Speed Up of Image Filtering using Parallel Programming Models

José Agustín Barrachina
IEEE Student Member
Instituto Tecnológico de Buenos Aires (ITBA)
Av Eduardo Madero 399, Buenos Aires, Argentina
Email: joseagustin.barra@gmail.com

François Bidet
École Polytechnique
Universit 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 [**?**].

2. Image Filtering Program

In this section the general C/C++ algorithm will be explained. A shell script was created in order to compare the program efficiency against numerous gif files with different size and properties like number of images or colors.

2.1. Description

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. For this purpose, we will work with GIF image because this formate has the following advantages:

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

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). [?]

2.2. Structure

The algorithm can be divided into three main parts:

1· GIF import (section 2.2.1)
2· SOBEL filter (section 2.2.2)
3· Export image (section 2.2.3)

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

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

2.2.2. 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

2.2.3. Export image. This is just the inverse of the first part (section 2.2.1). 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:

3.2. OpenMP

3.3. CUDA

4. Conclusion

Acknowledgments