Parallel LSB Steganography

The Mavericks

Florida International University

CDA 4104 Section U01 – Fall 2017

Abstract

Steganography, from Greek steganos, or "covered," and graphie, or "writing") is the art of hiding data behind other data, in such a way, that what poses as some type of data, for example an image, video file, audio file, or other data types, it actually contains hidden information not available at plain sight. The idea is not entirely about hiding information, but more about hiding the existence of it by camouflage. Our team took on the mission of hiding a message inside an image file in such a way that it would not modify the image, at least not noticeable to the human eye. Furthermore, the idea was to use a programming mechanism that would accelerate the time that it takes to “hide” a very large message inside an image. Like it is expected, the mechanism also applies to the reverse process, unveiling the message.

Introduction

There are many different strategies for practicing steganography. Our team used what it is called "Least Significant Bit algorithm for image steganography." Since an image is composed of a 2-D matrix, like it is explained in the "International Journal of Advance Computer Technology | VOLUME 3, NUMBER 4," the magic consist of using the individual grid points (pixels) to embed the secret message. In a colored image, a pixel is represented with 24 bits. Thus taking the least significant pixel give us one bit per pixel to store information. Since we are storing a message, every character takes 8 bits, hence we need 8 pixels per character.

Embedding a string inside an image is not an operation that is restricted to be done sequentially. The beginning of the string can be embedded in the image at the same time as the middle or any section of the string is being embedded into a different section of the image. Thus, our team decided to design a program that would divided the string and image into partitions of equal size and distribute the work load among the available processing cores of the computer.

This application utilizes the Message Passing Interface (MPI) library to distribute tasks among multiple processes for our encrypting / decrypting process. MPI is regarded as the standard message passing library among various software developers and organizations. MPI, or rather the Message Passing Interface, is not itself a library; in actuality, MPI is the specification for how a message passing library should be. That said, the main goal of MPI programs is to improve the performance of applications through parallel programming model, which is the goal of this project.

Process

The idea of this project is to parallelize the task of encrypting and decrypting a bitmap using MPI. To accomplish this, the first step of the process was to determine how the work should be split up among the different processes. However, to do this the root process must know ahead of time how many characters needed to be encrypted/decrypted to avoid sending useless tasks to any process. Once the number of characters is established, the encryption/decryption step begins.

An ASCII character in C is a total of 8 bits. Since we are using only 1 bit of each pixel to encrypt an image, a total of 8 pixels are needed to encrypt one character. Thus, a string of 20 characters, for instance, would require at least 160 pixels to encrypt.

This relationship is represented in the table below, as well as the image size one would need to encrypt the maximum number of characters represented by the header.

HEADER SIZE (BITS)	MAXIMUM CHARACTERS ENCRYPTED	IMAGE SIZE NEEDED (PIXELS)	APPROXIMATE IMAGE DIMENSIONS
1	1	9	3 by 3
2	3	26	5 by 5
4	15	124	12 by 12
8	255	2048	45 by 45
16	65,535	524,296	720 by 720
24	16,777,215	134,217,744	11,500 by 11,500
32	4,294,967,295	34,359,738,392	185,000 by 185,000

Step 1: Establishing the length of the hidden message

For decryption, the image must contain information on the number of characters that have been encrypted. This is done in the encryption step in the “header” of the image to be encrypted/decrypted. Essentially, the header is the first 16 or so pixels. The last bit from each of these pixels (the least-significant bit) is used to store the a bit corresponding to the value of the message’s length. This is a code snippet showcasing the process:

```
uint8_t      mask1 = 0b00000001,
             mask2 = 0b11111110;

int x;
for (x = 0; x < HEADER_SIZE; x++) {
       int bit = mask1 & (charsToEncode >> ((HEADER_SIZE - 1) - x));
       ((bmp) + x)->r = ((uint8_t)(bmp1 + x)->r & mask2) | bit;
}
```

As we see in the code snippet, each bit from *charsToEncode* is copied onto the least significant bit of the pixel with the use of masking and bit shifts.

Step 2: Dividing the task among each process.

This step is much simpler. For encryption, we already know the length of the message in the string. For decryption, we must first decode the header to find this value. After this is done, we assign each process with a number of pixels proportional to the total number of pixels required to represent the string message.

Step 3: MPI Serialization

In order to split up the tasks of encryption and decryption, we must be able to send pixels through MPI to each child process. Because pixels are a struct and MPI cannot simply see what a structure looks like on its own, we must define the layout of a pixel structure explicitly. This is done with the following code:

```
const int nitems = 3;
int blocklengths[3] = {1, 1, 1};
MPI_Datatype types[3] = {MPI_UNSIGNED_CHAR, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_CHAR};
MPI_Datatype mpi_pixel_type;
MPI_Aint offsets[3];
offsets[0] = offsetof(PIXEL, r);
offsets[1] = offsetof(PIXEL, g);
offsets[2] = offsetof(PIXEL, b);
ierr = MPI_Type_create_struct(nitems, blocklengths, offsets, types, &mpi_pixel_type);
ierr = MPI_Type_commit(&mpi_pixel_type);
```

Here, each pixel is defined in a structure that MPI can send and receive.

Step 4: Sending each task to each child process

Once the tasks are divided, each one is sent by the parent process via MPI's *MPI_Send()* function, and is thus received by the child process via the *MPI_Recv()* function. An example of sending a pixel this way is shown below.

```
ierr = MPI_Send(&bmp[pixelOffset[slaveID]], pixelsOut, mpi_pixel_type,
      slaveID, sendTag, MPI_COMM_WORLD);
```

Step 5: Encryption and Decryption

Once the process receives the pixels meant to encrypt or decrypt, they begin the process. The encryption is done similarly to the process shown in step 1 for the header, only this time the variable being encrypted changes every 8 bits (the number of bits in a character.

```
int octaCounter = 0;
int chartmp = messageIn[0];

int x = 0;
for (x = 0; x < pixelsIn; x++) {
      int bit = (mask1 & (chartmp >> (7 - (octaCounter % 8))));
      ((bmp2) + x)->r = ((uint8_t)(bmp2 + x)->r & mask2) | bit;
      chartmp = messageIn[octaCounter / 8];
      octaCounter++;
}
```

Decryption is a similar process, only it retrieves the value stored in the pixels rather than reassigning them.

```
int x;
int octaCount = 0, charValue = 0;
for (x = 0; x < pixelsIn; x++) {
      charValue += (mask & (uint8_t)(bmp2 + x)->r) << (7 - (octaCount % 8));

      octaCount++;
      if (octaCount % 8 == 0 && octaCount != 0) {
            [(octaCount / 8) - 1] = (char) charValue;
            charValue = 0;
      }
}
```

Once the task has been completed, the relevant information is sent back to the parent process through MPI. For encryption, these are the encrypted pixels. And for decryption, these are the decrypted characters.

Step 6: Putting it all together

Once each task has finished, the parent process must begin putting all the received information back together. To do this, the parent process must identify which child process it is receiving the information from, as the order of the information is important.

After this step, the program is done.

Measurements and Experimentation

In Part One, the variable is the number of words being encoded or decoded and times are recorded for the sequential version of the program and the parallel running eight cores. Values are collected at one hundred characters, one thousand characters, ten thousand characters, one-hundred thousand characters, one million characters, all the way through seven million characters in increments of one million at a time. Part two is an analysis of an interesting moment that Part One brought to light. Around two million words, the parallel version outperforms the sequential version. Part two takes a closer look at how the program behaves with values around this two-million-character mark. In part two, we also explore how using different numbers of cores affects the running time.

The image being encoded is a very large .bmp image. It is 9893 pixels wide and 6043 pixels tall. This means that it is composed of over 59 million pixels. Also, since we encode one character in eight pixels, it can hold about 7.4 million characters. This is an unrealistically sized image for steganography where the purpose is to hide sensitive content in plain sight. However, it will serve well for the purposes of testing how the two versions of the program handle a wide range of input sizes.

Results

Part One

Table of Recorded Values

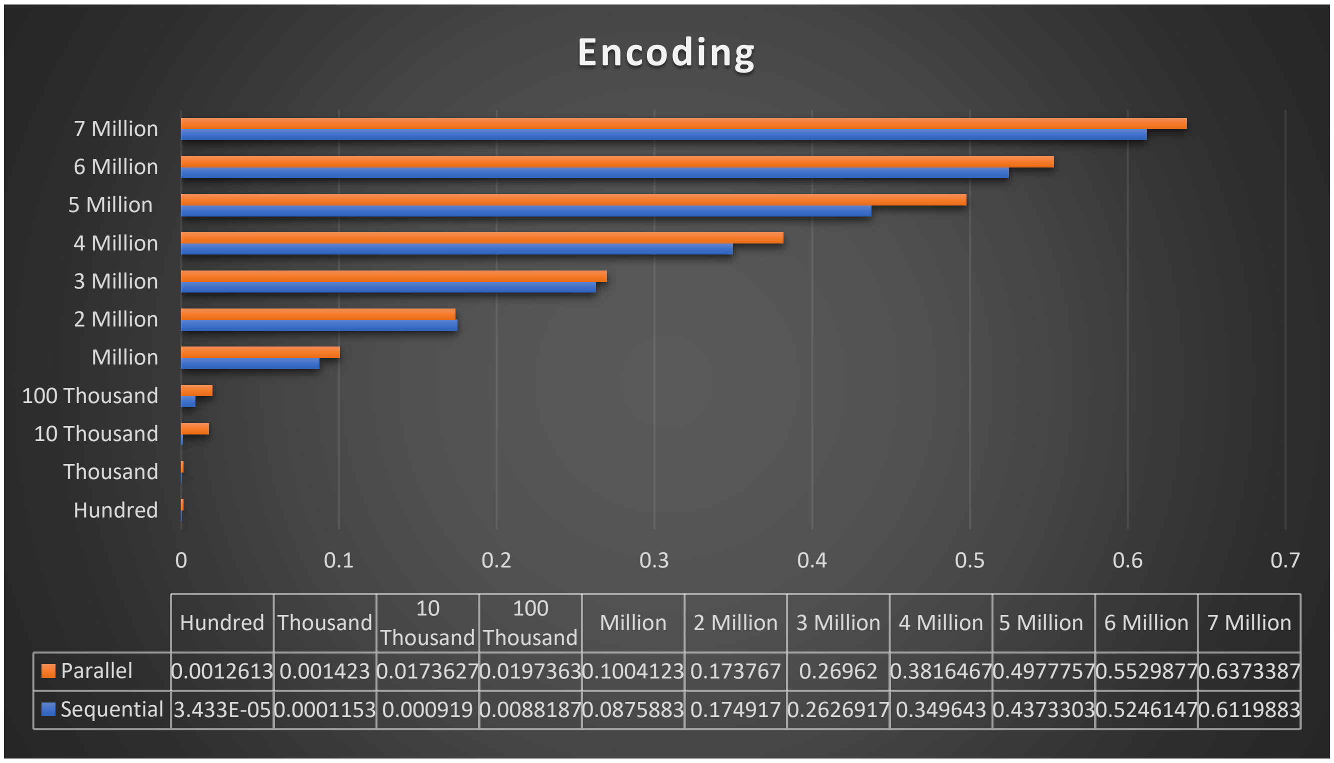
		Sequential		**Parallel (8 cores)**	
	Characters	**Encode Time**	**Decode Time**	**Encode Time**	**Decode Time**
Hundred	**100**	**0.000034**	**0.00003**	**0.001608**	**0.001023**
	100	**0.000035**	**0.00003**	**0.001663**	**0.000925**
	100	**0.000034**	**0.000029**	**0.000513**	**0.001231**
Thousand	**1000**	**0.000114**	**0.000095**	**0.001005**	**0.001116**
	1000	**0.000117**	**0.000091**	**0.002107**	**0.000889**
	1000	**0.000115**	**0.000103**	**0.001157**	**0.00144**
10 Thousand	**10000**	**0.000913**	**0.000732**	**0.015847**	**0.00551**
	10000	**0.000936**	**0.000728**	**0.016446**	**0.002385**
	10000	**0.000908**	**0.00074**	**0.019795**	**0.00208**
100 Thousand	**100000**	**0.008818**	**0.006954**	**0.026221**	**0.015689**
	100000	**0.008802**	**0.006932**	**0.017591**	**0.007952**
	100000	**0.008836**	**0.006933**	**0.015397**	**0.007315**
Million	**1000000**	**0.087749**	**0.06902**	**0.105634**	**0.051446**
	1000000	**0.087674**	**0.068948**	**0.085085**	**0.03616**
	1000000	**0.087342**	**0.069101**	**0.110518**	**0.055445**

	3 Value Averages				
	Hundred	**3.43333E-05**	**2.96667E-05**	**0.001261333**	**0.001059667**
	Thousand	**0.000115333**	**9.63333E-05**	**0.001423**	**0.001148333**
	10 Thousand	**0.000919**	**0.000733333**	**0.017362667**	**0.003325**
	100 Thousand	**0.008818667**	**0.006939667**	**0.019736333**	**0.010318667**
	Million	**0.087588333**	**0.069023**	**0.100412333**	**0.047683667**

		Sequential		**Parallel (8 cores)**	
	Characters	**Encode Time**	**Decode Time**	**Encode Time**	**Decode Time**
2 Million	**2 Million**	**0.174676**	**0.137482**	**0.197843**	**0.077766**
	2 Million	**0.175077**	**0.137231**	**0.167955**	**0.082716**
	2 Million	**0.174998**	**0.137668**	**0.155503**	**0.064745**
3 Million	**3 Million**	**0.262322**	**0.206362**	**0.273331**	**0.096811**
	3 Million	**0.263158**	**0.206919**	**0.297668**	**0.103414**
	3 Million	**0.262595**	**0.206474**	**0.237861**	**0.224806**
4 Million	**4 Million**	**0.349562**	**0.275449**	**0.400019**	**0.313578**
	4 Million	**0.349667**	**0.275188**	**0.351562**	**0.238446**
	4 Million	**0.3497**	**0.275207**	**0.393359**	**0.145691**
5 Million	**5 Million**	**0.437474**	**0.344076**	**0.519712**	**0.163633**
	5 Million	**0.43751**	**0.343552**	**0.421405**	**0.19242**
	5 Million	**0.437007**	**0.343194**	**0.55221**	**0.18342**
6 Million	**6 Million**	**0.524392**	**0.412211**	**0.480552**	**0.400683**
	6 Million	**0.524563**	**0.412545**	**0.491747**	**0.362139**
	6 Million	**0.524889**	**0.412589**	**0.686664**	**0.25368**
7 Million	**7 Million**	**0.612198**	**0.480627**	**0.576381**	**0.230449**
	7 Million	**0.61161**	**0.481142**	**0.703911**	**0.222326**
	7 Million	**0.612157**	**0.480754**	**0.631724**	**0.254108**

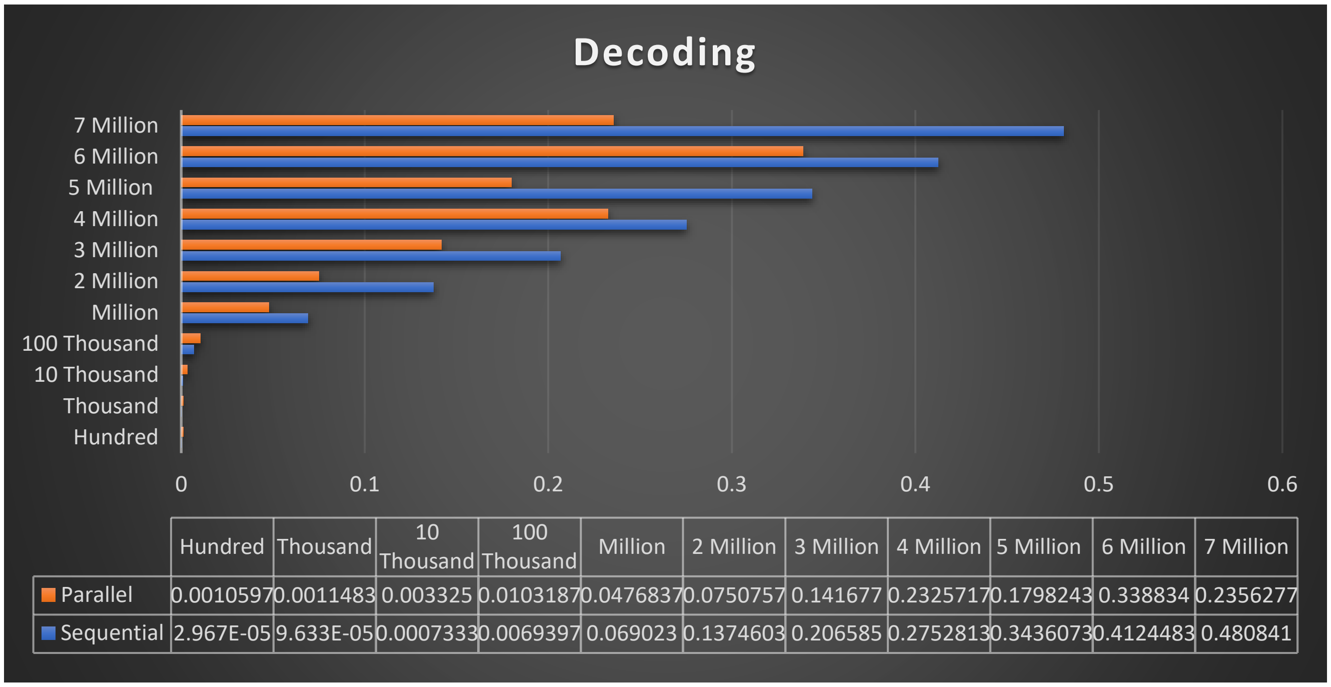
	3 Value Averages				
	2 Million	**0.174917**	**0.137460333**	**0.173767**	**0.075075667**
	3 Million	**0.262691667**	**0.206585**	**0.26962**	**0.141677**
	4 Million	**0.349643**	**0.275281333**	**0.381646667**	**0.232571667**
	5 Million	**0.437330333**	**0.343607333**	**0.497775667**	**0.179824333**
	6 Million	**0.524614667**	**0.412448333**	**0.552987667**	**0.338834**
	7 Million	**0.611988333**	**0.480841**	**0.637338667**	**0.235627667**

In these graphs, the vertical axis represents the number of characters being encoded in the pixels of an image and the horizontal axis represents the seconds that the programs take to run.



	Hundred	Thousand	10 Thousand	100 Thousand	Million	2 Million	3 Million	4 Million	5 Million	6 Million	7 Million
🟧 Parallel	0.0012613	0.001423	0.0173627	0.0197363	0.1004123	0.173767	0.26962	0.3816467	0.4977757	0.5529877	0.6373387
🟦 Sequential	3.433E-05	0.0001153	0.000919	0.0088187	0.0875883	0.174917	0.2626917	0.349643	0.4373303	0.5246147	0.6119883

The graph shows us that the sequential version of the program performs faster than the parallel in nearly every situation except around the two-million-character mark, where the 8 cores beat the sequential version by a about a thousandth of a second.



	Hundred	Thousand	10 Thousand	100 Thousand	Million	2 Million	3 Million	4 Million	5 Million	6 Million	7 Million
🟧 Parallel	0.0010597	0.0011483	0.003325	0.0103187	0.0476837	0.0750757	0.141677	0.2325717	0.1798243	0.338834	0.2356277
🟦 Sequential	2.967E-05	9.633E-05	0.0007333	0.0069397	0.069023	0.1374603	0.206585	0.2752813	0.3436073	0.4124483	0.480841

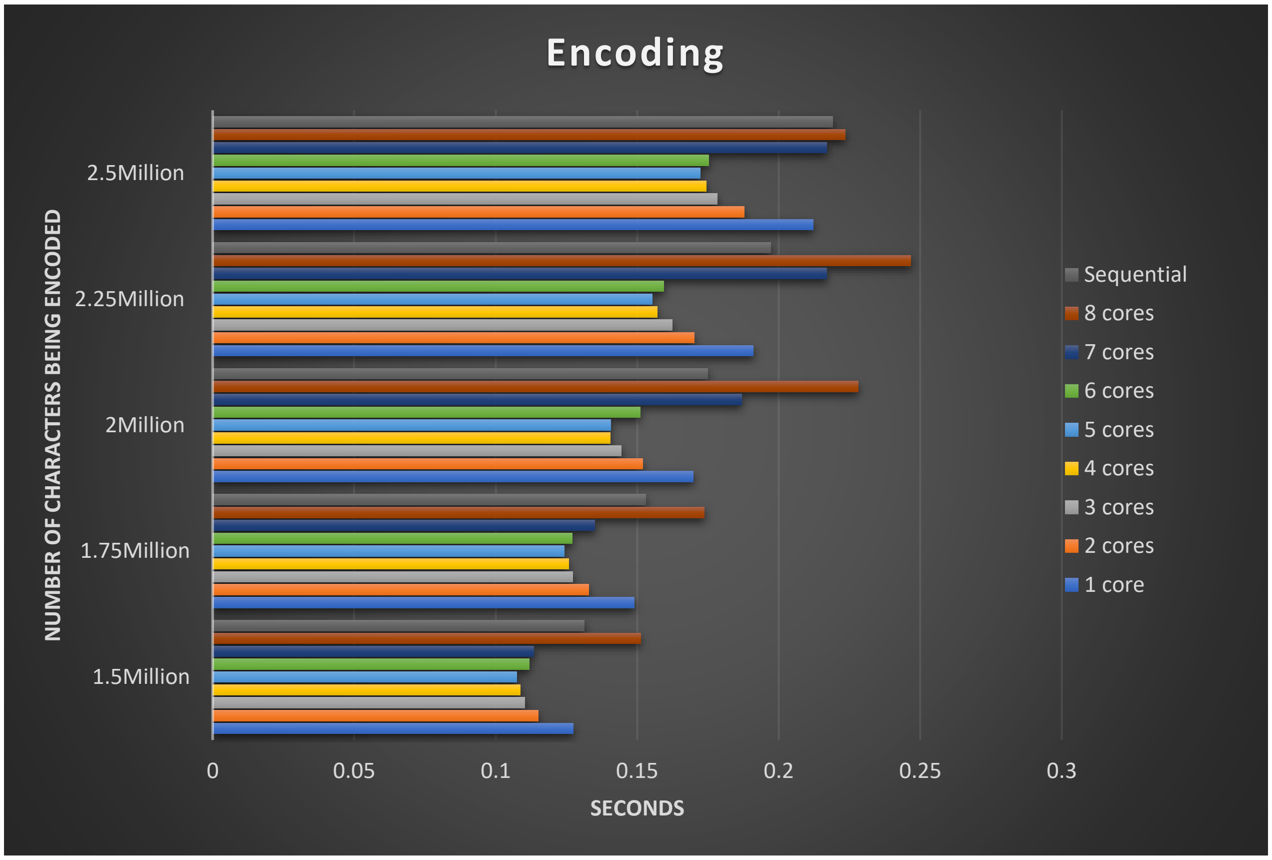
The parallel program begins to produce favorable results at one million characters, but then continues to get better with its best performance being with the encoding of seven million characters. Here it performs 49% faster than its sequential counterpart.

Part Two

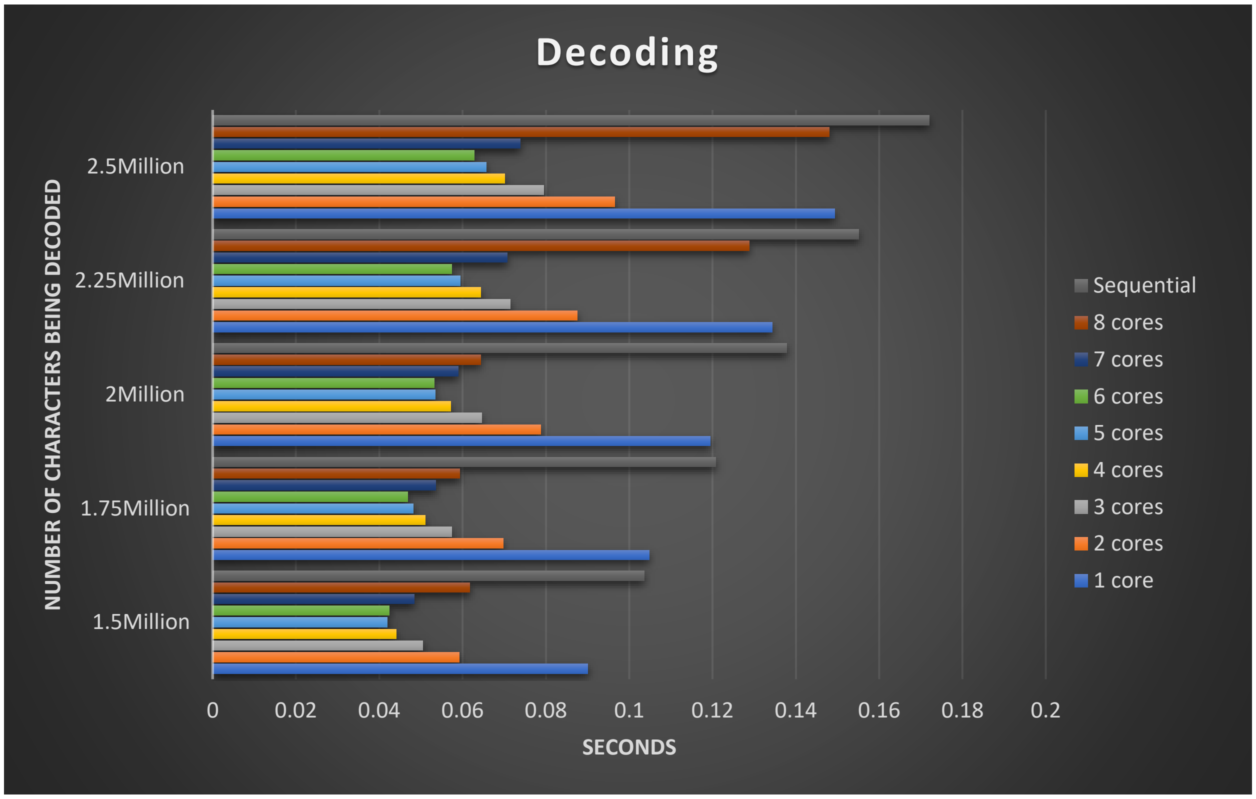
In this part, we present data collected around the two-million-character mark. In Part One, we observed that this is where the parallel version of the program beats the sequential version. The top, gray bar of each row of bars in the graph represents the sequential version of the program. The others represent how the parallel program behaves using different numbers of cores.

(The values collected for this part are included in the RecordedTimes sheet of the excel file attached, but have been omitted from this stylized presentation due to their cumbersome nature.)

Here again, the vertical axis represents the number of characters being encoded in the pixels of an image and the horizontal axis represents the seconds that the programs take to run.



It is noteworthy how using one through seven cores, at most of the data points ,outperforms the sequential program, while using all eight cores never outperformed the sequential version in this set of collected times.



In this graph, the best moment for decryption is at 2.5million words with 6 cores. Here, it is 36% faster than the sequential program. However, as we saw above, this is not the best overall moment for the parallel decrypt function as it continues to improve all the way through out last measurement at seven million characters.

Analysis

We were hoping to see an N-fold increase in performance, where N is the number of processes being used, but did not accomplish those results. In encoding, the parallel version of the program had its best moment with five cores and 2.5 million characters. Here, it showed a 79% boost in performance compared to the sequential version of the program. In decoding, the parallel version performed best at 7 million characters and 8 cores. It shows a 49% improvement over the sequential version. It is likely that these improvements are not better because of simplicity of the operation at the core of our algorithm. It is just changing one bit, so the overhead of distributing and then reassembling the image is not worth doing since one would not see the benefits of parallelization when dealing with messages and images of normal size.

Conclusion

Although we set out to improve the running time of the sequential version of our least significant bit steganography implementation by parallelizing the algorithm using MPI, we discovered that this is not a suitable method for such simple work. The transformation that lies at the core of this algorithm is simple turning a specific bit “on” or “off”. This is such a simple operation, that the MPI version of the program could not produce a significant improvement its sequential counterpart. This behavior is likely due to the overhead involved with partitioning the pixels and message to be encoded, sending them to the various processors, and then, the reassembling that takes place in the master processor. This parallelization would be more useful if apart from simply transforming the pixels to represent the message, the processors first had to encrypt each character of the message.

References

1. Barney, Blaise. Message Passing Interface (MPI). Lawrence Livermore National Laboratory. Webpage. https://computing.llnl.gov/tutorials/mpi/
2. Champakamala .B.S, Padmini.K, Radhika .D. K. *Least Significant Bit algorithm for image steganography.* Don Bosco Institute of Technology, Bangalore, India. Online PDF. http://ijact.org/volume3issue4/IJ0340004.pdf