Parallel Implementation of Image Filters

1st Hamed Vaheb
Faculty of Science, Technology and Medicine (FSTM)
University of Luxembourg
Esch-sur-Alzette, Luxembourg
hamed.vaheb.001@student.uni.lu

2nd Elnaz Khaveh
Faculty of Science, Technology and Medicine (FSTM)
University of Luxembourg
Esch-sur-Alzette, Luxembourg
elnaz.khaveh.001@student.uni.lu

***Abstract*—This report is for the project of parallel implementation of image processing algorithms, i.e., grayscale, blur, edge detection, and reflect on bitmap formatted images. All filters were parallelized using the OpenMP application programming interface (API). The implementations were performed on high performance computing (HPC) clusters. The results show that there was a minor improvement in runtime of the parallel implementation, when we set number of threads as two. Using higher number of threads didn't have an improving effect on the runtime. In order to report a scrutinized analysis and visualization of the code's performance, especially the effect of parallelism, the Vtune Profiler software from the Intel oneAPI Base Toolkit is used. The results show that using two threads with static scheduling led to slight improvement of the program, yet since the complexity of the filter functions used were not high enough even on large images, the effect of parallelism was not clearly evident.**

***Index Terms*—Image processing, Parallel processing, Multiprocessors, OpenMP**

Code

I. Introduction

One of the most prominent innovations in computer engineering has been development of multicore processors, which are composed of two or more independent cores in a single physical package [1]. Inspired by the demand for higher performance in the contemporary digital world, various types of processors, e.g., mobile, graphic, central processing units (CPUs), digital signal processors (DSPs), etc have a multicore design.

In the Past 50 years, the semiconductor industry has followed Moore's law [2]; however, there has been slow progress since 2010, which is mainly due to maximum transistor capacity, heating and cooling.

Considering these limitations, major CPU vendors have stepped away from increasing the raw clock rate and instead focused more on adding on-chip support for multi-threading by increasing the number of cores. Dual- and quad-core processors are now common.

There are numerous applications of advances in multi-threading in scientific works. The problem of image processing is not an exception, as it is a computationally intensive task which we aim to accelerate a basic version of it in this study using the parallel programming paradigm.

The remainder of this work is organized as the following: In section II, firstly in II-A, the four image filtering algorithms used for this work and their underlying theory are briefly explained. Secondly, in II-B, the OpenMP is briefly explained followed by an elaboration on the concepts of threading in II-B1, and scheduling in II-B2, as the two concepts play a key role in this work's implementation, which is the focus of section III, elaborating on the structure of the program. At section IV, after choosing the optimal scheduling type, the runtime of parallelized program is reported in IV-A for different threads, and in IV-B, the Vtune Profiler software from the Intel oneAPI Base Toolkit have been used to analyze and visualize the program's performance, especially the parallelism effect. Finally, in V, main findings are summarized.

II. Preliminary Concepts

A. Image Processing Algorithms

1) Reflect: Using this filter is what occurs in front of a mirror, i.e., the pixels are swapped horizontally. Therefore, the width of image is used here for finding the proper indices during swapping, and the width serves as the middle point in each pixels' row where its left pixels are swapped with right pixels.

2) Edge: Image edge refers to the highest part of partial intensity changes in images. Image edges are the most basic features of an image. In image processing algorithms, it is often useful to detect edges in an image, i.e., lines in the image that create a boundary between one object and another, which can play a key role in pattern recognition task. One way to achieve edge detection is by applying the Sobel operator to the image [3].

Intuitively, the Sobel operator is a discrete differentiation operator, which is computed by approximating gradient of the image intensity function $f(x,y)$. More precisely, the Sobel operator is the partial derivative of $f(x,y)$, which serves as the central computing $3*3$ neighbourhood at x,y directions.

In what follows, gradient approximation equations used in Sobel operator are provided:

$$G_x = \{f(x+1,y-1)+2f(x+1,y)+f(x+1,y+1)\} - \{f(x-1,y-1)+2f(x-1,y)+f(x-1,y-1)\} \quad (1)$$

$$G_y = \{f(x+1,y-1)+2f(x+1,y)+f(x+1,y+1)\} - \{f(x-1,y-1)+2f(x-1,y)+f(x-1,y-1)\} \quad (2)$$

To apply the operator image, the following convolution templates are used as kernels:

$$G_x \quad G_y$$
$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Using these kernels, G_x and G_y values are computed for each of the red, green, and blue channels for a pixel. Since each channel can contain only one value, G_x and G_y quantities are combined by computing their L2 norm, as the following:

$$g(x, y) = \sqrt{G_x^2 + G_y^2}$$

3) Grayscale: A light devoid of color is called monochromatic, which only contains intensity as an attribute. The term "gray level" is commonly used to denote monochromatic intensity. The range of monochromatic light from black to white is usually called the gray scale, and monochromatic images are frequently referred to as grayscale images [4].

In the grayscale filter, a chromatic image (RGB image) is inputted, and it is converted to grayscale image. To achieve this, all the color channels (i.e., R, G, and B) values should be set to an equal number. For finding the proper number, an averaging of the three channels is used.

4) Blur: There are numerous means of creating the blurring effect, i.e., softening of an image. For this purpose, the "box blur" algorithm is used, which operates by taking each pixel, and for each color channel, outputs an average of color values of the neighbouring pixels. It is the quickest blurring algorithm, but it has a drawback i.e., lacking smoothness of a Gaussian blur [5]. The algorithm is based on the fact that sum S of elements in the rectangular window can be decomposed into sums C of columns of this window, in the following manner:

$$S[I, J] = \sum_{k=-r}^{n} C(I, j + k)$$

B. OpenMP

Openmp is a parallel programming model for shared memory and distributed shared memory multiprocessors. It uses compiler "pragmas" for C/C++ and compiler directives in Fortran to decorate serial code with parallelization information that describes to the compiler how to turn it into a parallel code [6].

OpenMP is primarily designed for shared memory multiprocesseors. In the last decade, there has been a significant increase in usage of shared memory parallel systems. Not only have they become more prevalent, but the number of processors used in those system have been increased. However, most parallel programming models are originally designed for distributed memory systems. Therefore, there grew a lack of compatibility between the state of hardware and software APIs that support them. OpenMP aims at providing a standalone and portable API for writing shared memory parallel programs [7].

1) Threading: Threading provides a mechanism for programmers to divide their programs into more or less independent tasks with the property that when one thread is blocked, another thread can be run. An OpenMP application always begins with a single thread of control, called the "master thread", which exists for the duration of the program. During execution, we can include parallel regions inside master thread, at which point the master thread will fork new threads. At the end of the parallel region, the forked threads will terminate, and the master thread continues executing. Nested parallelism, in which work threads fork additional threads can be used to parallelize `for` loops [8]. Choosing number of threads depends on the amount of resources, and the application.

2) Scheduling: In OpenMP, as assignment of iterations to thread, which is called "scheduling", can have a marked effect on performance, the `schedule` clause can be used to assign iterations in a parallel `for` loop. Three types of scheduling is used in this work, which are explained in the following:

- `static`: The iterations can be assigned to the threads before the loop is executed.
- `dynamic`: The iterations are assigned to the threads while the loop is executing, so after a thread completes its current set of iterations, it can request more from the run-time system.
- `guided`: Similar to `dynamic` scheduling, but the chunk size starts off large and decreases to better handle load imbalance between iterations.

In order to find the optimal scheduling type as well as optimal number of threads, a series of experiments were carried out, which are explained in detail in IV-A.

III. Implementation

Two bitmap images are used to apply the filters (introduced in II-A) using the program. One of the images is a small image (720 KB) called "yard", and another is a large image (53.7 MB) called "field". The images are displayed in 1.

Fig. 1. Left: yard image (small), Right: field image (large)

The general framework of the this work's implementation is provided in figure 2.

The main function contains two major helper files, i.e., a bitmap header file (bmp.h) and a helper function (helpers.h). The former contains header for specific bitmap format images and in this helper a data structure called `RGBTRIPLE` is defined in the following way:

```
typedef struct
{
    BYTE  rgbtBlue;
```

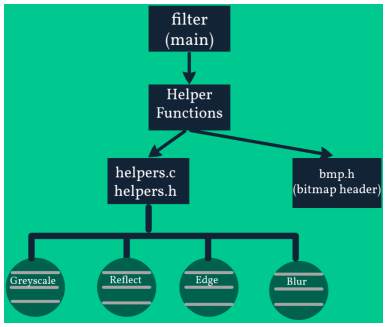

Fig. 2. Structure of the program.

```
    BYTE  rgbtGreen;
    BYTE  rgbtRed;
} __attribute__((__packed__))
RGBTRIPLE;
```

This data structure is the format used for reading, manipulating, and storing images in the program.

The latter helper file, `helpers.h`, addresses the helper functions in `helpers.c`, which are the filters applied to the images, i.e., grayscale, reflect, edge, and blur (explained in II-A). These functions will be called filter functions henceforth. All of the filter functions contain `for` loops that are parallelized using OpenMP. As seen in the figure 2, the grey horizontal lines are drawn on the filter functions, which indicate parallelized part of the program. For the purpose of illustration, the code for using OpenMP in the filter function `reflect` is provided in the following:

```
#pragma omp parallel for collapse(2)
for (int i = 0; i < height; i++)
{
  for (int j = 0; j < width / 2; j++)
  {
    RGBTRIPLE temp = image[i][j];
    image[i][j] = image[i][width - (j + 1)];
    image[i][width - (j + 1)] = temp;
  }
}
```

The main functions accepts three command line arguments, i.e., the type of the filter user wants (`b` for blur, `e` for edge detection, `g` for grayscale, and `r` for reflect), input image, and output image. An instance of running the complied filed of the program is provided in the following:

```
./a.out -g images/input.bmp
output/output.bmp
```

In this instance, the first argument "g" specifies the grayscale filter to be used. The second and third arguments determine the location of input image and output image respectively.

The result of applying filters on the yard image are displayed in 3.

The system used to run the implementations were the HPC's Iris Cluster supercomputer, with the following command:

```
si -N 1 --exclusive -t 01:00:00
```

Fig. 3. Result of filters applied on yard image. (Upper Left): Grayscale, (Upper Right): Blur, (Lower Left): Edge, (Lower Right): Reflect

The command implies 1 node is used and all the cores available will be used, based on the Iris cluster's default behavior.

In remainder of this work, the results of runinng the parallelized program is reported, discussed, and visualized.

IV. Results and Discussion

A. Runtime

The first goal of the experiments were finding the optimal scheduling type, and number of threads. For this purpose, a series of experiments were carried out on two images shown in 1, with the following options:

`omp_set_num_threads: 2, 4, 8, 16`
`schedule: static, dynamic, guided`

For all the possible threads and for both images, the `static` was the fastest scheduling type.

Using only `static` schedule, the runtime of different filters, for different threads are reported in tables 5 and 4 for the field image and yard image, respectively. On the yard image, the filters edge and blur was not applied, as it led to memory issues.

For comparison, an additional value 1 is added for Threads in tables, which indicates serial version of code (without using OpenMP at all).

Threads	Filter			
	G	E	B	R
1	***0.005780s**	***0.060381s**	***0.032254s**	***0.001788s**
2	0.006142s	0.314986s	0.281149s	0.002983s
4	0.010052s	0.344814s	0.311375s	0.006919s
8	0.017035s	0.361975s	0.324621s	0.014111s
16	0.033176s	0.402725s	0.392879s	0.028957s

Fig. 4. Runtime of yard image for different threads

Threads	Filter			
	G	E	B	R
1	0.380839s	-	-	0.109629s
2	***0.317647s**	-	-	***0.101627s**
4	0.350148s	-	-	0.112447s
8	0.371629s	-	-	0.121465s
16	0.391413s	-	-	0.137468s

Fig. 5. Runtime of field image for different threads

According to the runtime of yard image 4, the serial version is the faster version. The speculation for the reason is that the image is small, and the filter functions are not computationally complex, to the point that allocating resources for incorporating parallellism consumes more time than the filter function itself. This effect is reduced for field image, based on its runtime 5, as this image has significantly larger size than yard image, and therefore there is slight improvement in parallel version of the filter functions, when using 2 threads. But using more threads worsens the performance.

B. Intel Vtune Profiler

The Intel Vtune Profiler is used on the parallel version of the grayscale filter function on the field image, for two types, i.e., with 2 threads (the optimal number as seen in IV-A) and with 32 threads (the largest number of threads possible). In 6, the result of application performance snapshot (APS) tool is displayed for both number of threads. As evident from the figure, the OpenMP imbalance is near zero, and the parallelism did not have a noticeable improvement. This can be due to the computational complexity of the filter functions being not high enough, to the point that allocating resources for incorporating parallelism consume more time than the filter function itself. To have a more clear view of the consumption of resource and time by different functions of the program, Vtune Profiler results are displayed in 7, when using 2 threads, and in 8 when using 32 threads. In the above of the two aforementioned figures, the "Top Hotspots" are listed, which are the most active functions of the program. In below of the figures, the "Effective CPU Utilization Histogram" is provided, displaying a percentage of the wall time the specific number of CPUs were running simultaneously. Evidenced by the Top Hotspots of the two aforementioned figures, both types of the parallel version (2 threads and 32 threads), the forking function `__kmp_fork_barrier` takes the most time among the functions of the program, and it takes more time as number of threads increase. Therefore, the speculation is that if the complexity of the filter functions were high, or very larger image was used as input (although the ones used were the largest found on the internet), then the time consumption of the filter function exceeds the forking function, and the effect of parallelism and its improvement would become more evident.

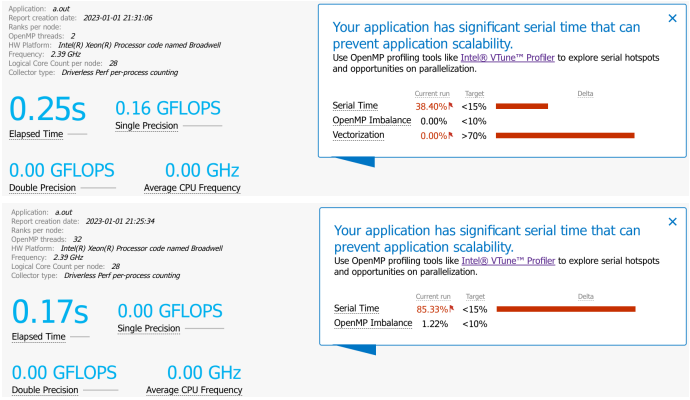

Fig. 6. APS for the field image with thread count 2 (above) and 32 (below)

Function	Module	CPU Time	% of CPU Time
grayscale	a.out	0.310s	67.4%
fread	libc.so.6	0.050s	10.9%
__kmp_fork_barrier	libiomp5.so	0.050s	10.9%
fwrite	libc.so.6	0.040s	8.7%
__kmp_api_omp_get_max_threads	libiomp5.so	0.010s	2.2%

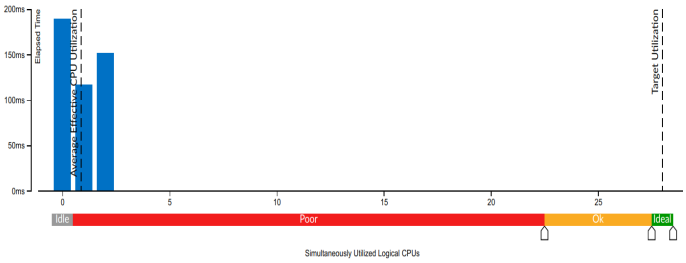

Fig. 7. Vtune Profiling of field image with 2 threads.
Above: Top Hotspots, Below: Effective CPU Utilization Histogram

V. Conclusion

This report explained parallel implementation of different image processing algorithms (filters) on bitmap formatted images. All filters were parallelized using the OpenMP application programming interface (API). The implementations were performed on high performance computing (HPC) clusters. Experimental results show that using static scheduling and using two threads led to the fastest runtime when using a large image, though the improvement was minor and not significantly evident. Evidenced by the intel Vtune profiler of the program, the forking function in the parallel version of the program consumes the most time, and this consumption increases with the number of threads. The speculation is that if more computationally complex filters are used, or very larger image is used, then the parallelism effect would become more evident.

Function	Module	CPU Time	% of CPU Time
__kmp_fork_barrier	libiomp5.so	1.554s	75.5%
grayscale	a.out	0.357s	17.3%
fread	libc.so.6	0.050s	2.4%
fwrite	libc.so.6	0.040s	1.9%
_INTERNAL8aaf6219::__kmp_itt_thread_name	libiomp5.so	0.020s	1.0%
[Others]	N/A*	0.039s	1.9%

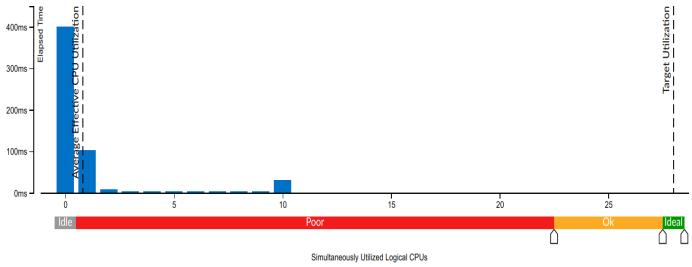

Fig. 8. Vtune Profiling of field image with 32 threads.
Above: Top Hotspots, Below: Effective CPU Utilization Histogram

REFERENCES

[1] G. Blake, R. Dreslinski, and T. Mudge, "A survey of multicore processors," *Signal Processing Magazine, IEEE*, vol. 26, pp. 26 – 37, 12 2009.

[2] G. E. Moore, "Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff." *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 3, pp. 33–35, 2006.

[3] Z. Jin-Yu, C. Yan, and H. Xian-Xiang, "Edge detection of images based on improved sobel operator and genetic algorithms," in *2009 International Conference on Image Analysis and Signal Processing*, 2009, pp. 31–35.

[4] R. C. Gonzalez and R. E. Woods, *Digital Image Processing (3rd Edition)*. USA: Prentice-Hall, Inc., 2006.

[5] R. Chandel and G. Gupta, "Image filtering algorithms and techniques: A review," 2013.

[6] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel Programming in OpenMP*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[7] M. Klemm and J. Cownie, *High Performance Parallel Runtimes*. Berlin, Boston: De Gruyter Oldenbourg, 2021. [Online]. Available: https://doi.org/10.1515/9783110632729

[8] P. Pacheco, *An Introduction to Parallel Programming*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.