Parallel GNU Optical Character Recognition

Laura Vasiliu
Parallel and Distributed Systems Master
The Faculty of Automatic Control and Computers
Bucharest, Romania
Email: lauravasiliu@ymail.com

Cristiana Voicu
Parallel and Distributed Systems Master
The Faculty of Automatic Control and Computers
Bucharest, Romania
Email: voicucristiana@yahoo.com

***Abstract*—The idea of Optical Character Recognition, known as OCR, was introduced for the main purpose of having the information in a digitized format, to easily access or search for the data. To achieve this, powerful and accurate algorithms are needed for character recognition. Two main issues in this field are the text line and word segmentation that influence the accuracy and the speed of processing. Parallel GOCR is meant to bring the original GOCR to better performance results in terms of processing time by running in parallel different parts of the tool. All the pipeline steps of characters processing will be done each in parallel.**

***Index Terms*—OCR, parallel programming, characters detection algorithms**

I. Introduction

Document image processing has been an important subject for the research area in the field of human machine interface for the last few decades. Optical character recognition, usually abbreviated to OCR, is the conversion of scanned images of handwritten or printed text into digitized text. One of its purposes it to make use of the data that is still not in a machine-encoded format and serve as a form of data entry from some sort of original paper data source. By having the information digitized , data can be easily electronically searched, stored more compactly and used in machine processes such as machine translation or text mining. OCR plays an important role for the research field in pattern recognition, artificial intelligence and computer vision.

The need for efficient and robust algorithms and systems for recognition is being felt around the world. For example, the postal department needs recognition for sorting the mail. Another common usage is the preserving of out-of-print old books by digitising them. Character recognition can also form a part in applications like intelligent scanning machines, text to speech converters or automatic language-to-language translators.

Fast and accurate algorithms are necessary for OCR systems to perform operations on document images such as preprocessing, segmentation, feature extraction and post processing. Text line and word segmentation are two main steps in any OCR system. Having the wrong segmentation may reduce the accuracy rate of OCR systems. The segmentation is very challenging in cases of different types of noises, degradations, and variation in writing and script characteristics. However, existing algorithms suffer from a flawed tradeoff between accuracy and speed.

In this paper, we introduce a modified tool, Parallel GOCR, based on the original GOCR. The original tool is an OCR (Optical Character Recognition) program, developed under the GNU Public License. It converts scanned images of text back to text files. GOCR [2] can be used with different frontends, which makes it very easy to port to different OSes and architectures.

Parallel GOCR is meant to significantly reduce the processing time by running several parts of the tool in parallel. It intends to make use of shared memory to ensure communication between multiple threads.

This paper is structured as followed. In section 2 we review the attempts to develop a parallel optical character recognition project. Section 3 describes the architecture of an OCR project and the changes needed to run in parallel. Section 4 describes a parallel OCR implementation, with all the functions we have changed. Section 5 presents some test scenarios and the results for each one, results that will let us conclude some ideas shown in section 6.

II. Related work

The first project that is related to our tool is Tesseract. This application is an accurate open source OCR engine. Combined with the Leptonica Image Processing Library it can read from a wide range of image formats and convert them to text in over 60 languages. Between 1995 and 2006 it had little work done on it, but since then it has been improved extensively by Google. It is released under the Apache License 2.0.

Until version 3.0 of Tesseract, this tool did not use any parallel model. In order to achieve better performance, an API instance of static methods was used. It was possible to use two different APIs alternately in the same thread, but using them concurrently in separate threads was not a good decision.

The main issues of the tool were the critical global variables that had to be changed, or the memory manager that needed protection from a mutex, or made thread-local. Either of these changes would be affected by portability issues.

From version 3.0, Tesseract introduced into the application the thread-safety concept. This way, multiple instances can be

used in parallel, being processed by multiple threads, with the minor exception that some control parameters are still global and affect all threads. To supply a proper environment for thread-safeness, all critical globals and statics were moved to members of the appropriate class.

Another project in the OCR field is DRISHTI OCR, having a multi threaded architecture [3]. DRISHTI stands for Document Recognition and Imaging Software for Handling Telugu as ISCII. It is a project developed by Lalita Nayal and Rakesh Kumar Gupta, two students at University of Hyderabad. The main purpose of the DRISHTI project was to improve the computational speed of the OCR.

In order to achieve this, the communication is based on the shared memory paradigm. Each thread could either access a whole line at a time or one word at a time. The main problem for the DRISHTI project was to synchronize the threads so that they could have a correct output. For this purpose, they used mutexes as a mean of synchronization.

The third related project, is Devanagari parallel OCR system tool [1]. Devanagari text line and word segmentation are carried out using modified standard profiling based segmentation approach and parallelized it on Graphics Processing Unit (GPU).

The project goal is to make segmentation faster for processing a large number of document images using parallel implementation of algorithms on GPU. Due to reduced costs in comparison to powerful parallel systems, GPUs have been chosen to match the design of the application. The approach employs extensive usage of highly multithreaded architecture and shared memory of multicore GPU.

An efficient use of shared memory is required to optimize parallel reduction in Compute Unified Device Architecture (CUDA). Experimental results show an achieved speedup of about 20x-30x over the serial implementation when running on a GPU named GeForce 9500 GT having 32 cores.

III. Architecture

As shown in Fig. 1, GOCR system starts with the optical scanning phase or getting the raw RGB data that is going to be digitized.

As input, the GOCR supports the following file formats: PNM, PBM, PGM, PPM, PCX (some), TGA. Other formats are automatically converted using netpbm-progs, gzip and bzip2 via the use of a unix pipe. GOCR creates a new job for every image. The raw data is then converted from RGB format into gray-level format which means that red text on green background will not be recognized. The gray-level text image is filtered to remove noise and dust.

The application does not need to train the program or store large font bases.

As the process of recognition is conducted character by character, the next step is to segment the text image into character lines and individual character blocks through horizontal and vertical projection. For this stage, each character will be

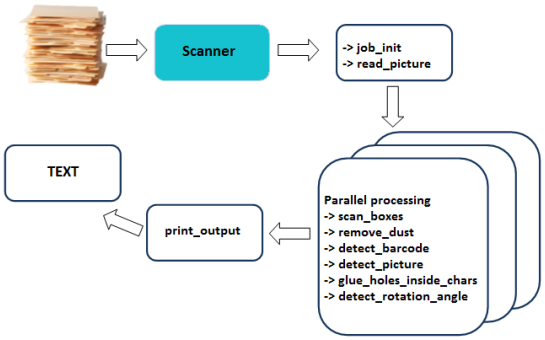

Figure 1. Parallel GOCR Design

put inside a box and processed. There are several steps in the process of recognizing the characters:

A. Segmentation of textual regions

The first step in recognizing characters is the segmentation of textual regions. This process is recursively divided in two parts. It searches the thickest horizontal or vertical gap through the box. In case the gap is less than five times the thickest gap no division will be performed. Otherwise, the process will be repeated with the newly two parts obtained from the previous division.

B. Line detection

After identifying the textual regions, lines of characters are detected by looking for interline spaces. These are characterized by a large number of non-black pixels in a row. The line detection plays an important role for obtaining good accuracy in character recognition. For example, it is difficult to distinguish between lowercase letter p and uppercase letter P without having a baseline (same total height). The lowercase version of p has a depth (the lower end is below the baseline) and therefore it's easy to distinguish from the uppercase version if the baseline is known. The line detection must find the baseline of every text line.

Image rotation (skewing) represents a problem, therefore the program first looks only at the left half of the image. When a line is found, the left half of the right side is scanned, because lines are often short. The variation in height gives an indication of the rotation angle. Using this angle, a second run detects lines more accurately. The profiling results of the original OCR revealed that the function that implements this functionality (detect_rotation_angle) spends the most processing time from the whole application. To achieve a better performance, we used a multi threaded environment by using shared memory as form of communication between threads. We used OpenMP [4]tasks as API for having a parallel processing.

C. Cluster detection

A cluster represents a group of pixels connected with each other. For detecting the cluster the algorithm for leaving a

maze was used.

Up to this point, the text information is localized in standalone character blocks, in RAM. GOCR is able to work with different recognition engines. Having this big advantage, GOCR has the possibility to compare results of different engines or, in case of a not recognized character, to inform the user which characters probably could be there. The base engine is the original engine used in the first implementation of GOCR.

The database engine was the second engine added to GOCR. The main algorithm compares not recognized characters with stored images and calculates a distance value. If the distance value is small enough, the character is treated as recognized.

The function that glues the holes is also CPU consuming. We have followed the same approach to parallel the processing, using OpenMp [5] tasks.

IV. OCR ISSUES

There are some cases when the recognition of the letters is difficult to implement because of some extra or missing pixels from the letter. For example, let's take the n letter from Fig. 2, that has some additional pixels at the bottom, which makes it difficult to detect. The proposed idea contains three main steps. At first, the horizontal and vertical pixels are marked with "=" for the horizontal ones and "I" for the vertical ones. The decision to mark one pixel with one of the symbols is taken based on the distance between the next horizontal and next vertical white pixels, that there are marked with "." .

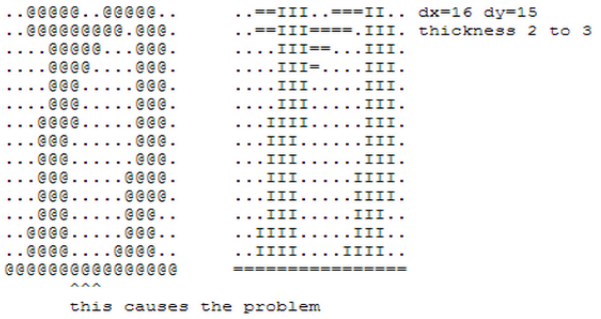

Figure 2. Example of picture with extra pixels

Secondly, the mean thickness of vertical and horizontal clusters is measured so that we can identify the main characteristics/lines of the letter. In the last step, we erase the unnecessary pixels and in the end we obtain the clear letter.

Previously, we mentioned that are cases when we need to add pixels in order to detect the character. In Fig. 3 we encounter these cases. Here we have the "m" letter but the legs are not glued. In case the engine fails recognizing, a filter is switched on and the engine starts again. The purpose of the filter is to add "O" pixels to connect the pixels that are slightly apart from one another.

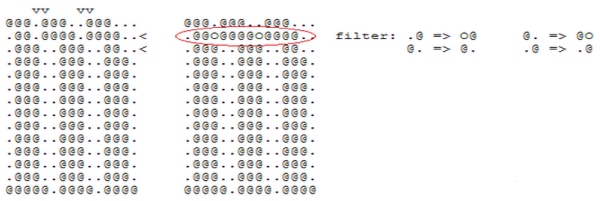

Figure 3. Adding pixels to determine the character

Not only the missing or the extra pixels represent a problem, but also the overlapping characters. In Fig. 4 we have a good example of two letters, "r" and "u", that overlap. For the first one, it is difficult to distinguish between them and we have to look for the weak connections. A partition of the entire cluster into boxes is made. Then, a test with the engine is made on each of the bounded boxes to try to recognize the character inside the box. In the end, a correction is made for the surrounding box of each character.

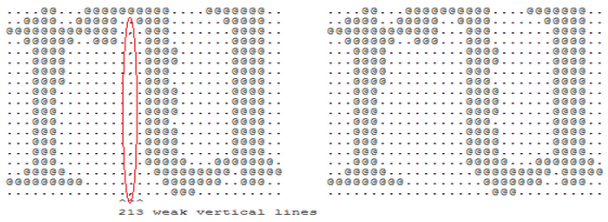

Figure 4. Overllaping characters

Another important issue for recognizing the letter is the misleading dust on paper represented by the "blind pixels", as can be observed in Fig. 5. These pixels are not connected with the character and are removed by using a fill-algorithm.

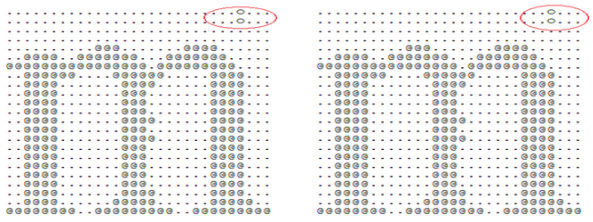

Figure 5. Removing dust

The noise is another problem for detecting the pixels. To eliminate noise, characters are identified by comparing them with others, already detected through a distance function.

To simplify the character recognition, the colored images are converted into gray ones when processing. In this case, the application will not be able to detect the green text on red background.

V. Implementation

Before starting to analyze our parallel approach, we profiled our application with the Solaris Studio profiling tool from Oracle. This way, we were able to discover the parts were GOCR was spending most of the time.

In our steps of implementing the parallel version of the GOCR we encountered several constraints. The first one was related to the limitation of the picture size. As designed, the GOCR keeps in memory the entire image while processing. This fact limits the number of images processed in parallel. Another important issue that we had constantly taken into consideration in our parallel implementation was not to lose accuracy. If we would have just divided the image in chunks processed by different threads, we would have had broken rows, broken letters and, as result, a great loss in accuracy.

Moreover, the steps for detecting the characters are in pipeline. This restricted us from processing different steps in parallel. So, as the serial version was designed, it narrowed our parallel design.

Our first approach regarding the implementation of the parallel version of the GOCR project was to use OpenMP 'for' directives. The 'for' directive splits the for-loop so that each thread in the current team handles a different portion of the loop. After we have investigated more the code, we understood that every cluster(the element representing the image for a letter), is represented in code as an element in a list. For this purpose, the serial version of GOCR has two files: list.h and list.c, where you can find the implementation of some macros and methods used to manipulate a list. The macro that influenced us in implementing the parallel version of GOCR is the following:

```
#define for_each_data(l)
 if (list_higher_level(l) == 0) {
  for ( ; (l)->current[(l)->level]
      && (l)->current[(l)->level]!=&(l)->stop; (l)->current[(l)->level] =
       (l)->current[(l)->level]->next ) {
```

Figure 6. Iterating the list with all the characters from the image

The second approach proved to be the most appropriate. Given the fact that the all information about the clusters from images are saved in a list, we conclude that we have to use OpenMp tasks, as can be noticed in Fig. 7. The task pragma can be useful for parallelizing irregular algorithms such as pointer chasing or recursive algorithms for which other OpenMP workshare constructs are inadequate.

We parallelized a few parts of the code because if we would have gone further we would have lost accuracy. We made an improvement into the function that detects the rotation angle of the text. We parallelized also inside the function that detected glue boxes inside another box. The small glue boxes represent extra pixels that don't belong to the initial box and have to be eliminated. The third function to which we improved the performance is the one that compared the chars that couldn't be detected from the first run and had to be compared with

```
#pragma omp parallel
{
List *l = &(job->res.boxlist);
#pragma omp single private(l)
{
for_each_data(l) {
  #pragma omp task
  {
  box2 = (struct box *)list_get_current(l);//&(job->res.boxlist));
  .
  .
  .
  . // a lot of mathematical calculations
  }
} end_for_each(l);//&(job->res.boxlist));
}//end of single..
}//end of parallel
```

Figure 7. Code snippet of parallel section

the one already found. All these three functions are CPU intensive because they make several searched in the structure that contains all the letters received as input.

VI. Experimental Setup and Results

A. Performance profiling

This section shows the profiling results for both serial (Fig. 8) and parallel (Fig. 9) approach and refinements made in light of this analysis using the Sun Studio profiling tool.

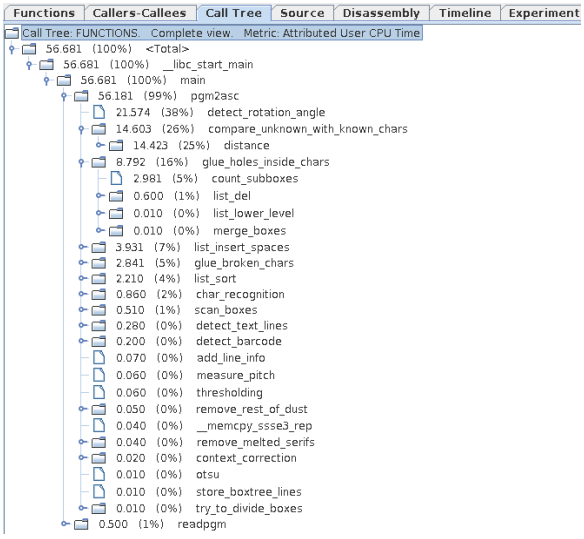

Figure 8. Profiling results of the serial GOCR

B. Results

We run our benchmark, once, on an environment on different scenarios with 2, 4, 8, 16 and 32 different Nehalem cores on the same node from the fep.grid.pub.ro cluster.

As set of data, we used a 15 MB ppm file with text and no figures. From the results pictures we can notice that the parallel GOCR obtains a good speedup for 8 threads, being 3.2. Moreover, we obtained 95 percent accuracy on the parallel version in comparison with the serial implementation.

As you can observe in Fig. 11, the speedup when the application is running on 2 threads is 1.9. This means that

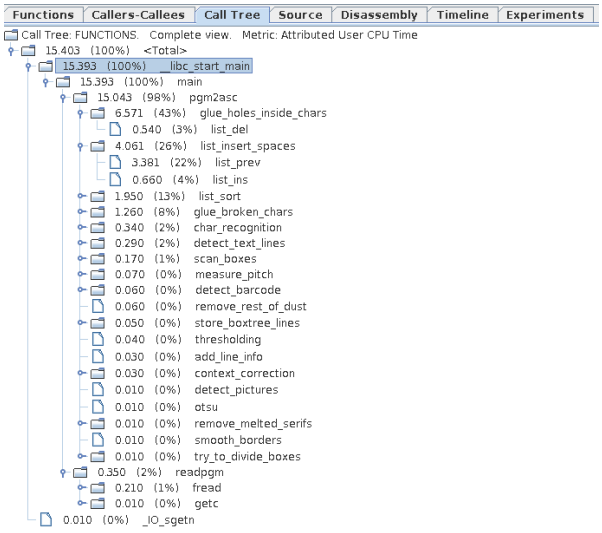

Figure 9. Profiling results of the parallel GOCR running on 8 threads

we have reached our goal, we have parallelized the part from the serial project that is taking the most important time. Also, another conclusion that can be drawn from this graphic, is that the I/O part doesn't take so much time, related to the time necessary for character recognition.

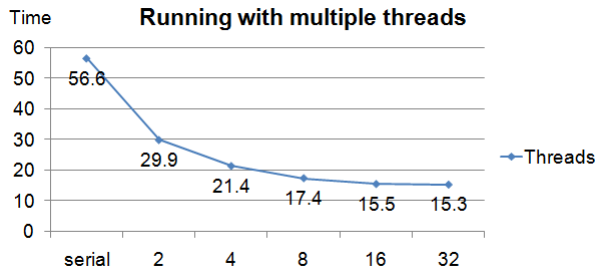

Figure 10. Results of running the application on multiple number of threads

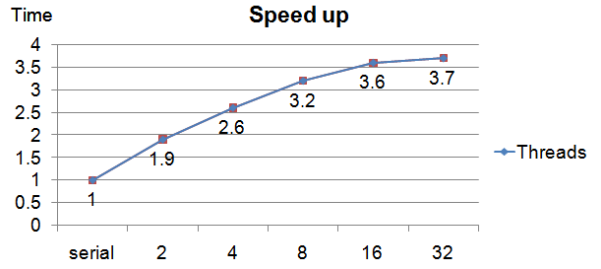

Figure 11. Speedup obtained from running the application on multiple number of threads

VII. Future work

Even if the code has been developed to the point where it can recognise well formed printed text with a high degree of accuracy, a real step forward could be developing a commercially product. We propose for future development the following extensions:

1. Line segmentation

As a feature, we propose designing a new line segmentation algorithm that could be able to handle correctly pages of text with multiple columns and poorly aligned text. We might have the case of important data from newspapers. Because these issues could easily affect the viability of the parallel approach, we consider that they are highly important and should be addressed before other extensions.

2. Extending the characters that can be recognized

This is an important request since the tool should be available on a wide range of languages. This feature would require quite little effort for implementing.

3. Improve performance on lower quality documents.

The tool should address different input qualities. From high resolution scanned documents to lower quality such as faxes and photocopied text.

So far, we talked about improving the recognition phase of the application. Once these issues are solved , there are a range of techniques available for improving the accuracy: an improved character segmentation algorithm, the use of contextual information and more sophisticated handling of multiple fonts - the classifier could remember the current font.

VIII. Conclusions

In this paper, we have briefly described the experience made by parallelising a sequential OCR application by using OpenMP paradigms. The parallel OCR project which has been implemented has demonstrated a good scalability when running on the Politehnica University Fep cluster and on personal computers, as well. Furthermore, the scalability can be easily improved as more sophisticated computational techniques are employed.

The results indicate that a parallel approach on a workstation cluster offers a promising means of achieving high performance OCR.

References

[1] Devanagari parallel ocr system. http://www.ijcaonline.org/volume24/number9/pxc3873988.pdf.
[2] Gocr. http://jocr.sourceforge.net/.
[3] Multi threading drishti ocr. http://dcis.uohyd.ernet.in/ chakcs/ProjectReport.odt.
[4] Openmp 3.1 specification released. http://openmp.org/wp/2011/07/openmp-31-specification-released.
[5] Openmp compilers. http://openmp.org/wp/openmp-compilers.