PROJECT REPORT

Parallel Image Compression Algorithms

Members:

K213309 – Mohammad Yehya

K213433 – Daniyal Haider

K213388 – Mahad Munir

Introduction

This project delves into the domain of Parallel Image Compression, exploring the fusion of image compression algorithms with parallel computing paradigms. The primary goal is to achieve accelerated image compression and decompression processes, thereby enhancing overall system performance.

Throughout the report, we will investigate various parallelization strategies and their impact on compression efficiency. Moreover, we will assess the trade-offs between computational complexity, memory requirements, and image quality, providing a comprehensive analysis of the proposed parallel image compression framework.

The computer architecture used for this demonstration is as follows:

Number of cores	4 (max 4)
Number of threads	8 (max 8)
Manufacturer	GenuineIntel
Name	Intel Core i7 8550U
Codename	Kaby Lake-R
Specification	Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
Package (platform ID)	Socket 1356 FCBGA (0x7)
Technology	14 nm
Core Speed	3692.8 MHz
Multiplier x Bus Speed	37.0 x 99.8 MHz
Base frequency (cores)	99.8 MHz
Base frequency (mem.)	99.8 MHz
Stock frequency	2000 MHz
Max frequency	4000 MHz
Instructions sets	MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX, AVX2, FMA3
L1 Data cache	4 x 32 KB (8-way, 64-byte line)
L1 Instruction cache	4 x 32 KB (8-way, 64-byte line)
L2 cache	4 x 256 KB (4-way, 64-byte line)
L3 cache	8 MB (16-way, 64-byte line)
Bus Specification	PCI-Express 3.0 (8.0 GT/s)
Graphic Interface	PCI-Express
Memory Type	DDR4
Memory Size	8 GBytes
Channels	Single
Memory Frequency	1197.1 MHz (3:36)

Display adapter	
ID	0x1080108
Name	NVIDIA GeForce 930MX
Board Manufacturer	Hewlett-Packard
Revision	A2
Codename	GM108
Core family	0x118 (GM108-A)
Cores	384
ROP Units	8
Technology	28 nm
Memory type	DDR3 (Hynix)
Memory size	2 GB
Memory bus width	64 bits
Current Link Width	x4
Current Link Speed	2.5 GT/s
Windows Version	Microsoft Windows 10 (10.0) Professional 64-bit (Build 19045)
DirectX Version	12.0

Architecture Details Generated form CPUID-CPU Z.

Problem

Despite significant advancements in image compression techniques, the escalating demand for high-quality visual content in various domains poses formidable challenges to current systems. Therefore the need for efficiency arises and one of the solutions is to change the model to a parallel programming paradigm. The paradigms we will be using are Open MP and MPI.

Algorithm

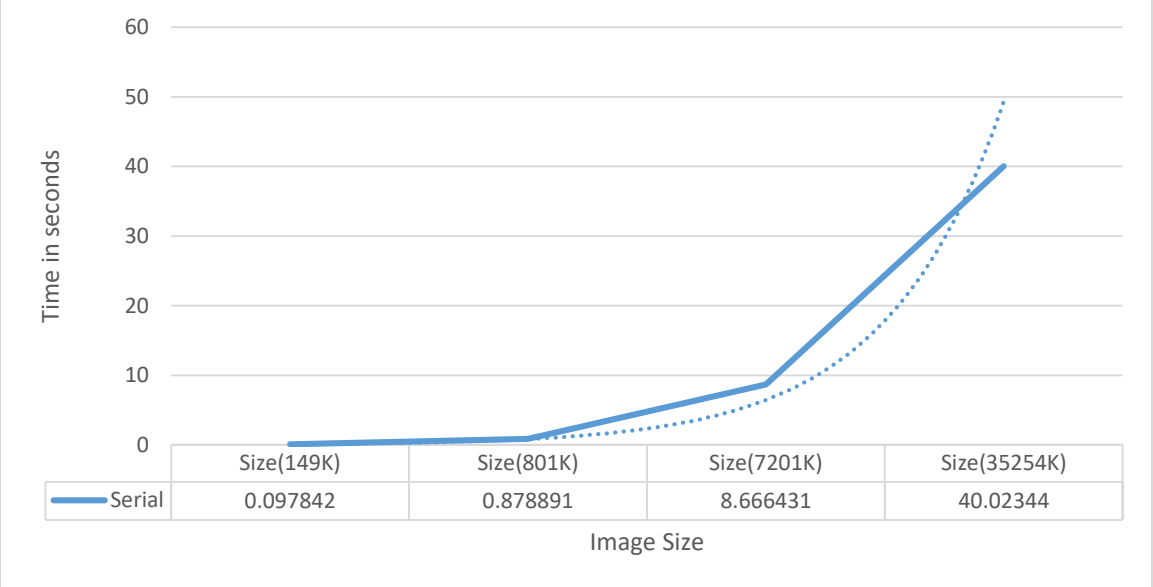
The main algorithm goes something like this:

1. Open the file to compress
2. Read the header file of the image
3. Count each pixel {3 bytes}
4. Create MinHeap
5. Create Huffman Tree
6. Encode Huffman Tree
7. Open new file
8. Write header files to new file
9. Compress data according to Huffman codes
10. Write Compressed Data to new file

This is the serial execution of the program. We will make some changes to parallelize this algorithm.

Serial Execution Time

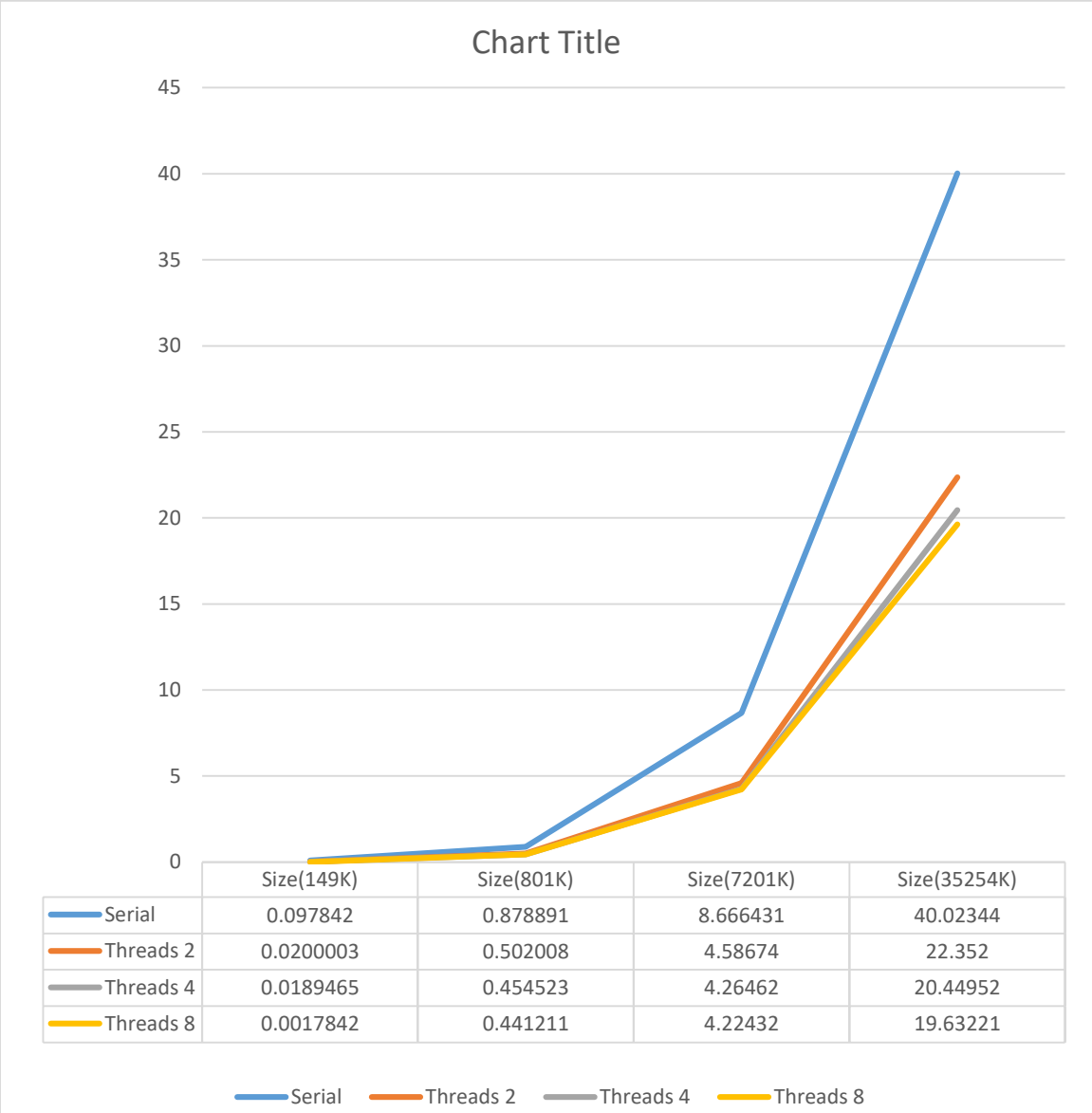
To execute serially, we will implement the algorithm mentioned above. Here are some examples of run times calculated on the above mentioned architecture:



As you can see, the time taken is exponentially increasing which shows how inefficient this algorithm is.

Parallel Execution Time (OMP)

Switching to OMP, we will need to devise a new strategy. We will now parallelize the data structures used to be able to handle multiple calculations at the same time. This is a graph of the time calculated:



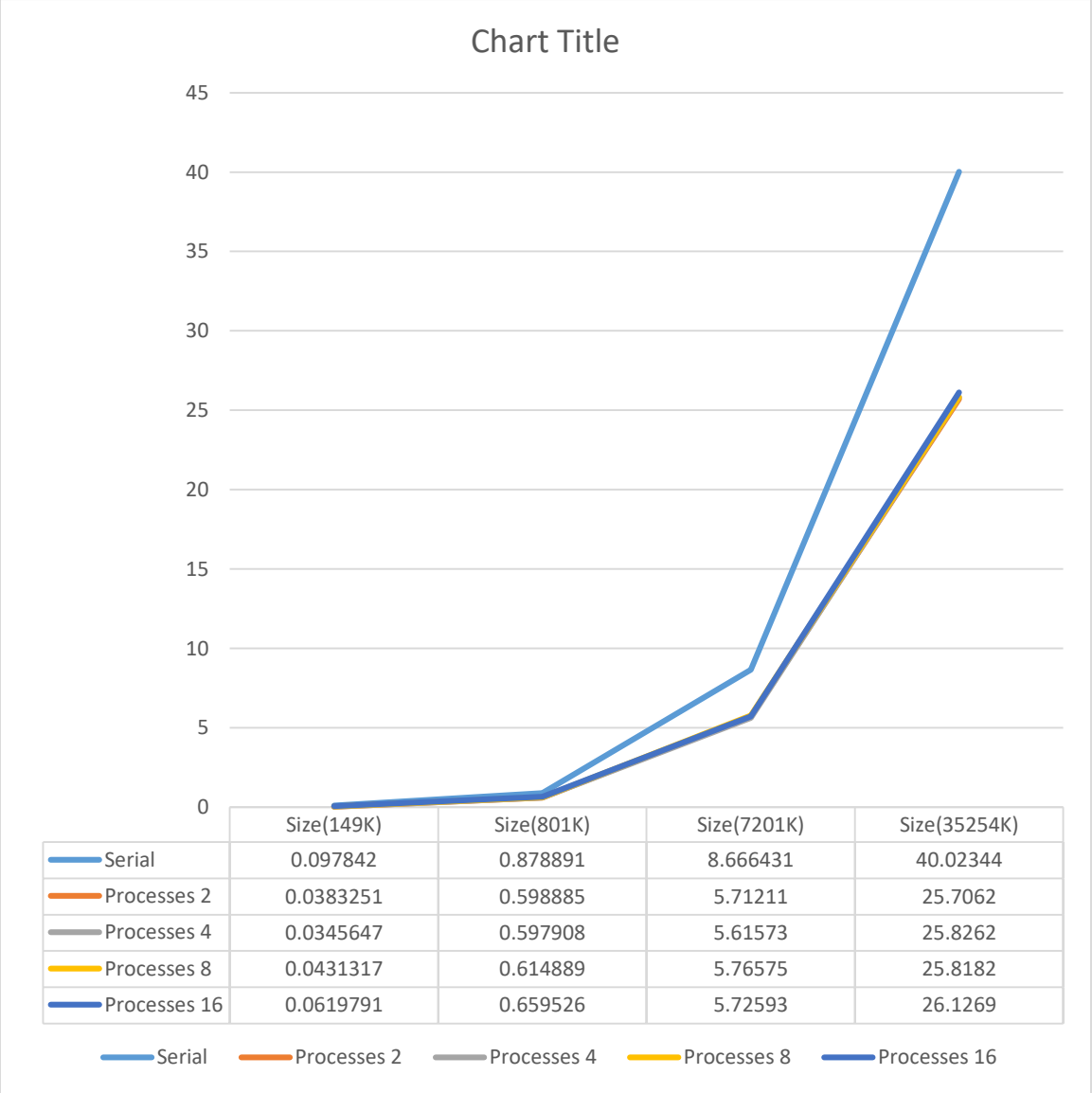
	Size(149K)	Size(801K)	Size(7201K)	Size(35254K)
Serial	0.097842	0.878891	8.666431	40.02344
Threads 2	0.0200003	0.502008	4.58674	22.352
Threads 4	0.0189465	0.454523	4.26462	20.44952
Threads 8	0.0017842	0.441211	4.22432	19.63221

The average speed up can be calculated for each size and then we can calculate a general speed up for this paradigm. Caps at 120 threads.

Size (149K) ->100-0.0017842/0.097842*100 = 98%
Size (801K) ->100-0.441211/0.878891*100 = 50%
Size (7201K) ->100-4.22432/8.666431*100 = 51%
Size (35254K) ->100-19.63221/40.02344*100 = 51%
Therefore the speed up can be seen to be approximately ~50%.

Parallel Execution Time (MPI)

Switching to MPI we will have to devise yet another strategy, but will be quite similar to that of OMP. We will now break our image into blocks. This is a graph of the time calculated:



	Size(149K)	Size(801K)	Size(7201K)	Size(35254K)
Serial	0.097842	0.878891	8.666431	40.02344
Processes 2	0.0383251	0.598885	5.71211	25.7062
Processes 4	0.0345647	0.597908	5.61573	25.8262
Processes 8	0.0431317	0.614889	5.76575	25.8182
Processes 16	0.0619791	0.659526	5.72593	26.1269

We can see that the graph for MPI is not that different from that of OpenMP. However it can be seen that when going past the system's limitations, the performance of the algorithms falls and becomes less efficient {like using 16 processes}. Caps at 1024 processes.

It can also be seen that OpenMP has faster times that those of MPI. This is because the MPI library function involve a message passing system whereas OpenMP follows a Fork-Join Parallelism model which allows us the define parallel regions.