Speed Up of Image Filtering using Parallel Programming Models

José Agustín Barrachina
IEEE Student Member
École Polytechnique
Université Paris-Saclay
Route de Saclay, 91128 Palaiseau, France
Email: joseagustin.barra@gmail.com

François Bidet
École Polytechnique
Université Paris-Saclay
Route de Saclay, 91128 Palaiseau, France
Email: francois.bidet@polytechnique.edu

***Abstract*—The aim of this project was to speed-up a target application through multiple parallelism models (MPI, OpenMP and CUDA) for various architectures (CPU and GPU) and conclude which approach is the most suitable for the specific application.**

1. Introduction

A base C/C++ code that implements an image filtering was given. The aim of this work is to optimize the operation time of the code by using parallel programming methods such as MPI, OpenMP and CUDA. The base C/C++ code covers a large spectrum of algorithms in HPC and Big Data. The Image Filtering can be seen as a direct application of Big Data programs. This application uses a stencil-based scheme to apply a filter to an existing image or set of images. It means the main process of this code is a traversal of each pixel of the image and the application of a 2D stencil [**?**].

1.1. Project Organization

This project was done using github version control repository. Each different strategy of parallelizing was made in a different branch, having as master branch the original serial program. So that in the end a merge was done with the strategies that proved to work better.

The different branches are as follows:

- **Master** Contains the file with the sequential code.
- **Result** Contains the final realization with a combination of CUDA, MPI and OpenMP.
- **MPI**$_{over\_images\_of\_gif}$ Applies MPI to treat each image of a gif separately.
- **Bidouille** Applies MPI to divide the work on each gif.
- **OpenMP**$_{divide\_image}$ Applies OpenMP to divide the work on each image.

1.2. Image Format

We will work with GIF image because this formate has the following advantages:

- Easy to manipulate
- Widespread format (social networks)
- Allow animation (multiple images)

1.3. Testing Work

To test the efficiency of the program, many files were created. A simple C program was done to make a summary of all the results obtained. This summary is only an addition of the time taken to apply the filter to each gif image.

Also, a shell file was created to run many gif images one after the other. Various variations of this shell were made to run the test only on gif files that share some property as for example, containing only one image or on the contrary containing more than one image.

With the help of ImageMagik [**?**]. Two shell files were created to compare the original (serial) program output with the new parallel optimized program in order to check that it works perfectly. The first bash file creates a output image that, if both images are exactly the same, should create a complete black image. Because this method can be difficult to be sure the images are exactly the same, a second bash file was created that compares pixel by pixel each image and output 0 if the image is completely identical.

2. Image Filtering Program

This topic focuses on a specific image filter that is useful to detect objects and edges inside various images. This filter is called Sobel and it applies on a greyscale image. Therefore, the first step is to transform the image from a color one to a set of gray pixels.

Then, a blur filter is applied to a small part of the images to exclude the borders. Thus, the main goal is to parallelize an application that apply multiple filters (grayscale, blur and Sobel) to a GIF Image (either a single image or an animated gif).

2.1. Structure

The algorithm can be divided into three main parts:

1. GIF import (section **??**)
2. SOBEL filter (section **??**)
3. Export image (section **??**)

A main function already measures and output the time taken by each of the three listed parts of the project.

2.1.1. GIF import. This part just load the gif file into a pixel array.

2.1.2. SOBEL filter. This part of the code applies actually three different filters:

1· **Gray Filter** Convert the pixels into a grayscale
2· **Blur Filter** Apply blur filter with convergence value
3· **Sobel Filter** Apply Sobel filter on pixels

2.1.3. Export image. This is just the inverse of the first part (section **??**). Converts the array of pixels and creates a gif file which is then stored into a folder.

3. Parallel Optimization Languages

3.1. MPI

MPI stands for Message Passing Interface, is a standardized and portable message-passing system to function on a wide variety of parallel computing architectures. MPI takes each worker as a process and standardizes the message passing operation between them.

3.1.1. Treating each gif image separately. Each gif image can have several images saved together, the core of this optimization method was to make each worker to treat each image separately in order to make the process faster. Two different ways of organizing the work distribution can be applied:

1) Static Work Distribution
2) Dynamic Work Distribution

In the first case, the Static Work Distribution just gives each worker a different but precomputed image. For example, should there be "N" images and "P" processes, we can make a rule that follows the following equation:

$$task \equiv n \mod P \tag{1}$$

Where task will be the number of the process that must work on the image number *n*. Following this rule, should we have 7 images and 7 processes, the distribution will be as follows:

It was chosen to make a Static no only because it was an easier implementation, but also because gif images are supposed to have all images similar to each other. Supposing each thread is run by a equivalent hardware (which is our case), the time taken by each process to filter each image will be almost the same, and using a dynamic distribution won't be of much help. On the contrary, working in dynamic mode will require a thread to control the main program a distribute the work, which will make the program to virtually loose one thread in order to change the distribution which as stated won't be of much use. Of course, this approach will not be so should each image of the gif file be very different to each other.

3.1.2. Treating subparts of each image. A master task splits the input images in subpart and sends each subpart to a slave task, which compute the blur filter. After the work, the master task merges each part of the original image and finishes the blur filter iteration.

This work can be done as long as the difference between the input image and the blured image is bigger than a predefined threshold.

The biggest problem of this solution is to split the input image in subparts with a overlap area.

We get the best reduction of time with 8 tasks on the same node than any number of tasks on many node because communications between nodes are much slower than in the same node.

3.2. OpenMP

OpenMP stands for Open Multi-Processing. It is an application programming interface (API) that supports milti-platform shared memory multiprocessing programming in C, C++ and Fortran on most platforms, processor architectures and operating systems. OpenMP provides a portable, scalable model for developers of shared memory parallel applications. It consist of a set of three primary API components:

1· Compiler Directives
2· Runtime Library Routines
3· Environmental Variables

OpenMP comes included in gcc compiler so that only a flag is needed for gcc to compile the Parallel code made with OpenMP.

3.2.1. Treating subparts of each image. As in **??**, the goal of this part was to implement a program that divides the image into sections and each thread treats each section. The approach was different than on the MPI implementation. Here, it was not the main which was treated but every specific filter function.

3.2.2. Results. The results were quite effective. Running the algorithm through many different gif images, the Blur Filter Time was reduced by around 75% . The Sobel Filter Time was reduced around 60% and the Gray Filter was reduced a 30%. This results are taken by adding all the different times together and comparing both results of the original program and the OpenMP implementation.

3.3. CUDA

4. Conclusion

Acknowledgments

References

[1] P. C. . F. Trahay. (2016) Inf560 evaluation. [Online]. Available: https://www.enseignement.polytechnique.fr/profs/informatique/Francois.Trahay/www/TD/projects/INF560-projects-0.html

[2] I. S. LLC. (1999) Convert, edit, or compose bitmap images. image magick. [Online]. Available: https://www.imagemagick.org/script/index.php