PARALLEL COMPUTER ARCHITECTURE AND PROGRAMMING
15-618

Parallel Image Compression using the JPEG Algorithm

1 URL

`https://spenceryu.github.io/15618-final-project/`

2 Summary

We implemented the JPEG compression algorithm using OpenMPI and OpenMP and compared the performance of the two implementations against our sequential version. We also explored implementing the algorithm using CUDA but ultimately decided against completing this implementation due to a variety of reasons.

3 Background

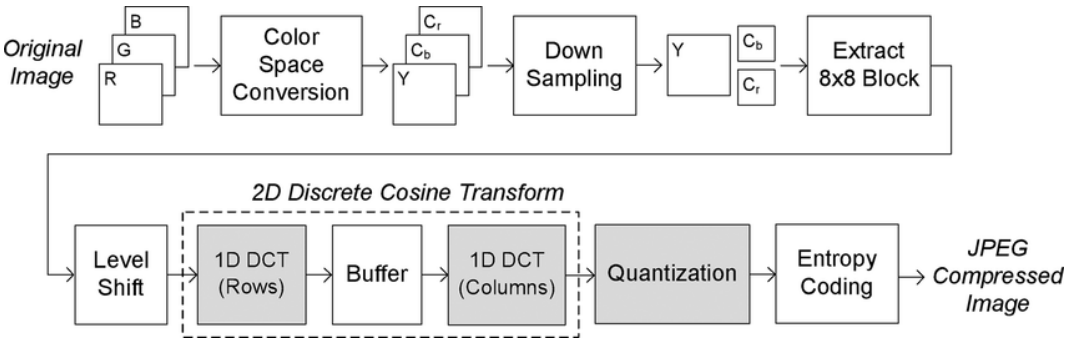


Figure: JPEG Compression Pipeline

The JPEG algorithm takes in an image and compresses it using a lossy compression algorithm. The algorithm follows a multi step pipeline. First, pixels are converted from the RGB color space to the YCbCr color space. This is important because the YCbCr color space contains most of the information that is important for our vision in the Y channel, so the Cb and Cr channels can be downsampled to save space. The YCbCr color space image is then converted into 8x8 macroblocks, on which several signal processing algorithms are applied. The Discrete Cosine Transform of each 8x8 block is taken. Discrete Cosine Transforms are often used in image processing for compression because they can store most of the visually important information from an image in just a few coefficients. Quantization using a quantization matrix is then used on the AC components (those not at the [0,0] location

of each block), in order to reduce the level of information that is preserved while storing most of the relevant color and spatial information. Quantization is effective at reducing the information needed to be stored when used in conjunction with the DCT, because most of the unimportant AC components of the signal will be reduced to 0. Differential Pulse Code Modulation is then used to reduce the information stored in the DC component (the component at the [0,0] location of each block) by storing the first element and its deltas. Run length (Huffman) encoding is used to compress the AC components of the block into tuples of encoded values and the number of times they occur. IO takes place when reading from raw images and writing to compressed images.

We created several key data structures to store relevant information. PixelRgba and PixelYcbcr structs were used to store the pixels of the image. ImageRgb, ImageYcbcr, and ImageBlocks structs were used to store the width, height, and pixels/blocks in the image. To store the Huffman compressed structures, we used RleTuple structs to store pairs of (encoded value, number of occurrences). A vector of RleTuple values is present in each EncodedBlockColor, which also stores the encode and decode tables as well as the DC value. Three EncodedBlockColor structs are stored in the EncodedBlock struct for the different channels, which represents a compressed block of the image. Mirror structures were created for EncodedBlockColor and EncodedBlock that did not use pointers to facilitate message passing and ease of memory arithmetic in OpenMPI structs.

Many steps in our process are computationally expensive, but the most computationally expensive parts of our pipeline are run length encoding and conversion from the RGB color space image to the YCbCr color space image, as well as from the YCbCr color space image to blocks. These steps benefit from parallelization as there are different levels of dependencies between each of the components of the image. The algorithm is applied through individual blocks which can be sent to different threads without interference, but there are steps where the order of the block processing matters. In addition, steps such as the conversion into blocks require the entire image to be completed properly. Other than this sequential step, the rest of the algorithm can be run in parallel.

Each step of the process described above is dependent on the previous step so the algorithm inherently has a lot of dependencies. The block decomposition cannot occur until the color space conversion (an expensive operation) is finished for all pixels of an image. Consider a 1024x768 image - small by today's standards - has 786432 pixels, each with separate color channels. This is a very good candidate for parallelism due to the computationally bound result as well as the large dataset size. Block decomposition is also expensive because all pixels have to be reorganized into blocks based on arithmetic (again, for an incredibly large number of pixels). This also makes it a good target for parallelism. Once block decomposition is complete, all other steps are only dependent on pixels in the same block. The next expensive

step is run length encoding - frequency tables have to be calculated and encoding has to be done using these frequency tables. This is also a good target for parallelism because run length encoding is independent for every block. Additionally, it is amenable to SIMD execution, as each block is 8x8 and would fit into 64 wide SIMD. Even with smaller SIMD vector sizes, computation potentially would still be sped up significantly.

4 Approach

Technologies Used

For our project, we used the gates machines (ghc). We used C++ for the sequential implementation of our program, and used OpenMPI and OpenMP for our parallel implementations. We were originally going to use CUDA but decided against it after beginning the implementation. The reasoning for this will be described later in the report.

Mapping the Problem

We had to change the original sequential algorithm to account for the OpenMPI version of our code. For the OpenMPI version of the algorithm we heavily used our own implementations of scatters, gathers, serializing, and deserializing for passing our custom data structures. We had to create data structures that did not use pointers, and serialize/deserialize all of our encoded blocks and pixels using those. It was also important to create these custom structures because it simplified the memory arithmetic immensely when allocating the buffers for our blocking send/recv calls. For the OpenMP version of our program, parallelization was far simpler and did not require serious rewriting and redefinition of data structures. Instead, we just had to reduce our reliance on some of the nicer C++ abstractions (such as range-based for loops). Our OpenMPI implementation involved mapping sets of blocks to threads, while OpenMP mapped iterations of for loops to threads, but most of the loops iterated over blocks so both approaches assigned threads in a similar manner.

In the OpenMPI implementation of the program, message passing is the only form of communication between processes running on individual threads. We found this to have lots of overhead related to serialization of our data structures, creating massive amounts of overhead. In the OpenMP version of our program, a shared memory implementation is used in order to achieve parallelism. We found this to be more effective than OpenMPI due to less overhead (this will be described in more detail in subsequent sections).

Arriving at the Solution

Arriving at a parallel solution took a very long time for us - far longer than we had anticipated. We had difficulties getting a parallel version of our code to work, and it took even longer to get a speedup over the sequential version of our program.

Initially, we thought that we would just be able to use CUDA, map a CUDA thread block to a set of macroblocks, and perform all of the computation on the GPU. This proved to be far too idealistic, as we did not consider how CUDA does not have support for C++ STL abstractions such as maps. This would require such a substantial rewrite of the program (the entire sequential version would have to be scrapped because of the data structures used) that we decided it would be worth exploring other avenues of parallelization. Additionally, from a performance-oriented standpoint, we realized that the amount of added parallelism from such a massive number of threads would still likely be overshadowed by the overhead from such massive amounts of memory affected by cudaMalloc() calls as well as the overhead from copying between the host and the device. We realized we would be unlikely to see a substantial performance boost and decided to instead explore an OpenMPI based approach to parallelization.

OpenMPI turned out to be far more fruitful of an approach and did not require a full rewrite of our program, due to its support for C++ STL functions. However, the main drawback of using OpenMPI was the massive amount of overhead needed for message serialization, deserialization, and the creation of related data structures that did not contain pointers instead of values. Additionally, some strange errors came out of the fact that we were allocating so much space; the program would segfault unless we used allocate memory for all of our buffers. When we analyzed the completed implementation, we found that for small images, OpenMPI provided a substantial performance boost. However, we found that for large images, our performance was even worse than that of the sequential version of our program. Even though we had a huge improvement in the speed of each step that was slow in the serial implementation (such as RLE), the amount of overhead when communicating (due to serialization/deserialization being ridiculously inefficient) caused the overall performance of this parallel implementation to scale rather poorly to larger images. We then decided that we should take a look at OpenMP to see if the flexibility of the shared memory model would yield better performance.

OpenMP was far simpler to use than either of the two parallel frameworks we had explored. We discovered that OpenMP did not support some of the nice C++ abstractions, which necessitated some rewriting to ensure that both OpenMPI and OpenMP would run correctly with our pipeline. We also had problems getting our Makefile to set up properly because of how we were using multiple compilers (mpic++ and g++) as well as the confusing process of using objects compiled from both binaries. After we got these problems sorted out, we found that the OpenMP implementation performed better than our OpenMPI implementation. OpenMPI had higher speedup for individual steps in the process, but for larger images, the communication overhead far overshadowed the speedup provided in each step, resulting in higher overall speedup for OpenMP. This is because OpenMP uses a shared memory

model that has low communication overhead between various threads. By comparison, having overhead to serialize and deserialize millions of pixels for OpenMPI incurs a huge unavoidable performance penalty.

Existing code

We used the lodepng image decoder (https://lodev.org/lodepng/) in order to read PNG images into our program and write PNG images to verify the correctness of our compression algorithm.

5 Results

Our project was successful at demonstrating how various implementations of parallelization for the sequential JPEG algorithm exhibit different behaviors at varying image sizes.

Measurements

We measured performance of our algorithm using speedup over the sequential implementation with time from CycleTimer::currentSeconds(). We measured the time each step of the algorithm takes as well as the overall time for the compression algorithm to run. This is a good way of measuring the overall performance of our algorithm, since the performance of a compression algorithm should be judged on its total run time rather than the active amount of computation time. By comparison, the CPU time would not have been a good measure of performance because a lot of what limits the performance of image processing on large amounts of data is that the computation is IO bound.

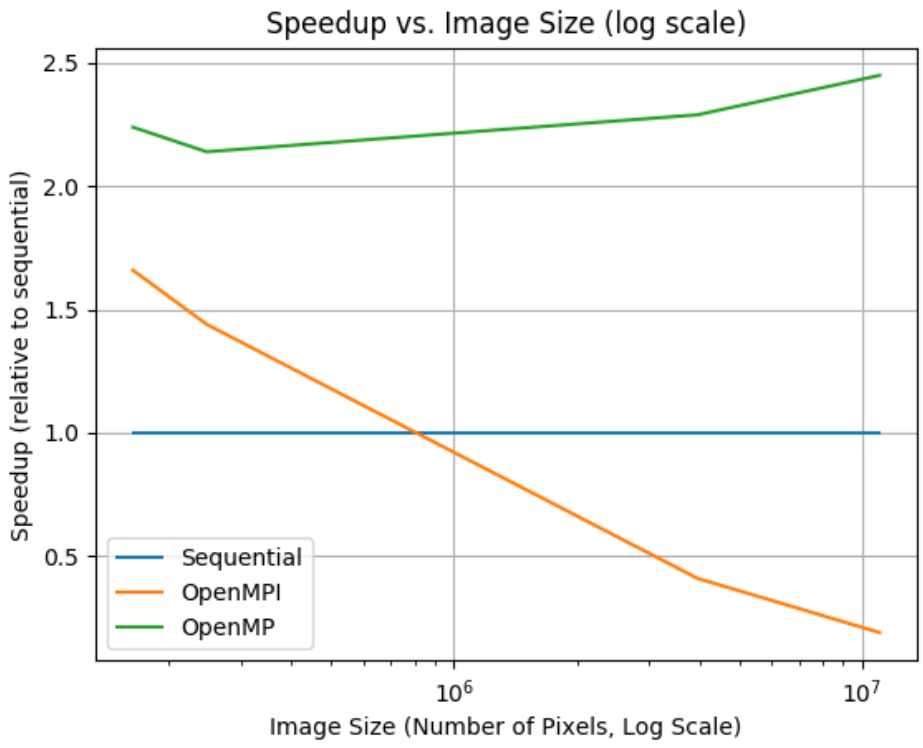
Experimental Setup

Requests to our program were generated using a bash script that invokes the correct binary parameters for our given sequential or parallel implementation. We used several images of different resolutions and color types to test the run time performance of our algorithm. The MATLAB cameraman reference image that we used is grayscale and is 495x500. The resolution chart image from Wikipedia that we used is 4320 x 2560 and in color, and has many long thin lines and bold colors over a mostly gray background. The MATLAB peppers reference image that we used is in color, has many sharp differences in color, and is 508x381. The cookie image that we used (2796 x 1414) is from Bon Appetit's chocolate chip cookie recipe, and is a cookie on a white stone countertop. All of these images vary significantly in their properties and we used them to explore how the image we are compressing is affected (if at all) by the properties of the image.

Experimental Configurations

Our three implementations of the JPEG algorithm all run on the ghc cluster machines (ghc). We chose not to use the latedays cluster because the latedays cluster performs message passing between multiple machines, something that is generally not done as part of the use case for image compression. Our baseline sequential code is a single threaded CPU implementation of the JPEG algorithm using C++. It heavily uses C++ constructs to optimize the performance, as it relies on features like smart pointers and C++ STL map. Our code that was parallelized using OpenMPI uses message passing as its source of parallelism. It is also implemented using C++, and is a parallel CPU implementation of the algorithm. Our parallelized version that uses OpenMP uses shared memory as its source of parallelism, and is also implemented using C++. All three versions of our implementation use the same functions for the actual processing of the image and its blocks.

Speedup Graphs



Problem Size

We found that for small images (around 500x500, or 250k pixels) there was still a noticeable improvement in performance when using both MPI and OMP compared to our sequential baseline. As the image size grows, the overhead caused by the MPI implementation's gather encoded blocks (plus related serialization overhead) penalizes performance far more heavily

than the gains made by parallelizing the algorithm in this fashion. However, the OpenMP shared memory implementation has a slightly higher speedup as the image size increases. The sequential program inherently scales poorly with increasing image size as the algorithm is computation bound by serial computation.

Performance: peppers.png (508x381)

Step	**Sequential**	**MPI**	**OMP**
Load Image	0.023s	0.018s	0.020s
Setup MPI	N/A	0.000s	N/A
Convert Bytes to Image	0.041s	0.006s	0.023s
Convert RGB to YCbCr	0.038s	0.005s	0.020s
Gather YCbCr Pixels	N/A	0.030s	N/A
Convert YCbCr to Blocks	0.040s	0.075s	0.032s
DCT	0.006s	0.001s	0.003s
Quantize/DPCM	0.014s	0.002s	0.011s
RLE	0.169s	0.023s	0.040s
Gather Encoded Blocks	N/A	0.042s	N/A
Encode Compressed Image	0.003s	0.001s	0.001s
Total Time	0.336s	0.202s	0.150s
Speedup	**1x**	**1.66x**	**2.24x**

Performance: cameraman.png (495x500)

Step	**Sequential**	**MPI**	**OMP**
Load Image	0.020s	0.020s	0.019s
Setup MPI	N/A	0.000s	N/A
Convert Bytes to Image	0.047s	0.008s	0.032s
Convert RGB to YCbCr	0.044s	0.007s	0.027s
Gather YCbCr Pixels	N/A	0.046s	N/A
Convert YCbCr to Blocks	0.047s	0.094s	0.043s
DCT	0.007s	0.002s	0.004s
Quantize/DPCM	0.016s	0.002s	0.010s
RLE	0.208s	0.028s	0.047s
Gather Encoded Blocks	N/A	0.063s	N/A
Encode Compressed Image	0.002s	0.002s	0.002s
Total Time	0.394s	0.273s	0.184s
Speedup	**1x**	**1.44x**	**2.14x**

Performance: cookie.png (2796x1414)

Step	**Sequential**	**MPI**	**OMP**
Load Image	0.198s	0.166s	0.181s
Setup MPI	N/A	0.000s	N/A
Convert Bytes to Image	0.411s	0.075s	0.306s
Convert RGB to YCbCr	0.459s	0.066s	0.269s
Gather YCbCr Pixels	N/A	0.475s	N/A
Convert YCbCr to Blocks	0.534s	1.220s	0.485s
DCT	0.093s	0.029s	0.064s
Quantize/DPCM	0.211s	0.044s	0.154s
RLE	2.970s	0.432s	0.665s
Gather Encoded Blocks	N/A	9.283s	N/A
Encode Compressed Image	0.005s	0.010s	0.005s
Total Time	4.873s	11.801s	2.131s
Speedup	**1x**	**0.41x**	**2.29x**

Performance: reschart.png (4320x2560)

Step	**Sequential**	**MPI**	**OMP**
Load Image	0.291s	0.261s	0.197s
Setup MPI	N/A	0.000s	N/A
Convert Bytes to Image	1.162s	0.236s	0.771s
Convert RGB to YCbCr	1.242s	0.226s	0.742s
Gather YCbCr Pixels	N/A	1.243s	N/A
Convert YCbCr to Blocks	1.516s	3.431s	1.262s
DCT	0.269s	0.085s	0.151s
Quantize/DPCM	0.605s	0.126s	0.395s
RLE	7.508s	1.099s	1.597s
Gather Encoded Blocks	N/A	59.956s	N/A
Encode Compressed Image	0.011s	0.010s	0.021s
Total Time	12.612s	66.674s	5.147s
Speedup	**1x**	**0.19x**	**2.45x**

Analysis of Speedup

For a breakdown of performance by execution time relative to the total runtime of the algorithm, see the above section (Problem Size). Under both parallelized versions of the algorithm, speedup was limited by the fact that each step of the algorithm has to be completed before the next can be started. This dependency chain inherently limits the amount of parallelization that can occur in our program. However, because all computations after

a certain point are done on 8x8 macroblocks, the performance penalty from these data dependencies is somewhat mitigated.

Under the OpenMPI implementation, we were limited by the serialization/gather/deserialization related to encoded blocks. This is because encoded blocks under run length/Huffman encoding do not naturally lend themselves to serialization through message passing. The encoded blocks rely heavily on pointers to existing data and use of STL map to make efficient encoding and decoding tables. In order to pass these data structures between threads, all of the values need to be copied into arrays and other derived MPI datatypes. For small images, performance with OpenMPI was adequate as the overhead associated with RLE was not that large, and therefore the gather process was rather quick (overhead associated with peppers.png in the MPI implementation only took about 24% of the time that the serial algorithm took to perform RLE, and the actual RLE operation only took 13% of the time that the serial algorithm took to perform RLE). Overall performance using MPI was around 40% better for the peppers.png image, while performance was about 31% better for the camerman.png image. As the image size increased, we saw a massive drop in performance relative to the sequential version of our code. The cookie.png image took 240% as long to run on the MPI implementation compared to the sequential implementation, with most of the slowdown (190%) due to the amount of time gathering and serializing/deserializing the encoded blocks took. The serialize/gather/deserialize, which took 59.956s for our large image (reschart) far outstrips any performance gains possible by optimizing the structure of the message passing protocol. Considering that the sequential algorithm runs on reschart in 12.612s, that singular gather with associated overhead takes 475% of the time that the full sequential takes to run. Even if we were to remove all pointers in our data structures and stack allocate everything (to attempt minimizing serialization overhead), then we would just segfault as the stack runs out of memory (a problem we ran into when allocating buffers for a subset of the blocks in our gather calls). This unavoidable slow sequential section of the OpenMPI implementation forced us to find an alternative approach.

Under the OpenMP implementation, computational performance scaled well with the size of the image. We found that the best pragma to use was a `parallel for`. Dynamic scheduling didn't provide an advantage because block sizes were fixed, every block took a similar amount of time to pass through each step of the pipeline. We suspect we were limited by the inherent serial nature of each step of the JPEG algorithm. Under all of the images we tested, we had over a 2x speedup for OMP compared to the sequential implementation, with the highest levels of speedup actually coming in the largest image (a 2.45x speedup for reschart.png). Individual steps of the process did not perform as well under OpenMP as they did with OpenMPI. One such example is the RLE step of the compression process for reschart. The OpenMP implementation took 45% longer than OpenMPI did to complete this step. However, due to the shared memory nature of OpenMP, it did not need the 60 second

gather immediately following RLE for OpenMPI. Similarly, OpenMP is outperformed in almost every other step of the pipeline, but its communication overhead advantage becomes increasingly obvious as image size increases. Future focus should be given to long steps like RLE where OpenMPI shows that greater speedup can be achieved, or even just more cutting edge compression algorithms. Given the trend of higher and higher image resolution, the OpenMP implementation is the clear best choice for JPEG compression as it scales better to larger images.

Target Machine

Our choice of target machine was a good choice - we initially were going to use GPU (CUDA) parallelism but decided against it for several reasons, one of which was that the theoretical performance gains we would achieve from such an implementation are quite minimal due to large amounts of overhead related to 1) having to create our own (unoptimized) version of C++ STL data structures, and 2) memory copying for millions of pixels between the device and host incurs a huge amount of overhead. We would have had to completely reimplement data structures to work in CUDA as well, which is not as interesting as actually seeing the difference in performance we get as a a result. Focusing instead on various CPU based parallelizations of the JPEG algorithm yielded interesting insights and avoided the need to rewrite our code entirely from scratch when parallelizing it.

6 References

https://en.wikipedia.org/wiki/JPEG
https://mpeg.chiariglione.org/standards/mpeg-1
https://www.researchgate.net/figure/Block-diagram-of-MPEG-1-algorithm-encoder_fig8_2985381 https://www.whydomath.org/node/wavlets/basicjpg.html
https://www.eetimes.com/an-overview-of-video-compression-algorithms/#
https://www.researchgate.net/profile/Deborah_Estrin2/publication/224440999/figure/fig1/AS:393669950623760@1470869638885/JPEG-compression-algorithm.png
https://lodev.org/lodepng/

7 Distribution of Work

The work was distributed evenly between both of us as we both worked on system design, implementation of functions, integration of components, debugging, and analysis.