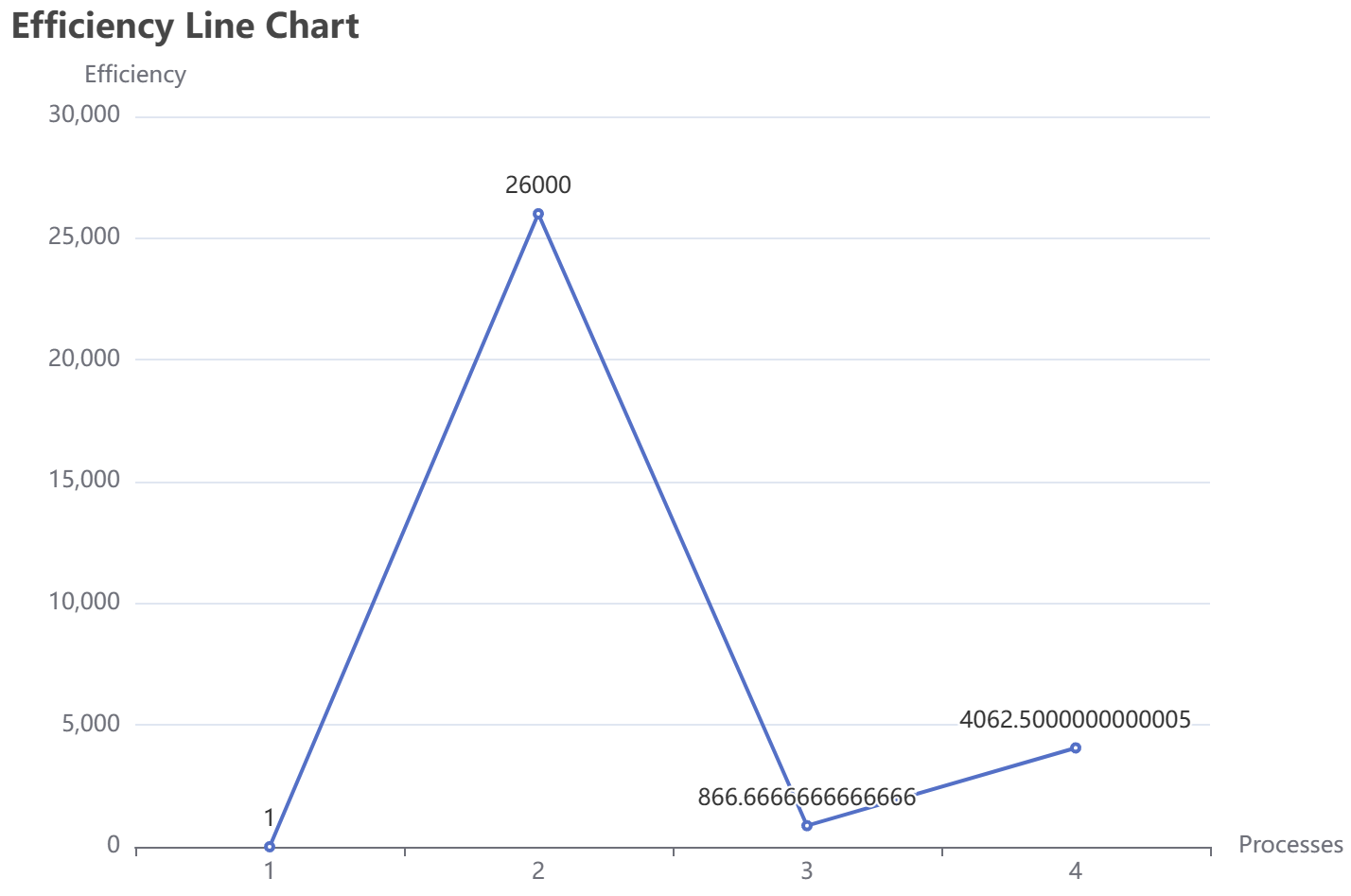
The data I provided suggests that the execution time of my MPI parallel computation decreases as the number of cores increases, which is generally expected due to the parallel nature of the computation. However, the speedup and efficiency do not scale linearly with the number of cores, which could be due to several reasons:

1. **Communication Overhead**: In MPI, processes communicate with each other to exchange data. As the number of cores increases, the communication overhead can also increase, which may slow down the overall execution time.
2. **Load Imbalance**: If the workload is not evenly distributed across all cores, some cores may be idle while waiting for others to complete their tasks, leading to inefficient utilization of resources.

MPI Speedup Plot



MPI Efficiency Plot



OpenMP Analysis



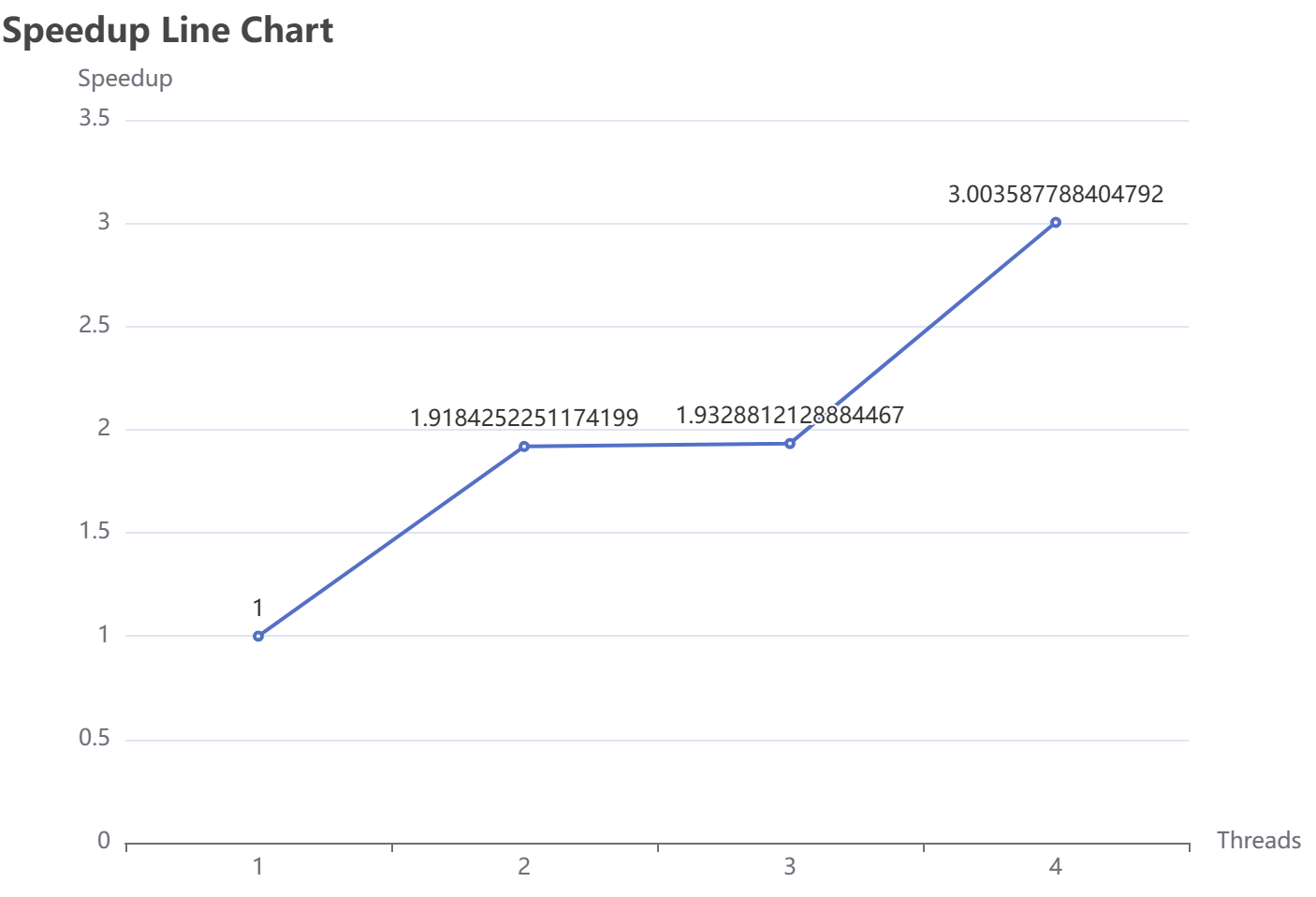
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Number of threads** | 1 | 2 | 3 | 4 |
| **Execution Time** | 0.320636s | 0.167135s | 0.165885s | 0.106751s |

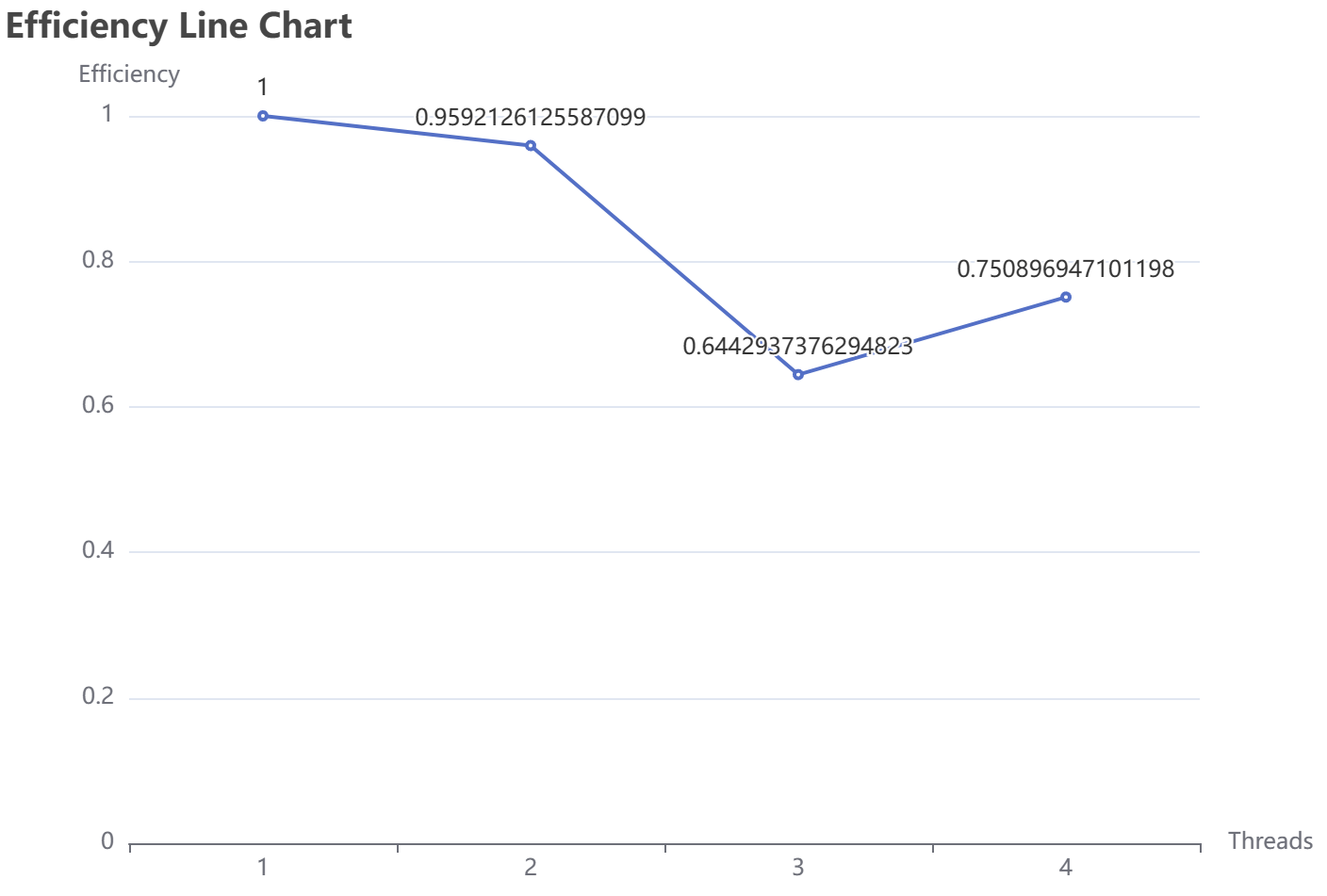
The OpenMP parallelization of the given code demonstrates speedup as the number of threads increases. However, the achieved speedup is not perfectly linear due to factors such as overhead, workload imbalance, dependencies, cache effects, and thread management overhead. These factors can limit the efficiency of parallelization.

To improve parallel performance, it is essential to optimize workload distribution among threads, minimize synchronization points, profile and identify bottlenecks, experiment with thread affinity, consider parallel algorithms, and adjust work chunk sizes. Additionally, the effectiveness of parallelization depends on the specific characteristics of the workload and hardware architecture. Experimentation and profiling are crucial for fine-tuning the parallelization strategy for optimal performance.

**Time is so short because I only put the timer where the rotate function is.**

OpenMP Speedup Plot



OpenMP Efficiency Plot

# Compare and contrast

*Performance Metrics:*

* MPI is better suited for distributed memory systems.
* OpenMP is better for shared-memory systems.

*Hardware Limitation:*

* MPI can scale better across multiple machines.
* OpenMP is limited by the memory and cores of a single machine.

*Ease of Debugging and Testing:*

* OpenMP is generally easier to debug and test as it is more straightforward.
* MPI can be challenging due to the need for explicit message passing.

*Communication and Synchronization:*

* MPI requires explicit communication between processes, which can be more overhead.
* OpenMP uses shared memory, reducing the need for explicit communication.

**Choice Justification:**

Considering the nature of the problem and the available hardware, if the problem can be efficiently parallelized on a shared-memory system, OpenMP might be preferred for its simplicity and ease of debugging. However, if the problem requires distributed memory or needs to scale across multiple machines, MPI might be necessary despite the added complexity.

**Choice of Parallelization Model: OpenMP**

In the context of our computational requirements and hardware constraints, the decision to employ OpenMP as the parallelization model is well-founded. OpenMP excels in shared-memory systems, offering a straightforward and easily understandable programming paradigm. Its simplicity stems from the use of compiler directives, allowing developers to identify parallel sections of code without explicit message passing. This characteristic not only streamlines the development process but also facilitates debugging and testing, as existing tools and techniques for sequential code can be seamlessly applied. OpenMP's shared-memory model is particularly advantageous when the computational workload can be effectively parallelized within the memory and core limitations of a single machine. In scenarios where distributed memory or scalability across multiple machines is not a strict requirement, OpenMP proves to be an efficient and developer-friendly choice, contributing to a smoother and more accessible parallel programming experience.

# Reflection

MPI Implementation Reflection:

**Challenges:**

1. **Synchronization Complexity:** Achieving synchronization among parallel processes for image rotation demands a nuanced approach. Ensuring that image blocks are manipulated coherently without conflicts requires sophisticated coordination.
2. **Load Balancing Dynamics:** Efficiently distributing image data across MPI processes while maintaining a balanced workload is challenging, especially when confronted with varying computational loads due to diverse image characteristics and rotation degrees.
3. **Communication Overheads:** Managing MPI communication patterns becomes intricate, with considerations for minimizing communication overheads while optimizing data exchange among processes.

**Solutions:**

1. **Synchronization Strategies:** The MPI implementation effectively tackles synchronization complexities by employing meticulous strategies to ensure seamless collaboration among processes during image block rotation.
2. **Load Balancing Optimization:** While the existing load balancing mechanism is robust, exploring adaptive strategies tailored to specific image characteristics and rotation degrees could provide further optimizations for enhanced performance.
3. **Efficient MPI Communication:** MPI functions are judiciously utilized to facilitate efficient communication, emphasizing the seamless exchange of data among processes during parallel image rotation.

OpenMP Implementation Reflection:

**Challenges:**

1. **Data Dependency Management:** Parallelizing image rotation using OpenMP demands a meticulous approach to managing data dependencies to prevent interference within parallel regions. Coordinating threads to ensure data integrity is crucial.
2. **Race Conditions Mitigation:** The concurrent rotation of image blocks in parallel sections introduces the potential for race conditions. Preventing conflicts through careful thread synchronization becomes paramount for the reliability of the parallel algorithm.

**Solutions:**

1. **Leveraging OpenMP Sections:** The OpenMP implementation effectively utilizes sections to parallelize the rotation of distinct image blocks, showcasing an organized approach to concurrent operations.
2. **Synchronization Mechanisms:** To fortify against race conditions, additional synchronization mechanisms, such as critical sections or locks, are employed to ensure that threads collaborate seamlessly and maintain the integrity of the rotated image.

In both the MPI and OpenMP implementations, the delicate balance between achieving parallelism and managing complexities is evident. The MPI solution excels in distributed environments, emphasizing efficient communication and load balancing, while the OpenMP implementation navigates shared-memory architectures with precision in synchronization details. Further exploration of optimization avenues, adaptive load balancing, and reinforcement of synchronization mechanisms emerges as a continuous process for enhancing the robustness and scalability of both parallel solutions for image manipulation algorithms.