978-1-5090-3525-0/16/$31.00 ©2016 IEEE

Performance Characterization and Parallelization of Tesseract Optical Character Recognition on Multicore Architectures

Sunghwan Bae Jialing Zhang Seung Woo Son
University of Massachusetts Lowell

1. Introduction

Optical Character Recognition, or OCR, is one of the major topics in computer vision technology. It is widely used in various applications, such as a digital libraries, automatic banking systems, and mailing services. Tesseract OCR Engine, which we evaluate in this paper, is one of renowned OCR programs. It was originally developed by Hewlett Packard Lab between 1985 and 1995, and has been maintained by Google since 2006 [1]. Initially, this program was designed to recognize English text only, however, it has been enhanced to support other languages as more training models were added [2].

OCR process including Tesseract is known to be very compute intensive because of the computation involving image and mathematical processing to obtain higher recognition accuracy. While there has been a significant improvement in the recognition accuracy of Tesseract OCR, parallelization has not been extensively studied. Also, there is a plethora of multicore architectures, thus it is highly considered to achieve better performance by utilizing those parallel architectures. In this paper, performance characterization has been performed using a profiling tool to find a target, and then, parallelizing the identified target using an appropriate parallel programming method.

The main goal of this paper is characterizing the performance of the Tesseract OCR program and accelerating compute-intensive loops on multicore architectures. Our main contributions are as follows. First, we analyze the Tesseract OCR program using a binary instrumentation tool and identify target loops that can be parallelized. This allows us to decide which parallelization method needs to be applied and how to modify the loops in order to make them run in parallel. Second, we apply parallel methods on the loops based on the characterization analysis. Lastly, we discuss issues and limitations in parallelizing the Tesseract OCR and suggest appropriate solutions as needed.

2. Our Approach

Performance Characterization. Tesseract OCR has multiple steps to recognize characters from an image. The first one is line and word finding [3]. It finds text lines by analyzing page layout and presumed text size. Even if the text line is curved or slanted, Tesseract OCR is able to recognize the character by Baseline Fitting [3]. Next is word recognition. It has two steps; one is recognizing characters from a word, and the other is a process for improving accuracy called an “adaptive classifier” [3]. This process is a potential candidate for parallelization because it is executed iteratively when a document image has many words.

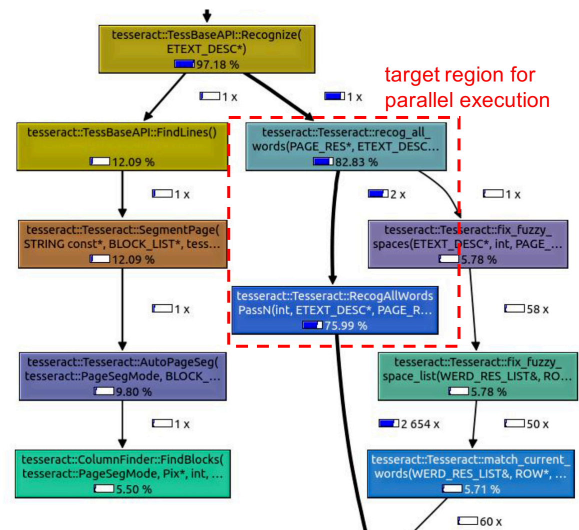

Figure 1. Part of call graph of KCachegrind based on Callgrind result file.

To characterize the performance of Tesseract in detail, we used the Valgrind instrumentation framework, specifically a profiling tool called Callgrind. Callgrind profiles the program and collects the function call history, the number of calls, and relationships between functions during runtime [4]. The collected information can be visualized using the KCachegrind tool after generating the report file by specifying `--tool=callgrind` on the Valgrind command line.

Figure 1 is a part of the call graph that shows an iterative function call. As shown in the figure, `RecogAllWordsPassN()` repeatedly calls `classify_word_and_language()` 2,654 times. Thus `RecogAllWordsPassN()` can be parallelized if there is no dependency. In terms of the time contribution, it accounts for 75.99% of total elapsed time, so significant performance improvement can be expected if this module was successfully parallelized.

Parallelization Strategy. According to our performance characterization and call graph analysis, the potential can-

978-1-5090-3525-0/16/$31.00 ©2016 IEEE

didate for parallelization is `RecogAllWordsPassN()`. Based on our performance characterization, we found that OpenMP API can be an optimal option for this program as it allows users to use multiple threads to perform multiple iteration process concurrently and to use shared memory which can be accessed by those threads.

There are 9 steps called "Pass" in this program to recognize characters after an image segmentation process divides one or more input images into word blocks. The candidate function to be parallelized has the first two steps implemented on itself, which are the processes of recognizing characters from the word block images. The other steps are processes of correcting results from previous steps to improve the accuracy of the result.

Since the target function is designed to support two passes, Pass 1 and Pass 2, several issues need to be addressed before parallelization. First, although the function has only one loop, each iteration involves many steps, such as updating monitoring progress, changing language setting in case of failure, and fuzzy correction which could be useful in the second phase. The second issue is the serialized class object to store recognition results. Whenever a process of a single word block is completed, it calls a function of the class object to shift its pointer of memory to store next result. The last one is dependencies between each word recognition process. A separate memory data structure that stores a word has a pointer member pointing to the previous word which can be used to change character recognition settings dynamically based on previous result. However, this can be an obstacle for parallelization.

To avoid these problems, we separate the target routine into two separate routines. One reason is that the code in Pass 2 is relatively complex because it needs to take care of previous result. Another reason is the dependency of each word recognition process. Although the result of previous word recognition is not used in Pass 1, it can be used in Pass 2 if Pass 2 is executed in serial. The last reason is that it is much easier to handle the pointer to PAGE_RES_IT, which is a class object for storing results from Pass 1.

3. Experimental Evaluation

To evaluate the impact of our parallelization methods, we built and evaluated Tesseract on the Massachusetts Green High Performance Computing Center (MGHPCC) [5]. To build the Tesseract OCR, we used the following modules: gcc/4.7.4 and multiple image libraries (libtiff/4.0.3, libjpeg/6b, jpeg/9a, and giflib/5.1.0). Tesseract also requires Leptonica [6], which is a widely used library for image processing application in order to support diverse image formats.

We measured the performance as well as the recognition accuracy of our parallelized Tesseract OCR using a sample image which had 1,141 words. The accuracy of character recognition from original Tesseract OCR was 98.86% which was calculated using the String Similarity Tool [7]. Table 1 is the performance breakdown of the parallelized Tesseract. It clearly shows that the parallelized target routine, `RecogAllWordsPassN()`, is successfully parallelized up to 16 threads. It also shows that as the number of threads increases, the performance improvement is saturated because of the critical section among the passes. Overall, the total elapsed time has been improved up to 33% as a result of OpenMP parallelization. In terms of the recognition accuracy, we were able to maintain the accuracy of 98.86% regardless of the number of threads.

TABLE 1. PERFORMANCE BREAKDOWN OF THE PARALLELIZED TESSERACT WHILE INCREASING THE NUMBER OF THREADS.

Number of threads	Elapsed Time (sec)		Recognition Accuracy
	RecogAllWordsPassN() only	Total	
1	5.41	13.81	98.96
2	2.55	11.18	98.96
4	1.17	9.93	98.96
8	0.58	9.56	98.96
16	0.32	9.30	98.96

4. Conclusion and Future Work

The Tesseract OCR engine supports many different languages and has a great accuracy of character recognition, but its performance on multicore processors has not been studied exhaustively. The compute-intensive loops, identified by using the Valgrind profiling tool, are intrinsically serial routines that update the recognized characters while iterating multiple passes. It also has dependencies where the current word recognition phase uses the result from the previous recognition phase. In addition to this, there is a memory collision issue. In this paper, we reduced these dependencies, thereby allowing us to apply parallelization on compute-intensive loops using OpenMP. Our experimental evaluation using a document image containing 1,142 words demonstrates a scalable speedup for the target loop up to 16 threads. Therefore, the overall performance has been improved up to 33% as compared to the serial version.

In our future work, we plan to improve the performance further by resolving the memory collision between Pass 1 and 2 as it prevents threads among different passes from simultaneously running. We also plan to accelerate the target kernels using other parallelization mechanisms, such as CUDA and OpenCL.

References

[1] L. Vincent, "Announcing Tesseract OCR." [Online]. Available: http://googlecode.blogspot.com/2006/08/announcing-tesseract-ocr.html

[2] A. Dovev, "Training Tesseract." [Online]. Available: https://github.com/tesseract-ocr/tesseract/wiki/TrainingTesseract

[3] R. Smith, "An Overview of the Tesseract OCR Engine," in *Proceedings of the Ninth International Conference on Document Analysis and Recognition - Volume 02*, ser. ICDAR '07, 2007, pp. 629–633.

[4] J. Seward, N. Nethercote, and J. Weidendorfer, *Valgrind 3.3 - Advanced Debugging and Profiling for GNU/Linux Applications*. Network Theory Ltd., 2008.

[5] "Massachusetts Green High Performance Computing Cluster." [Online]. Available: http://wiki.umassrc.org/wiki/index.php/Main_Page

[6] H. Gehrke, "OCR - Optical Character Recognition." [Online]. Available: https://help.ubuntu.com/community/OCR

[7] "String similarity test." [Online]. Available: https://www.tools4noobs.com/online_tools/string_similarity/