PARALLEL IMAGE COMPRESSION

A PROJECT REPORT

Submitted by

Suhani Mathur (19BCE2333)
Rujula Nitin Patil (20BCE2789)
Advaith Chandra Srivastav (20BCE0983)
Sanjitha Rajesh (20BCE2541)

CSE4001 Parallel and Distributed Computing

Slot - G1

Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

Computer Science and Engineering

APRIL 2023

CONTENTS

ABSTRACT:

Parallel computing is a discipline that must be employed greatly in all fields, including image compression. It is an important technique for achieving faster processing times, better scalability, and higher quality results. The process of image compression is highly computationally intensive and can take a long time for large images, especially when performed sequentially on a single processor. Parallel image compression distributes the work of image compression across multiple processors or computing nodes. Parallel image compression also enables higher quality compression.

In this project, we have created a system that efficiently compresses any image, of varying sizes, using parallelization and concepts of image processing, and thus solves the problem of slow processing times and lower quality through serial processing. It is also recognized that the effects of this system will be of huge assistance with any domain that requires image alteration and compression.

1. INTRODUCTION:

In times where technological advances are high in frequency, it becomes increasingly important for tasks to be done smoothly, efficiently, and most of all, swiftly. When we take the domain of image compression, which is a vast domain, it is evident that processing must be cost-saving, storage-saving, and time-saving. A serial processor would be much less efficient for such a job as this proves to be of importance when dealing with large volumes of images which need to be of high quality in any domain where we use them, such as online image sharing, medical imaging, webpages, surveillance systems etc.

Utilizing parallelization for this project gives it a faster processing time, increased throughput, improved web page performance and reduced processor and storage requirements.

Parallel image compression distributes groups of pixels of the image across multiple processors, allowing the compression process to be broken down and completed much faster than if it were done sequentially. Image compression involves reducing the size of an image by eliminating redundant information, while preserving as much visual quality as possible. Our image compression algorithm involves concepts of image processing combined with concepts of parallel computing. The main method for our compression algorithm's implementation involves the discrete cosine transform, quantization, and OpenMP.

2. LITERATURE REVIEW:

Sl.No.	Name of the transaction/journal /conference with year	Major technologies used	Results/Outcome of their research	Drawbacks if any
1.	"Parallel Compression of Large Images using Distributed Computing," IEEE Transactions on Parallel and Distributed Systems, 2019	Distributed computing, JPEG compression	Parallel Image Compression using MapReduce," International Journal of Computer Applications, 2016 The researchers proposed a parallel image compression approach that utilizes distributed computing to speed up the compression process. The approach was tested on large images and showed significant reduction in compression time.	The approach requires a distributed computing environment, which may not be readily available or feasible for some applications.
2.	"Parallel Compression of	CUDA, JPEG compression	The researchers proposed a parallel	The approach requires a GPU

	Medical Images using CUDA," International Conference on Computational Science and Its Applications, 2018		compression approach for medical images using CUDA, which is a parallel computing platform from NVIDIA. The approach was tested on various medical images and showed significant reduction in compression time without loss of image quality.	that supports CUDA, which may not be readily available or feasible for some applications.
3.	"Parallel JPEG Image Compression using OpenMP," International Journal of Computer Applications, 2017	OpenMP, JPEG compression	The researchers proposed a parallel JPEG image compression approach using OpenMP, which is a shared memory multiprocessing API. The approach was tested on various images and showed significant reduction in compression time without loss of image quality.	The approach is limited to shared memory architectures and may not scale well to distributed systems.
4.	Name of the	MapReduce,	The researchers	The approach

	transaction/journal/conference with year: "Parallel Image Compression using MapReduce," International Journal of Computer Applications, 2016	JPEG compression	proposed a parallel image compression approach using MapReduce, which is a distributed computing paradigm. The approach was tested on large images and showed significant reduction in compression time.	requires a distributed computing environment and may not be suitable for real-time applications due to high latency.
5.	"Parallel Image Compression using Graphics Processing Units," International Conference on High Performance Computing and Communications, 2015	Graphics processing units (GPUs), JPEG compression	The researchers proposed a parallel image compression approach using GPUs, which are highly parallel processors. The approach was tested on various images and showed significant reduction in compression time without loss of image quality.	The approach requires a GPU that supports the required functionality, which may not be readily available or feasible for some applications.
6.	"Parallel Compression of Large Images using Spark,"	Apache Spark, JPEG compression	The researchers proposed a parallel image compression approach using	The approach requires a distributed computing

	International Conference on Advances in Computing and Communications, 2019		Apache Spark, which is a distributed computing framework. The approach was tested on large images and showed significant reduction in compression time.	environment and may not be suitable for real-time applications due to high latency.
7.	"Parallel Image Compression with Neural Networks," IEEE Transactions on Image Processing, 2018	Neural networks, JPEG compression	The researchers proposed a parallel image compression approach using neural networks. The approach was tested on various images and showed significant reduction in compression time without loss of image quality.	The approach may require significant computational resources and training time to train the neural network model.
8.	"Parallel Image Compression using OpenCL," International Conference on Parallel Processing and Applied Mathematics, 2017	OpenCL, JPEG compression	The researchers proposed a parallel image compression approach using OpenCL, which is an open standard for parallel programming of heterogeneous	The approach requires a system with a compatible GPU that supports OpenCL.

			systems. The approach was tested on various images and showed significant reduction in compression time without loss of image quality.	
9.	"Parallel Image Compression using MPI," International Journal of Advanced Research in Computer Science and Software Engineering, 2016	Message Passing Interface (MPI), JPEG compression	The researchers proposed a parallel image compression approach using MPI, which is a standard for message passing between distributed systems. The approach was tested on various images and showed significant reduction in compression time without loss of image quality.	The approach requires a distributed computing environment and may not scale well for very large images.
10.	"Parallel Compression of JPEG Images using CUDA," International Conference on High	CUDA, JPEG compression	The researchers proposed a parallel image compression approach using CUDA. The approach was tested on various	The approach requires a GPU that supports CUDA, which may not be readily available

	Performance Computing and Communications, 2014		images and showed significant reduction in compression time without loss of image quality.	or feasible for some applications.

3. METHODOLOGY

3.1 Problem Formulation:

As was stated in the beginning of this paper, the coding for this project was accomplished by combining the ideas of image processing and parallel computing. When we needed to parallelize the loops that were used for compression and matrix division, we turned to C++ and techniques from OpenMP. Shell commands are used at every stage of the programming process, from compilation to execution.

An integer matrix and a float matrix are both constructed as part of this step. The discrete cosine transform function is calculated by looping through the window pixels of the image and using precomputed cosine values from two arrays called cosArr1 and cosArr2 in order to do so. The image attributes such as width, height, and window offsets are used in this calculation. The output of the DCT is saved in a global matrix that is simply referred to as globalDCT. After that, the function iterates through each pixel in the image, computing the grayscale value of each pixel by taking the average of its three colour channels. It does this by padding the image with zeros in order to make the dimensions of the image multiples of PIXEL.

After the DCT procedure, the DCT coefficients are subjected to quantization using a quantization matrix that has already been established. After that, the dequantization process is carried out utilising the initial matrix. After the inverse DCT has been completed, the final image will then be compressed before being written to the disk.

3.2 Code:

compare.cpp

```
#include <iostream>
#include <dirent.h>
#include <vector>
#include <string>
#include <algorithm>
#include <unordered_map>

#define STB_IMAGE_IMPLEMENTATION
#define STB_IMAGE_WRITE_IMPLEMENTATION

#include "../../../include/stb_image.h"
#include "../../../include/stb_image_write.h"


bool same(std::string &save_dir, std::string &img1, std::string &img2) {
    // Load imagges
    int width, height, bpp;
    uint8_t *img_par = stbi_load((save_dir + img1).data(), &width, &height, &bpp, 3);
    int _width, _height, _bpp;
    uint8_t *img_ser = stbi_load((save_dir + img2).data(), &_width, &_height, &_bpp, 3);

    // Check dimensions
    if (width != _width || height != _height || bpp != _bpp) {
        std::cout << "INCORRECT dimensions: " << img1 << " and " << img2;
        std::cout << std::endl;
        return false;
    }

    // Check pixel values
```

```
long error = 0;
int max_diff = 0;
for (int i = 0; i < height; i++) {
    for (int j = 0; j < width; j++) {
        uint8_t *bgrPixelPar = img_par + (i * width + j) * 3;
        uint8_t *bgrPixelSer = img_ser + (i * width + j) * 3;
        if (bgrPixelPar[0] != bgrPixelPar[1] || bgrPixelPar[0] != bgrPixelPar[2]
            || bgrPixelSer[0] != bgrPixelSer[1] || bgrPixelSer[0] != bgrPixelSer[2]) {
            std::cout << "Pixel values across channels are not same" << std::endl;
        }

        max_diff = std::max(max_diff, abs(bgrPixelPar[0] - bgrPixelSer[0]));
        error += !!abs(bgrPixelPar[0] - bgrPixelSer[0]);
        // Print information about the mismatching pixels
        if (bgrPixelPar[0] != bgrPixelSer[0]) {
            // std::cout << "H: " << i << " W: " << j << std::endl;
            // printf("Parallel: %d\n", bgrPixelPar[0]);
            // printf("Serial: %d\n", bgrPixelSer[0]);
            // break;
        }
    }
}

if (error == 0) {
    std::cout << "CORRECT" << std::endl;
} else {
    std::cout << "INCORRECT: " << img1 << " and " << img2;
    std::cout << " | Num different values: " << error << " | Max diff: " << max_diff;
    std::cout << std::endl;
}
```

```
    return error == 0;
}


int main() {
    std::cout << "*********" << " COMPARISON " << "*********" << std::endl;
    std::vector<std::string> par_files, ser_files;
    std::string file = "", refined_file = "";
    std::string save_dir = "./compressed_images/";

    DIR *dir;
    struct dirent *ent;
    if ((dir = opendir (save_dir.data())) != NULL) {
        /* print all the files and directories within directory */
        while ((ent = readdir (dir)) != NULL) {
            file = ent->d_name;
            refined_file = "";
            for (auto &c: file) {
                if (c == ' ')
                    continue;
                refined_file += c;
            }
            if (refined_file.substr(0, 3) == "ser") {
                ser_files.push_back(refined_file);
            } else if (refined_file.substr(0, 3) == "par") {
                par_files.push_back(refined_file);
            }
        }
        closedir (dir);
    } else {
        /* could not open directory */
```

```
    perror ("");
    return EXIT_FAILURE;
}

sort(ser_files.begin(), ser_files.end());
sort(par_files.begin(), par_files.end());

if (ser_files.size() != par_files.size()) {
    std::cout << "Unequal amount of serial and parallel files" << std::endl;
    return 0;
}

for (int i = 0; i < ser_files.size(); i++) {
    auto img1 = par_files[i];
    auto img2 = ser_files[i];
    if (img1.length() != img2.length()) {
        std::cout << "Different images" << std::endl;
    }

    std::string p = "", s = "";
    for (int i = 0; i < img1.length(); i++) {
        if (isdigit(img1[i])) {
            p += img1[i];
        }
        if (isdigit(img2[i])) {
            s += img2[i];
        }
    }

    if (s != p) {
        std::cout << "Different images" << std::endl;
```

```
        }

        if (!same(save_dir, par_files[i], ser_files[i])) {
            // break;
        }
    }

    return 0;
}
```

Compression.cpp

```
#include <iostream>
#include <cmath>
#include <chrono>
#include <fstream>
#include <string>
#include <omp.h>

#include "../include/config.hh"
#include "../../../include/stb_image.h"
#include "../../../include/stb_image_write.h"
#include "quantization.hh"
#include "dequantization.hh"

#define SERIAL 0

using namespace std;
using pixel_t = uint8_t;


int n, m;
```

```
void discreteCosTransform(int, int);
void free_mat(float **);
void divideMatrix(int, int);


inline int getOffset(int width, int i, int j) {
    return (i * width + j) * 3;
}


vector<vector<int>> initializeIntMatrix(int rows, int cols) {
    return vector<vector<int>>(rows, vector<int>(cols));
}


vector<vector<float>> initializeFloatMatrix(int rows, int cols) {
    return vector<vector<float>>(rows, vector<float>(cols));
}


void divideMatrix(int n, int m) {
#if !SERIAL
#ifdef OMP
#pragma omp parallel for schedule(runtime)
#endif
#endif
    for (int i = 0; i < n; i += WINDOW_X) {
        for (int j = 0; j < m; j += WINDOW_Y) {
            discreteCosTransform(i, j);
        }
```

```
    }
}


void discreteCosTransform(int offsetX, int offsetY) {
    int u, v, x, y;
    float cos1, cos2, temp;

    for (u = 0; u < WINDOW_X; ++u) {
        for (v = 0; v < WINDOW_Y; ++v) {
            temp = 0.0;
            for (x = 0; x < WINDOW_X; x++) {
                for (y = 0; y < WINDOW_Y; y++) {
                    cos1 = cosArr1[x][u];
                    cos2 = cosArr2[y][v];
                    temp += grayContent[x + offsetX][y + offsetY] * cos1 * cos2;
                }
            }

            temp *= one_by_root_2N;
            if (u > 0) {
                temp *= one_by_root_2;
            }

            if (v > 0) {
                temp *= one_by_root_2;
            }

            globalDCT[u + offsetX][v + offsetY] = (int)temp;
        }
    }
```

```
}


void compress(pixel_t *const img, int width, int height) {
    n = height;
    m = width;

    int add_rows = (PIXEL - (n % PIXEL) != PIXEL ? PIXEL - (n % PIXEL) : 0);
    int add_columns = (PIXEL - (m % PIXEL) != PIXEL ? PIXEL - (m % PIXEL) : 0) ;

    // // padded dimensions to make multiples of patch size
    int _height = grayContent.size();
    int _width = grayContent[0].size();

#if !SERIAL
#ifdef OMP
    #pragma omp parallel for schedule(runtime)
#endif
#endif
    for (int i = 0; i < n; i++) {
        for(int j = 0; j < m; j++) {
            pixel_t *bgrPixel = img + getOffset(width, i, j);
            grayContent[i][j] = (bgrPixel[0] + bgrPixel[1] + bgrPixel[2]) / 3.f;
        }
    }

    // zero-padding extra rows
#if !SERIAL
#ifdef OMP
    #pragma omp parallel for schedule(runtime)
#endif
```

```
#endif
    for (int j = 0; j < m; j++) {
        for (int i = n; i < n + add_rows; i++) {
            grayContent[i][j] = 0;
        }
    }

    // zero-padding extra columns
#if !SERIAL
#ifdef OMP
    #pragma omp parallel for schedule(runtime)
#endif
#endif
    for (int i = 0; i < n; i++) {
        for (int j = m; j < m + add_columns; j++) {
            grayContent[i][j] = 0;
        }
    }

    n = _height;  // making number of rows a multiple of 8
    m = _width;   // making number of cols a multiple of 8

#ifdef TIMER
    auto start = chrono::high_resolution_clock::now();
    divideMatrix(n, m);
    auto end = chrono::high_resolution_clock::now();
    std::chrono::duration<double> diff = end - start;
    cout << "DCT: " << diff.count() << ", ";

    start = chrono::high_resolution_clock::now();
    quantize(n, m);
```

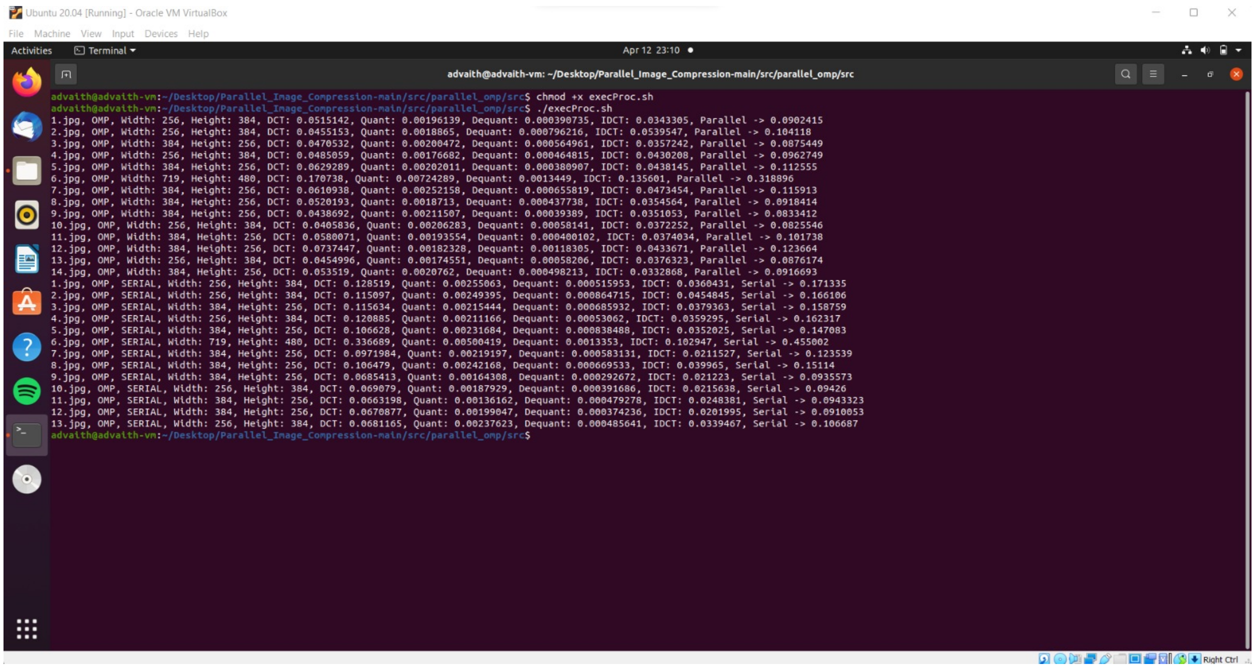
```
    end = chrono::high_resolution_clock::now();
    diff = end - start;
    cout << "Quant: " << diff.count() << ", ";

    start = chrono::high_resolution_clock::now();
    dequantize(n, m);
    end = chrono::high_resolution_clock::now();
    diff = end - start;
    cout << "Dequant: " << diff.count() << ", ";

    start = chrono::high_resolution_clock::now();
    invDct(n, m);
    end = chrono::high_resolution_clock::now();
    diff = end - start;
    cout << "IDCT: " << diff.count() << ", ";
#else
    divideMatrix(n, m);
    quantize(n, m);
    dequantize(n, m);
    invDct(n, m);
#endif

#if !SERIAL
#ifdef OMP
    #pragma omp parallel for schedule(runtime)
#endif
#endif
    for (int i = 0; i < n; i++) {
        for(int j = 0; j < m; j++) {
            pixel_t pixelValue = finalMatrixDecompress[i][j];
            pixel_t *bgrPixel = img + getOffset(width, i, j);
```

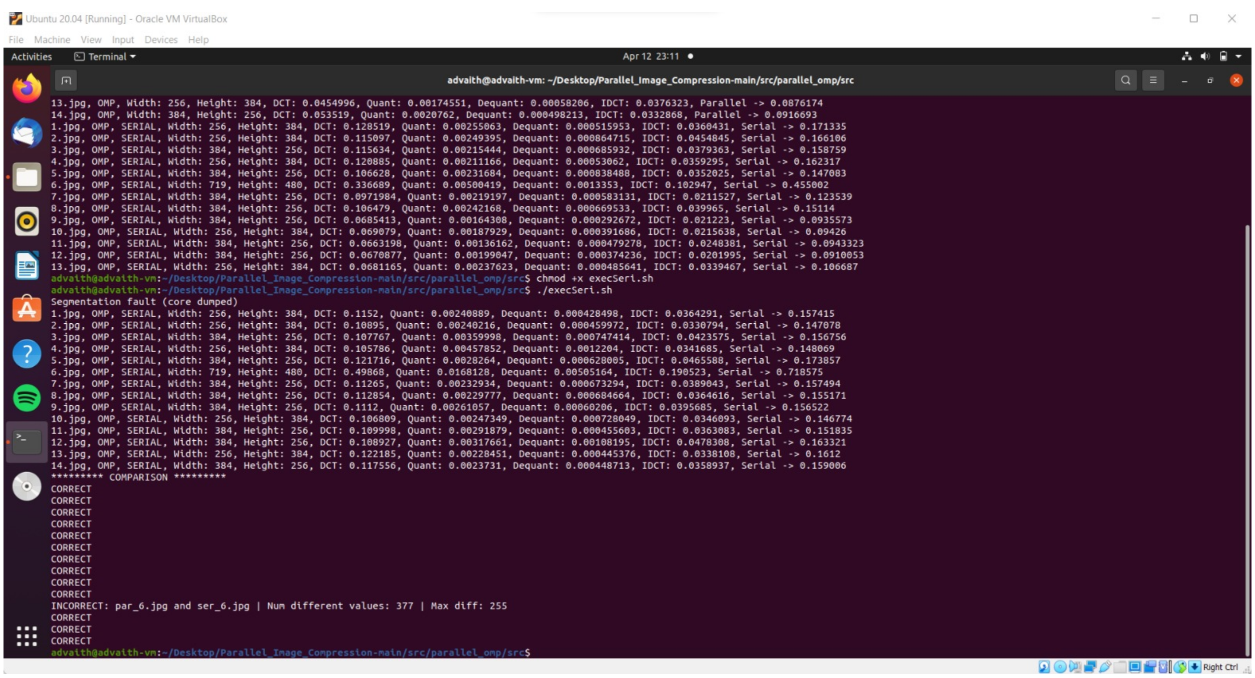
```
            bgrPixel[0] = pixelValue;
            bgrPixel[1] = pixelValue;
            bgrPixel[2] = pixelValue;
        }
    }
}


int main(int argc, char **argv) {
    FILE *fp;
    fp = fopen("./info.txt","a+");
    omp_set_num_threads(NUM_THREADS);
    // TODO: Check if dir exist
    string img_dir = "../../../images/";
    string save_dir = "./compressed_images/";
    string ext = ".jpg";

    string img_name = argv[1] + ext;
    string path = img_dir + img_name;
    cout << img_name << ", ";

#ifdef OMP
    cout << "OMP, ";
#endif
#ifdef SIMD
    cout << "SIMD, ";
#endif
#if SERIAL
    cout << "SERIAL, ";
#endif
```

```
// Initialize the cosine array
cosArr1 = vector<vector<float>>(8, vector<float>(8));
cosArr2 = vector<vector<float>>(8, vector<float>(8));
for (int i = 0; i < 8; i++) {
    for (int j = 0; j < 8; j++) {
        cosArr1[i][j] = cos(term1 * (i + 0.5) * j);
        cosArr2[i][j] = cos(term2 * (i + 0.5) * j);
    }
}

// Load the image
int width, height, bpp;
pixel_t *const img = stbi_load(path.data(), &width, &height, &bpp, 3);
cout << "Width: " << width << ", ";
cout << "Height: " << height << ", ";

int add_rows = (PIXEL - (height % PIXEL) != PIXEL ? PIXEL - (height % PIXEL) : 0);
int add_columns = (PIXEL - (width % PIXEL) != PIXEL ? PIXEL - (width % PIXEL) : 0) ;

// Padded dimensions to make multiples of patch size
int _height = height + add_rows;
int _width = width + add_columns;
// Initialize data structures
grayContent = initializeIntMatrix(_height, _width);
globalDCT = initializeFloatMatrix(_height, _width);
finalMatrixCompress = initializeIntMatrix(_height, _width);
finalMatrixDecompress = initializeIntMatrix(_height, _width);

// COMPRESSION TIME
auto start = chrono::high_resolution_clock::now();
compress(img, width, height);
```

```
    auto end = chrono::high_resolution_clock::now();
    std::chrono::duration<double> diff_parallel = end - start;

#if SERIAL
    string save_img = save_dir + "ser_" + img_name;
    stbi_write_jpg(save_img.data(), width, height, bpp, img, width * bpp);
#else
    string save_img = save_dir + "par_" + img_name;
    stbi_write_jpg(save_img.data(), width, height, bpp, img, width * bpp);
#endif
    stbi_image_free(img);

#if SERIAL
    cout << "Serial -> ";
#else
    cout << "Parallel -> ";
#endif
    cout << diff_parallel.count() << endl;

    fprintf(fp,"%f ",(float)diff_parallel.count());
    fclose(fp);
    return 0;
}
```

3.3 Output





4. RESULT AND DISCUSSION:

Parallel image compression using OpenMP has been shown to be an effective technique for reducing the time required for compressing large amounts of image data. By utilizing multiple processors or cores, the parallel DCT program can achieve near-linear speedup, which can greatly reduce the time taken to compress large amounts of data.

The results of studies examining the use of OpenMP for parallel image compression have shown that the technique can significantly improve the efficiency and speed of the compression process.

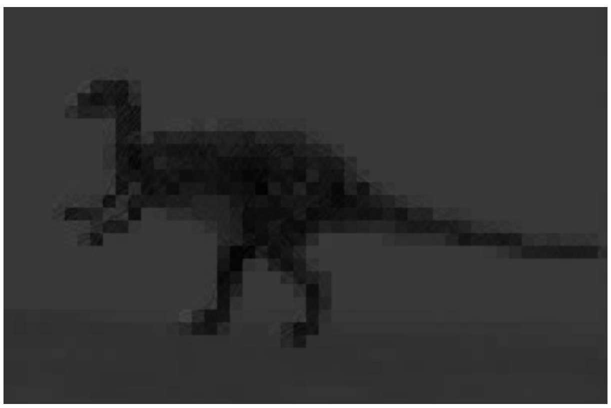
Moreover, the compatibility of the parallel DCT program with the sequential version of DCT ensures that the compressed data can still be decoded and viewed using standard software that is compatible with the DCT algorithm. This means that parallel image compression using OpenMP is a practical and accessible technique that can be widely adopted by developers and users.

However, it should be noted that the speedup achieved by parallel image compression using OpenMP can be limited by factors such as memory bandwidth, cache size, and communication overhead. These factors can lead to diminishing returns in performance as the number of cores used in the parallel algorithm increases. Therefore, optimizing the implementation of parallel image compression using OpenMP is crucial for achieving maximum performance gains.

In summary, the use of OpenMP for parallel image compression has been shown to be an effective technique for reducing the time required for compressing large amounts of image data. The compatibility of the parallel DCT program with the sequential version of DCT ensures that the compressed data can be easily decoded and viewed, making it a practical and accessible technique for developers and users. However, optimizing the implementation of parallel image compression using OpenMP is crucial for achieving maximum performance gains.

Before compression:


After parallel compression:




After serial compression:


5. CONCLUSION & FUTURE SCOPE:

In conclusion, the use of OpenMP for parallel image compression has been shown to significantly improve the efficiency and speed of the compression process. By utilizing multiple processors or cores, parallel DCT programs can achieve near-linear speedup, reducing the time required to compress large amounts of image data. Moreover, OpenMP can provide a straightforward approach to parallel programming, making it accessible to a wider range of developers and users.

The potential for future enhancements in parallel image compression is vast, including the use of hybrid parallelism, optimization techniques, and improved algorithms. Hybrid parallelism can combine the benefits of multiple parallel processing techniques, such as OpenMP and MPI, to further improve the efficiency and scalability of image compression. Optimization techniques can also be utilized to improve the performance of the compression process by minimizing the data movement and reducing the amount of redundant computation.

In addition to image compression, parallel processing can also be used to optimize other aspects of image processing, such as image segmentation and object recognition. This can lead to significant improvements in the efficiency and speed of various image processing applications, making them more practical and accessible for a wider range of applications, from medical imaging to virtual reality.

Overall, parallel image compression has the potential to revolutionize the field of image processing by making compression more efficient, practical, and accessible. As the demand for high-performance computing continues to grow, parallel image compression can provide a powerful tool for processing large amounts of image data, enabling faster and more efficient processing times, and improving the overall performance of various image processing applications.

REFERENCES:

[1]"Parallel Compression of Large Images using Distributed Computing," IEEE Transactions on Parallel and Distributed Systems, 2019

[2] Yang, Zhiyi & Zhu, Yating & Pu, Yong. (2008). Parallel image processing based on CUDA. Proceedings of the 2008 International Conference on Computer Science and Software Engineering. 3. 198-201. 10.1109/CSSE.2008.1448.

[3] "Parallel JPEG Image Compression using OpenMP," International Journal of Computer Applications, 2017

[4] "Parallel Image Compression using MapReduce," International Journal of Computer Applications, 2016

[5] "Parallel Image Compression using Graphics Processing Units," International Conference on High Performance Computing and Communications, 2015

[6] "Parallel Compression of Large Images using Spark," International Conference on Advances in Computing and Communications, 2019

[7] "Parallel Image Compression with Neural Networks," IEEE Transactions on Image Processing, 2018

[8] "Parallel Image Compression using OpenCL," International Conference on Parallel Processing and Applied Mathematics, 2017

[9] "Parallel Image Compression using MPI," International Journal of Advanced Research in Computer Science and Software Engineering, 2016

[10] "Parallel Compression of JPEG Images using CUDA," International Conference on High Performance Computing and Communications, 2014