2013 IEEE 12th International Symposium on Parallel and Distributed Computing

Data Hiding Using Steganography

Monica Adriana Dăgădiţă
Emil Ioan Sluşanschi
Răzvan Dobre
University Politehnica of Bucharest
Faculty of Automatic Control and Computers
Computer Science and Engineering Department
Email: monica.dagadita@cti.pub.ro, emil.slusanschi@cs.pub.ro, razvan.dobre@cs.pub.ro

***Abstract*—**
It is a truth universally acknowledged that "a picture is worth a thousand words". The emerge of digital media has taken this saying to a complete new level. By using steganography, one can hide not only 1000, but thousands of words even in an average-sized image. This article presents various types of techniques used by modern digital steganography, as well as the implementation of the least significant bit (LSB) method. The main objective is to develop an application that uses LSB insertion in order to encode data into a cover image. Both a serial and parallel version are proposed and an analysis of the performances is made using images ranging from 1.9 to 131 megapixels.

I. Introduction

Throughout time, confidentiality has always been important. Whether it is carved in stone, written on paper or sent over the Internet, correspondence between two persons is exposed to tampering or eavesdropping. Therefore, it is necessary to provide a mechanism that protects written information. One of the most common methods of securing transmitted data is cryptography. The purpose of cryptography is to disguise *plaintext* in such a way that it becomes unreadable. The resulted text, called *cyphertext* can then be safely transmitted to the destination. Knowing the algorithm used for encryption, only the receiver will be able to obtain the original message.

Access to powerful computers does not only mean better encryption. It can equally be a tool used for breaking a cipher or decrypting a message. No matter how powerful the encryption algorithm is, encrypted data will always arouse suspicion. This is where steganography can help. Steganography can be defined as "the art and science of communicating in a way which hides the existence of communication" [1]. Although cryptography and steganography are often confounded, they are essentially different. Whilst the first one "scrambles a message so it cannot be understood", the other one "hides the message so it cannot be seen" [2]. It is commonly believed, mistankenly, that steganography could replace cryptography. On the contrary, by using the two techniques together one can create a solid and powerful encryption system, better than each of the two components.

With digital data as a mean of communication, messages can be hidden in the ones and zeros of text files, pictures, audio or video files. However, in order for the message to remain secret, only certain bits can be used. Hence, before the beginning of the information-hiding process, the stenographic system must identify the redundant bits - those bits that can be modified without altering the integrity of the file [3]. After this step, the least significant bits from the cover medium can be replaced with data that has to be hidden.

II. State of the Art

A. Overview of Steganography

Almost all digital file formats can be used for steganography. However, the most suitable ones are those with a high degree of redundancy. Redundancy is defined as "the bits of an object that provide accuracy far greater than necessary for the object's use and display" [4]. According to this criteria, images and audio files are the most suited for this purpose.

"Camouflaging" the message is essential. It is the main asset of steganography. Once the presence of the message is detected, its contents are no longer safe. This is why data encryption is recommended to be used in conjunction with data encoding. Instead of directly hiding a message, in a preprocessing stage it could br encrypted and afterwards encoded into the cover file.

1) Text Steganography: An example of a very famous approach is hiding the secret message in every n^{th} character of a text. It is not very secure, as knowledge of n automatically reveals the secret. A safer way of data hiding is using a publicly available cover source, like a book or a newspaper and using a secret code that contains sequences of three numbers - page, line and character index. This way, the message can only be revealed by having both the secret code and the stego cover.

2) Image Steganography: Hiding information inside images is the most popular technique nowadays. The main reason is that in the Internet traffic this type of media is the most frequently used. However different image-based algorithms are, there is one common factor among all, namely they all take advantage of the limitations of the human eye. If the message is hidden in such a way that the cover does not arouse suspicion, it will most probably not be discovered. On the contrary, if the noise in an image is visible or there are any other signs of the initial image being altered, a steganalysis tool will easily decode the hidden text.

3) Audio and Video Steganography: There are several ways in which information can be hidden inside an audio file. A common method is using the least significant bits. This

978-0-7695-5018-3/13 $26.00 © 2013 IEEE
DOI 10.1109/ISPDC.2013.29

way the modification of the initial file will not generate audible sounds. Another well known practice is masking, where a faint, but audible sound will become inaudible if in the presence of another louder but audible sound [4]. Also exploiting human limitations it is possible to encode data using frequencies that are inaudible to the human ear. Above $20,000$ Hz, no message will be detected [5]. Video files are generally collections of images and sounds, and consequently all methods applicable to these two file formats can also be used for videos.

B. Image Steganography

Images are the most popular files used for data hiding. For almost all image file formats there is at least one steganographic algorithm. All these algorithms can be grouped in three categories: spatial domain, frequency domain, and adaptive methods [6].

1) Image Spatial Domain Steganography: Spatial domain methods are based on encoding at the level of the LSBs. Least significant bit insertion is a common, simple approach to embedd information into a cover image. Depending on the size of the message and that of the image, one can use from the 1^{st} to the 4^{th} LSB. Using four bits however, will most likely produce visible artefacts. In its simplest form, LSB makes use of BMP images, since they use lossy compression [4]. Pixels' values are stored explicitly, easing the process of data embedding.

Palette based images such as GIF files are also a very popular format, which supports up to 8 bits per pixel, thus allowing a single image to reference a palette of up to 256 distinct colors. One disadvantage of this method is that an altered pixel value could point to a totally different color than the initial one. If for BMP images, modification of the LSB means a new but very similar color to the original one, in this case the new value is a new palette index. A solution is the sorting of the lookup table so that similar colors have adjacent positions.

2) Image Frequency Domain Steganography: The most common file format used for this type of steganography is JPEG. It is a very popular file format, mainly because of its small sizes. JPEG is a frequently used method of lossy compression for digital photography. The degree of compression can be adjusted, allowing a selectable tradeoff between storage size and image quality. The first step of JPEG compression is converting the RGB representation to YUV. Y corresponds to luminance or brightness, while U and V stand for chrominance and color. Taking advantage of the increased sensitivity of the human eye to changes in brightness, the color data is downsampled in order to reduce the total size of the file. The second step is the actual transformation of the image; usually a Discrete Cosine Transform (DCT) is used, but there is also possible to use Discrete Fourier Transform (DFT) [4]. The next step is quantization. Here, another property of the human eye is exploited, namely that the human eye is good at spotting small differences in brightness over a relatively large area, but not that good so as to distinguish between different strengths in the high frequency brightness [4]. Consequently, magnitudes of the high-frequency components can be stored with a lower accuracy than the low-frequency components. And finally, the resulting data for all $8x8$ blocks is compressed with a lossless variant of Huffman encoding.

With a lossy compression scheme, JPEG files were at first considered useless for steganography. Hiding information into the redundant bits would have been of no use, since these are left out in the compression process. However, properties of this algorithm have been exploited in order to develop steganographic techniques. One of the most important aspects of JPEG compression is the fact that it is divided into lossy and lossless stages. The DCT and quantization phases are lossy, but the final Huffman stage is not. Consequently, LSB insertion could be done exaclty before the Huffman encoding is applied. Encoding data at this stage is not only safe – as there will be no loss –, but it is also difficult to detect since it is not in the visual domain.

3) Adaptive Steganography: The previous two sections described algorithms that work either with the image spatial domain or the frequency domain. There are however steganographic techniques that make use of both. These methods take statistical global features of an image before attempting to interact with its LSB or DCT coefficients [6]. Then, based on these statistics, the most suitable places for data hiding will be identified. In this way, the stego-message will be less likely to be discovered. One of the simplest and most used methods of steganalysis is noise detectection – a good algorithm will avoid smooth areas of uniform color. A complex color scheme is also important, as variations will not be easy to detect.

C. Steganlysis

As more and more techniques of data hiding are developed, methods of detecting the use of steganography also advance [5]. Analogous to cryptanalysis, steganalysis is the science of detecting messages that are hidden using steganography. In order to detect secret data, different image processing techniques can be applied, such as: image filtering, rotation, cropping and translation. The stego-image's structure must be analysed in order to measure its statistical properties: histograms, correlations between pixels, distance and direction [6].

Among the challenges steganalysis has to deal with we consider that:

- The suspect file may or may not have hidden data encoded in its contents.
- Hidden data can be encrypted before encoding.
- The suspected file may contain noise of irrelevant data encoded.

The cover data can be a text, an image and even an audio or video file. With the development of steganographic techniques adapted for each file format, different types of steganalysis methods have also emerged. Due to the increased popularity of digital image steganography, image steganalysis techniques are the most numerous ones. A widely used method involves statistical interpretation. While LSB might not seem

very important, it actually can offer plenty of information regarding the contents of an image. The assumption that the least significant bits are mostly random, is wrong. Even the modification of a small percentage of LSBs could be easily detected. In fact, it is desirable that all the LSBs are modified, although the image's storage space is too large compared to the message's size.

III. Hardware and Software Configuration

All tests of the application were done on the NCIT Cluster of the University Politehnica of Bucharest. For the profiling and development part we used the dual processor Quad-Core Intel 2GHz Xeon E5405 IBM HS21 systems. For the testing we used the dual processor Six-Core AMD 2.6 GHz Opteron IBm LS22 systems. Both LS22 and HS21 systems are running Red Hat Enterprise Linux. The compiler used is gcc version 4.6.0 and for profiling we eomploy Sun Studio Performance Analyzer.

IV. Algorithm Description

Hiding information inside images is one of the most popular steganographic techniques used nowadays. The method we choose for data encryption is the Least Significant Bit (LSB) approach using BMP images.

A. Bitmap Images

An important property of this type of images is the fact that all pixels are explicitly written in the file, a number for each component of each pixel: red, green, blue. A popular type of bitmap files is that in which each pixel is represented using 24 bits - one byte for each color component. This gives 256 possibilities for each color plane, respectively $16,777,216$ for one pixel. Altering the least significant bits will result in a color slightly different from the original one. What is most important to steganography is the fact that the human eye is unable to detect such differences.

B. Data Hiding Using LSB Insertion

Least significant bit insertion is one of the most commonly used methods of data hiding, due to its simplicity. The amount of data that can be hidden into an image depends on the size of the image and the number of least significant bits used for encryption. Having a $200x200$ image and using only the least significant bit from each color component one gets $120,000$ bits that can be used in the process of data hiding. If we want to hide characters, the amount of data that can be encrypted is $15,000$ characters. This would represent a text that is twice as long compared to the Declaration of Independence [1]. However, if we need more space we can always use a bigger image or more bits from each pixel, taking into consideration that the more bits we use, the lower the quality of the final image will be.

The process of data hiding is quite simple. Using one bit from each color component of a pixel, we get three bits per pixel; this means we need three pixels to hide a letter, i.e. one byte. Figure 1 contains an example of how the

11011001	01000110	11100111
00010010	01100111	00001011
00111000	11010101	00100011
1101100**1**	0100011**0**	1110011**0**
0001001**0**	0110011**0**	0000101**0**
0011100**1**	1101010**1**	00100011

Fig. 1. Example of LSB insertion

letter $A(10000011)$ can be hidden into an image, using only the LSB for each pixel's color component. The first three lines are the original values for the three pixels needed. The first column is red, the second blue and the third green. The last bits from the next three lines hide the letter A – the underlined zeros and ones – the red color marks the bits that have changed in order to hide the data.

As shown in Figure 1, only some of the bits' values change. On average, the LSB requires only half the bits in an image to change [2]. Moreover, even using both the least and the second least significant bit will not affect the image noticeably.

11011001	01000110	11100100
00010001	01100101	00001011
10111000	11010100	00100001
11011011	01010100	11100100
00010001	00100110	01001011
01111000	11010101	00100010
11011001	01000101	10100111

Fig. 2. Section of a stego-image

C. Data Recovery

The process of data recovery is a very simple one. The needed input is the image and the number of bits used. After reading the color values for each pixel, extracting the corresponding bits and concatenating their values, the hidden character can be easily recovered.

An example of how data can be extracted from a stego-image is shown in Figure 2. In this scenario data is hidden using two bits from each color component. Extracting the last two bits of each number and grouping them eight by eight we obtain 01100001 (*a*), 01110000 (*p*), 01110000 (*p*), 01101100 (*l*), 01100101 (*e*). So, the hidden word is "apple".

V. Implementation

This section discusses the two implementations of the previously described algorithm. Both encoding and decoding are implemented in serial and parallel versions by using PThreads [7], [8]. Furthermore, in order to balance the work-load, the boss-worker model [9] is used.

A. Serial implementation

The serial implementation consists of three main functions, namely:

- **readBMP()** - map the input image to a local data structure;
- **hideData()** - encode the desired message using the least significant bits corresponding to each pixel;
- **getData()** - receive the cover image as input, extract the least significant bits, concatenate them, and output the hidden data.

B. Parallel implementation

Before implementing a parallel version of the algorithm we profiled the serial version. The results is shown in Figure 3. All three important functions, *readBMP(), hideData()* and *writeImage()* take almost the same amount of time. The approach used to speed up the process was to overlap reading the input image and hiding the data for the first part an then also overlap reading the stego-image and recovering data, for the second part.

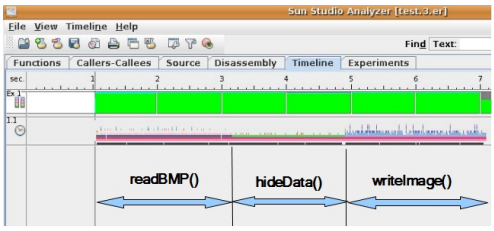

Fig. 3. Profiling for serial data hide

In order to balance the work as much as possible we used the boss-worker threading model. The parallel implementation uses PThreads. Similar to the serial version of the algorithm, the parallel code also consists of two main parts, one for data hiding and one for data decoding.

1) Hide Data: Taking advantage of the boss-worker threading model, the data hiding process is divided into smaller jobs that are processed by worker threads. The size of the job is very important – the total number of jobs should be at least equal to the number of available threads. In such a situation, the total processing time would be setted by the slowest thread. On the other hand, having smaller jobs would allow faster threads to work more and thus considerably reduce the total amount of time.

In order to divide the workload, the input image must be split into small slices, one per job. The serial version of the algorithm only depends on one constant, that is the number of bits per pixel that can hide data. The parallel version has one additional parameter, which is the number of pixels per slice. Having these two values, the boss thread can compute the amount of data that can be hidden into a slice of the image.

Slicing the image into small pieces also assumes dividing the text to be hidden into fragments. Taking into consideration the number of bits used from each pixel and the size of an image chunk, the boss thread assigns chunks of files to each job.

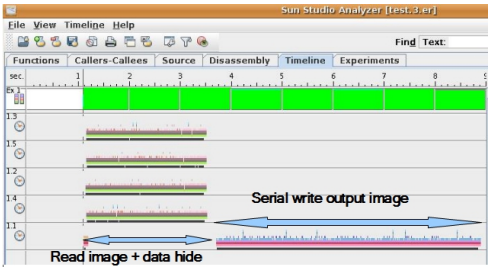

Fig. 4. Profiling of parallel hide data - 4 threads

In Figure 4 one can see the profiling of the parallel version for data hiding, using 4 threads. There are two main sections here: the parallel one, corresponding to image reading and data hiding and the serial one, representing the *writeBMP()* function. Although the total amount of time decreases, there is still a problem – no matter how much we speed up the first section, writing the final image will always have a constant time, thus affecting the total speedup of the algorithm, according to Amdahl's Law. The obvious solution would be for the worker threads to write the file concurrently. However, this would require a parallel file system. A compromise solution was to write parts of the output file separately and then concatenate them using a bash script. Consequently, profiling an improved version of *hideData()* can be observed in Figure 5.

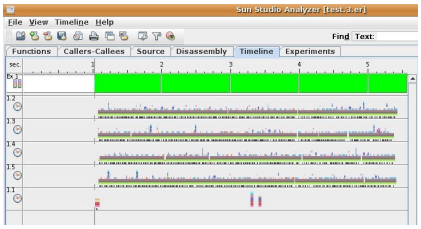

Fig. 5. Profiling of parallel hide data + parallel write image - 4 threads

2) Recover Data: The process of recovering data is a simple one. After extracting a job from the queue, the worker thread starts reconstructing the original message char by char. Profiling of the parallel version of data recovery is exibiting a similar behaviour to the improved version of the parallel version of Hide Data, as shown in Figure 5.

VI. Case Studies and Results

The first thing we tested after the serial version of the algorithm was complete, was the "capacity" of an image. The test image's size was 750 x 563. The input text files used for testing were books downloaded from www.gutenberg.org. However, the capacity of an image is not the only important aspect. We are also interested in the "ability" of an image to really hide the message it is carrying. In order to test both aspects – the capacity as well as the hiding efficiency – we

used 1 to 8 bits to encode data. The results obtained are shown in Figures 6 and 7.

Fig. 6. Output image after encoding - upper-left: 1 bit, upper-right: 2 bits, lower-left: 3 bits, lower-right: 4 bits

As depicted in Figure 6, using up to four of the least significant bits in a color component will not affect the quality of the image significantly. However, the last image, corresponding to the test where four bits were used, is visibly different from the first one. The amount of data hidden in this test case is $624K$ of text, about the first nine and a half chapters of Jules Vernes' *Twenty Thousand Leagues Under the Sea*.

Furthermore, Figure 7 contains output images for tests in which 5, 6, 7 and even all 8 bits were used.

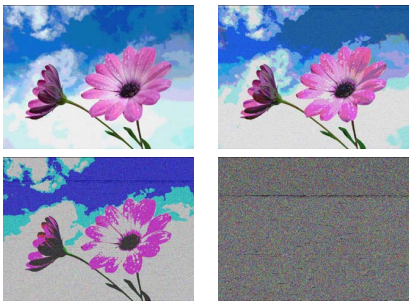

Fig. 7. Output image after encoding - upper-left: 5 bits, upper-right: 6 bits, lower-left: 7 bits, lower-right: 8 bits

As expected, all images exibit increasing amount of noise. The amount of data hidden is obviously higher, but any of the four images above would arouse suspicion. As mentioned before, the key factor in steganography is hiding the message transmitted. This is why there has to be a compromise between the size of the message transmitted and the aspect of the final stego-image.

In order to test the scalability of the application we used 3 different image sizes:

Test case 1: size 1600 x 1200 (1.9 MP)

Test case 2: size 6048 x 4032 (24.3 MP)

Test case 3: size 13413 x 9821 (131.7 MP)

For each test case we ran the serial version of the algorithm and the two parallel versions, i.e. with the serial write and parallel write stego-image respectively. In order to be able to hide enough data and not affect the image visibly, we used two and three of the least significant bits from each color component of a pixel. The results obtained are presented in the following subsections.

A. Test Case 1

This is the smallest test image used in our experiments, with 1600 pixels in width and 1200 in height. When using only the two least significant bits, this image can hide up to $1,440,000$ bytes of data.

1) Hide Data: The smallest time obtained for the serial version of the algorithm is **0.45 s**. Afterwards, tests were made using 2 up to 12 threads.

For this image size, serial writing of the output image did not affect the total time significantly. Still, we also tested the second version of parallel hiding data. As mentioned in the previous chapter, the output obtained are smaller files that must be concatenated in order to obtain the final stego-image. This final operation is achieved with a bash script. The amount of time needed for it is about 0.2 s and it must be added to the total time hiding data and creating the smaller files take.

A comparison between the performances of the two parallel versions of the code is given in Figure 8. We can see that neither of the parallel versions of the algorithm brings a significant improvement in matters of time. Maxim speedup reached is 1.63, when running the first version of the parallel algorithm on 10 threads. Regardless the number of threads used, the first version seems to be more efficient. The main reason is the increased number of I/O operations compared to the amount of data processed, for the second version.

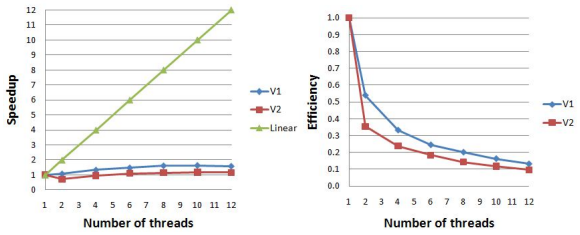

Fig. 8. Hide Data - Version1 vs Version2 - left: speedup, right: efficiency

The input and output images can be observed in Figure 9. Using 2 bits from each color component to hide data, the final final stego-image contains 1.4 MB of text. This equates to Jules Vernes' *Twenty Thousand Leagues Under the Sea* and *Abandoned*, as well as the first nine chapters of Lewis Carroll's *Alice's Adventures in Wonderland* . There are no visible differences between the two images, meaning the message is safe to be transmitted, without risking to arouse suspicion.

2) Recover Data: This section presents the results obtained for the second part of the algorithm, recovering the data hidden. Similar to data hiding, there are also three main steps: reading the input stego-image, decoding the message and writing it to one or more output files.

The small amount of data that can be hidden into a 1.9 MP image, makes the parallel version of data recovery inefficient.

Fig. 9. left: image before data hiding, right: image after data hiding

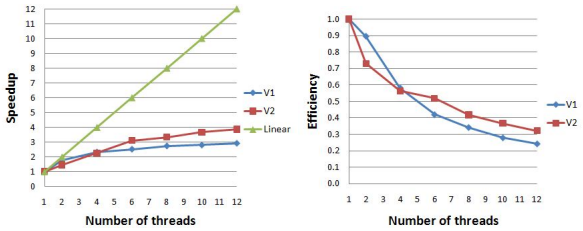

Fig. 10. Hide Data - Version1 vs Version2 - left: speedup, right: efficiency

In all test cases - from 2 to 12 threads - the serial version proved to more adequate.

B. Test case 2

This subsection presents results obtained for the middle-sized image, of resolution $6048x4032$, meaning 24.3 MP. Because of the high resolution of the image, we decided to use 3 of the least significant bits in each color channel. This leads to a total of 27 MB of space available for the message, still without visible changes in the final stego-image.

1) Hide Data: For this test case, the total serial time for data hiding was **8.81 s**. The final stego-image contained 24 text files, amongst which: *Anna Karenina, Moby Dick, Les Miserables, The Count of Monte Cristo*, as well as the complete works of Jane Austen and Jules Verne.

For the first parallel version of the algorithm, the smallest time was 3.013 seconds, for 12 threads. The slow decrease of the running times is a direct consequence of the serial writing of the final stego-image, accounting for about 2.7 seconds, as well as for the intensive I/O operations.

The maximum speedup reached for the first version of parallel data hiding is 2.92. However, the most efficient test case was the one using 2 threads – with an efficiency of 0.9. For more than 2 threads, even though speedup continues to slowly increase, the efficiency of the algorithm decreases fast. We did not use more than 12 threads because it was the maximum number available and also because it would not have improved the total time due to the I/O serial overhead.

The second version of the parallel data hiding proved to be more efficient for this test case, as illustrated in Figure 10. Concatenating the pieces of the output image using the bash script took 1.2 seconds. Added to this, the best time obtained was 2.28 seconds for 12 threads. If version 1 was more efficient for the previous test case,, in this 24 MP image parallel writing makes a difference. For more then 6 threads, the speedup of the second version increases faster than the one corresponding to the first version of the algorithm. Although the evolution is not a spectacular one, it can't be expected to be better than this for the computational intensity exibited by the present algorithms.

Though 3 bits were used, the initial and the stego-image cannot be differentiated only by looking. A higher resolution means more details and therefore more places to hide data. However, when using 4 bits of this image, the output becomes blurry. Figure 11 shows the 24 MP image before and after data hiding.

Fig. 11. left: image before data hiding, right: image after data hiding

2) Recover Data: Data recovery was also accelerated almost 4 times with the parallel version. Once again the use of 12 threads brought the best performances, leading to a 3.92 speedup. The serial time was thus reduced from 5.12 to 1.31 seconds.

Using 3 bits from each color channel and $500,000$ pixel chunks, the image was divided into 49 slices, each containing around 0.5 MB of data. 49 slices equals 49 'recover-jobs' that were put in the queue by the boss. Each one of them was processed by a worker and the data recovered was written into a file, thus leading to 49 output files. All the output files were then unified using a python script in 0.72 seconds.

C. Test case 3

The image used for this test case was the one with the highest resolution. It is a roadmap of Romania, having $13,413$ pixels in width and 9821 pixels in height. We decided to also use 3 bits from each color component for this case. This leads to 148 MB of space to store the secret message.

1) Hide Data: With such a high resolution image and almost 150 MB of data to hide, the serial version took **47.88 s**. The first parallel version only managed to reduce the total amount of time to 15.05 seconds, for 8 threads. The maximum achieved speedup is 3.18 for 8 and 12 threads.

Similar to the first parallel version, the second one also reaches a maximum speedup for 8 threads. In this case performance is slightly better, with a maximum speedup of 3.85. The duration of the final image parts concatenation was 4.4 seconds, leading to a minimum time of 12.42 seconds. In terms of efficiency, the first version was better, with 0.92 for 2 threads. For the second version, the highest value was also obtained for 2 threads, but it was 0.89.

Up to 4 threads with both versions of the algorithm have the same efficiency are depicted in Figure 12. For 6 and 8 threads the second version's parallel write of the output file pieces and their concatenation is less time consuming and for more

than 10 threads the overhead introduced by creating the worker threads is higher than the time saved having more workers.

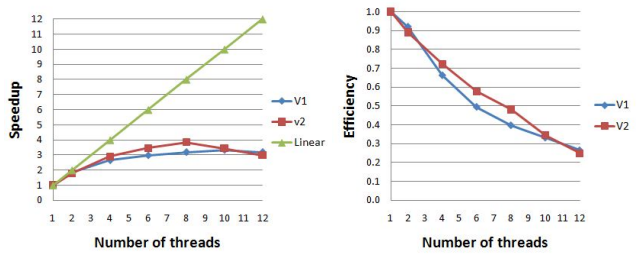

Fig. 12. Hide Data - Version1 vs Version2 - left: speedup, right: efficiency

Figure 13 contains the initial image of the roadmap and the final image, after data hiding is done. Although the second image contains a 146 MB text message there are no noticeable differences.

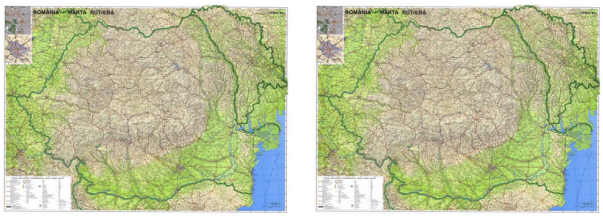

Fig. 13. Left: image before data hiding, Right: image after data hiding

2) Recover Data: Serial data recovery took **24.27** seconds. The parallel algorithm managed to reduce this time to a minimum value of 9.1 seconds, for 8 threads. With pixel chunks of size $2,000,000$, the image was divided into 66 slices. The resulting 66 output files created by the worker threads were merged into the final stego image in 2.94 seconds.

D. Image Hiding

The image size in the previous test case is obviously too large for sending just text. It is not often that someone wants to hide 130 MB of text. However, a 132 MP image could be used to hide another type of message, such as other images. Text is not the only way two persons can communicate. Lately image traffic over the Internet has increased exponentially. Why not use steganography to hide other images? Whether the message is a text or an image, it can be reduced to a sequence of ones and zeros. So, if a text message can be hidden, so can an image.

The algorithm that we implemented takes as input ascii files. In order to hide images we used the *PPM (portable pixmap)* file format, which is very similar to the BMP format. The header however is simplified - it contains an identifier of the PPM format, the width and height of the image and the maximum color value. A pixel is represented by three numbers, corresponding to the red, green and blue components. In contrast to the BMP pixel matrix, in a PPM image pixels are stored in normal order, just as they appear on the screen.

The file that we wanted to hide is an image of a galaxy, taken by one of NASA's space telescopes. Its dimensions are $4,000x4,000$ pixels, leading to a 16 MP image. Converted to PPM format, the space occupied by the image is 119 MB. This image to be encoded is presented in Figure 14.

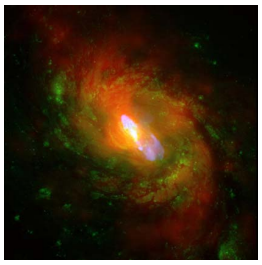

Fig. 14. Stego message - image of a galaxy

Having such a large input file as stego message, we decided to use the roadmap image described in the previous section as cover. Using 3 bits to hide data, it provided enough space to hide the complete input image. The final stego-cover can be observed in Figure 15.

Fig. 15. Final stego image

VII. Conclusions

The goal of this project was to implement an application that uses the LSB steganography method in order to hide and recover data. Because communication involves a sender and a receiver, there are two ways in which the application can run: as an encoder or as a decoder. For the encoding part the message is hidden into the least significant bits of a bmp image, thus resulting the stego-image. This image is then given to the decoder to extract the data that has been hidden.

The parallelization of the serial version of the algorithm was done using boss-worker threading model. Both data hiding and data recovery can be done in parallel. Neither process depends on the number of threads, and thus any combination of threads can be used. The only variables that must be constant for both processes are the number of bits used from each color component and the size of the chunks in which the input image is divided in order to parallel encode and decode the data.

To test the limits and performance of the algorithms we used three different images of: 1.9 MP, 24.3 MP and 131.7 MP respectively. The messages that were hidden into the input images were either books or other images.

The best performance for the data hiding part was obtained for test cases 2 and 3. For the smallest image, parallelization is not really justified, because the overhead introduced by the creation of boss and worker threads is too much compared to the time saved with parallel processing. This is why maximum speedup reached for this test case was only 1.61 for 8. For the other two tests, parallel hiding was more efficient with a speedup of 3.86 for 12 for the middle image and a speedup of 3.85 for 8 threads on the largest image. However, in terms of efficiency the application did not excel in any of our test cases. The highest efficiency value was 0.92 for the third test case with 2 threads. In all situations efficiency decreased significantly with the increase in the number of threads, mainly because of the serial portion of write I/O.

As expected, the performance for data recovery was similar to the hiding part, since they exhibit a similar structure. The second test case leads to the best results with a speedup of 3.92 for 12 threads. For the smallest data set the parallel data recovery was slower than the serial version and for the largest image we obtained a speedup of 2.39 for 10 threads.

There are a couple of factos that must be taken into consideration before running the application. The first, and the most important one is the size of the cover image compared to the dimensions of the message that must be hidden. In case more than three bits must be used from each color component, there is a high probability of the message's presence to be detected, thus defeating the entire purpose of steganography. Once a suitable cover image is found one can choose between using the serial implementation or a parallel one. In case the parallel version is ran, another factor that must be taken into consideration is the number of threads used – using 10 worker threads is not always 5 times more efficient than using 2. Last but not least, the type of file system can also be taken into account. For a parallel system the second parallelization method is definitely the best match; however, serial writing in the same output file may also be suitable, depending on the size of the resulting output file.

In conclusion, the main objectives of this project were accomplished. The result was an application that can be used to encode into, and decode data from images. The serial execution time was reduced to a quarter. The main reason speedup did not reach higher values than 4 is the increase in I/O operations. Data processing is not very time consuming, as it is reduced to bit operations, which are very fast.

VIII. Future Work

One of the first improvements that can be brought to the application is its adaptation to a parallel file system. In this way each thread could write its chunk of the image directly to the output file, without needing a bash script to concatenate pieces of files. I/O operations are currently the biggest bottleneck for applications dealing with large input or output data flows.

The application can be further enhanced by designing algorithms that work with different files types, not only bmp images. Audio and video files are also great stego covers. In addition to this, the range of input files to be hidden can also diversify, containing other text and image image formats and even audio or video.

In order to better hide the data in case the message is too small compared to the input image, more complex algorithms can be used. Hidden bits can be dispersed throughout all the image using a unique key that only the sender and the receiver posses.

While simple to implement, the LSB hiding method is quite easy to detect. In order for the output stego images to pass steganalysis tests, the application could embed data only in certain regions of the image. Experimentaly it has been proven that data hiding into skin tone areas is less likely to be detected. The reason this happens is the fact that these regions produce typically less distortion than other image areas, as has been shown in [6].

One of the main advantages that steganography has over cryptography is the fact that the message is invisible to someone that intercepts data exchanges between two persons. However, if an image is suspected of containing hidden data, there are many steganalysis tools that could easily decode the content of the secret message. Most methods used for this purpose are based on statistical analysis, mainly used for noise detection. So, once a stego message is detected its whole content is revealed. Thus by using both steganography an cryptography one could create a powerful tool for data hiding - more powerful than either technique, if applied on itself.

Acknowledgments

This research is partially supported by the HP-SEE EU-FP7 Project Nr. 261499/2010.

References

[1] B. Granthan, "Bitmap steganography: An intoduction," April 1997.
[2] N. F. Johnson and S. Jajodia, "Exploring steganography: Seeing the unseen," April 2007.
[3] N. Provos and P. Honeyman, "Hide and seek: An introduction to steganography," January 2004.
[4] T. Morkel, J. H. P. Eloff, and M. S. Olivier, "An overview of image steganography," in *Proceedings of the Fifth Annual Information Security South Africa Conference ISSA2005*, L. L. Hein S Venter, Jan H P Eloff and M. M. Eloff, Eds., Sandton, South Africa, June/July 2005, published electronically.
[5] J. Krenn, "Steganography and steganalysis," January 2004.
[6] A. Cheddad, "Steganoflage: A new image steganography algorithm," Ph.D. dissertation, University of Ulster, September 2009.
[7] J. P. F. Bradford Nichols, Dick Buttlar, *Pthreads Programming: A POSIX Standard for Better Multiprocessing*. O'Reilly Media, 1996.
[8] D. R. Butenhof, *Programming with POSIX Threads*. Addison-Wesley Professional, 1997.
[9] M. D. Scott J. Norton, *Thread Time: The MultiThreaded Programming Guide*. Prentice Hall PTR, 1996.