High Performance Computing

Image Search Project
By Developing Parallel Application
(performance analysis report)

Name: Meisam Amjad

Objective

The objective of this project is to gain experience with OpenMP to develop parallel applications by developing a reasonably straightforward image searching program and verifying its performance on Red Hawk cluster server.

Background

This project is a relatively straightforward (aka brute force) image searching approach, which requires the development of an OpenMP-parallelized image searching program. The images being used in this project are Portable Network Graphics (PNG) files (see: http://en.wikipedia.org/wiki/Portable_Network_Graphics) that stores each pixel in an image. In other words, the image is a matrix of pixels and each pixel consists of four 8-bit values corresponding to the Red-Green-Blue-Alpha values. There are several different ways to view PNG files on Linux. The common command that I use is:

```
$ eog star_mask.png
```

The PNG class (review the supplied PNG.h and PNG.cpp files) that facilitates reading and writing of PNG files that contain images in RGBA (Red-Green-Blue-Alpha) format. Note that the supplied PNG class requires the images to be in RGBA format otherwise it will not read such files (and will generate an exception). The pixels in an image may be obtained via call to the PNG::getBuffer() method. This method returns a "flat" buffer in which the pixels are stored in a row-major organization as illustrated in the figure below:

p11	p12	P13
P21	p22	p23
p31	p32	P33
p41	p42	P43

Conceptual organization of pixels in an image.

p11	p12	p13	p21	p22	p23	p31	p32	p33	p41	p42	p43

Row-major organization of pixel data in the buffer returned by `PNG::getBuffer()` method.

Each pixel consists of four 8-bit values stored in the order Red-Green-Blue-Alpha and they can be accessed as suggested below:

```
PNG img;
img.load("Mammogram.png");
// Maybe some more code here...
const std::vector<unsigned char>& buffer = img.getBuffer();
// Use a getPixelIndex method to convert row, col to index in buffer
const int index = getPixelIndex(10, 10, img.getWidth());
std::cout << "red   = " << buffer[index]
          << "green = " << buffer[index + 1]
          << "blue  = " << buffer[index + 2]
          << "Alpha = " << buffer[index + 3] << std::endl;
```

Compiling:

The supplied PNG.cpp utilizes the system image processing libraries. Consequently, when compiling your program ensure you link with the library as shown below:

```
$ g++ -g –Wall –O3 –stdc++14 –fopenmp ImageSearch.cpp PNG.cpp –o ImageSearch -lpng
```

Description:

This project requires development of a suitably parallelized OpenMP- based image search program that can identify and mark (with a red-box) occurrences of a given sub-image in a larger image as illustrated in the figure below:

The Larger image: This is the image to search in (first command-line argument to the program)	
The sub-image: This is the sub-image, aka *mask*, to search for (second command-line argument to program) in the given larger image.	
Resulting image: This is an image generated by the program in which sub- images in a black-box (output image name is supplied as third command-line argument to program). **Note: You need to zoom-in to see the red-boxes around each one of the 50 stars.**	

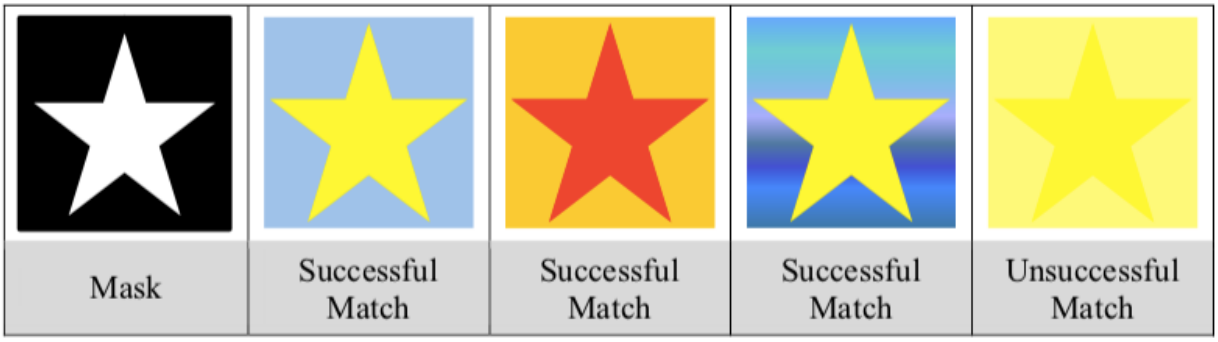
The image search program designed to handle searching for matches using an image "mask". An image mask is a black-and-white image that essentially represents a pattern to search for. Examples of image masks are shown in the adjacent figure. The pixels in an image mask are interpreted as follows:

Example Masks

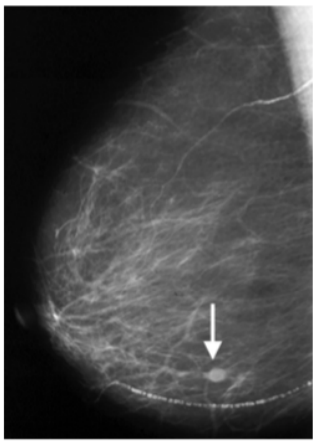
- Pixels that are black essentially define the "background" for the information we are searching.

- Pixels that are not-black (or shades of grey) is the information or meaningful pattern to be identified.

The objective of the search is to essentially distinguish the meaningful information immaterial of the (background and foreground) colors – the key distinguishing factor is sufficient contrast between the background and foreground (with black-and-white being the extreme contrast) as illustrated by the following examples:



Searching using masks is a powerful strategy not only for color agnostic processing but is also a handy strategy for processing images and information where the contrast in regions of the images is important. For example, this strategy can be used to identify high risk areas with potential tumors from mammograms (see adjacent image), high-contrast CT scans and MRI images.



With a mask the key distinguishing factor is the contrast between background (identified by black pixels in the mask) versus the information (or foreground identified by white pixels in the mask). Consequently, the image search proceed in the following manner:

1. Logically slide/move the mask over the image, row-by-row and column-by-column (from top-left to bottom-right of search image; somewhat akin to blocks in block matrix multiplication) to delineate the region of the image to search and check for potential match as suggested below.

2. Given a region of the image to search, it first computes the average background-averaging the color of the pixels corresponding to the black pixels in the mask. For example, given the series of star images above, in the first successful case, all pixels corresponding to the light-blue color would be averaged (since the corresponding pixels in the mask are black).

3. Once the average background color has been determined, next rechecks each pixel to verify if it matches (or mismatches) the mask in the following manner:

    - If the corresponding pixel in the mask is black, then the pixel should be "*same shade*[§]" of the average background.

    - If the corresponding pixel in the mask is white, then the pixel should not be the "*same shade*[§]" as the average background.

    § In this context, two colors are considered to be the "*same shade*" as determined by pixel color tolerance value (sixth optional command-line argument to the program).

4. The net-matching pixels are computed by subtracting the number of mismatching pixels (see previous step) from the number of matching pixels (see previous step), that is: netMatch = matchPixCount – mismatchPixCount. If the net number of matching pixels is more than the required percentage pixel (perPixMatch) match (fifth optional command-line argument to the program) then a match is found – that is, area matches if netMatch > mask.getWidth() * mask.getHeight() * perPixMatch / 100.

5. Once a region of the image has been matched, it must not be included as part of any other matches. We aim for using a list that tracks the regions that has been matched.

6. Once all the regions have been searched, it sorts the list of matches (based on row and then on column) and prints the list of matched regions. The matching regions are printed in the form: row, col, row + mask.height, col + mask.width (see sample output for details)

7. For each matched region, draws a red box around the region.

Search Options:

The search for sub-images must permit the following additional options:

1. **Percentage pixel match:** A section of the search image must is considered as a match to the given sub-image if sufficient number of pixels matches. This parameter specifies the percentage number of pixels that must match (based on second option below) in order to decide that a region of the image is a match to the given sub-image. This parameter is a value in the range 1 to 100 (corresponding to 0% to 100%). For example, if this parameter is 50, then a 50% pixel match is deemed sufficient to identify a section of the search image as a match to the search image. Accordingly, with a 50% match the image search program should be able to identify stars (using the search sub-image shown earlier) and identify 50 stars on the following image (which has some partial stars showing (possibly because the flag was waving or the camera had some distortion in it). This is an optional command-line argument to the program. This value may be specified as the fifth command-line argument to the program. If sufficient number of arguments is not supplied, then the default value for this option is 75 (corresponding to 75% match).

Source image (with some of the stars distorted)

Resulting image with the stars identified using a 50% match of pixels.

2. **Pixel color tolerance:** The three color values for each pixel (namely RGB) does not need to match exactly and can match approximately with a given tolerance. This option is an integer the range 1 to 255 that indicates the acceptable tolerance (or difference) between pixel color values in the search image versus the sub- image. For example, assume that a pixel has values **<**230, 0, 255**>** and the tolerance is set to 10 then the following pixels would be a match (as none of the individual RGB values differ by more than 10): <240, 0, 255>, <220, 9, 250>, <230, 9, 246> while the following pixels will not be a match: <200, 0, 255>, <230, 20, 255>, <210, 0, 240). In other words, close shades of the same color are acceptable but not shades of other colors. This is an optional command-line argument to the program. This value may be specified as the sixth command-line argument to the program. If sufficient number of arguments is not supplied, then the default value for this option is 32.

Command-line Arguments to program:

The following 6 command-line arguments (the last three are optional) are accepted and suitably be processed by the program:

1. The first required command-line argument is the path to the large PNG file to be searched-in.

2. The second required command-line argument is the path to the PNG file that contains the sub-image to be searched for.

3. The third required command-line argument is the name of the output PNG file to which the resulting image is to be written.

4. The fourth optional command-line argument is a Boolean string ("true" or "false") to indicate if the search sub-image is a mask. In this homework you may assume this parameter is always true. (and safely ignore it)

5. The fifth optional command-line argument is the desired percentage pixel match (as described earlier) to determine a match between the given sub-image and the search image. **If this parameter is not specified then it uses 75 as the default value**.

6. The sixth optional command-line argument is the pixel color tolerance (as described earlier). **If this argument is not specified it then uses 32 as the default value**.

Sample Outputs:

```
$ ./ImageSearch TestImage.png and_mask.png result.png true 75 16
sub-image matched at: 73, 630, 85, 660
sub-image matched at: 120, 310, 132, 340
sub-image matched at: 202, 677, 214, 707
sub-image matched at: 226, 864, 238, 894
sub-image matched at: 274, 67, 286, 97
Number of matches: 5
```

```
$ ./ImageSearch MiamiMarcumCenter.png WindowPane_mask.png result.png true 50 64
sub-image matched at: 567, 818, 601, 859
sub-image matched at: 567, 1021, 601, 1062
sub-image matched at: 568, 619, 602, 660
sub-image matched at: 568, 1226, 602, 1267
sub-image matched at: 578, 1791, 612, 1832
sub-image matched at: 582, 1996, 616, 2037
sub-image matched at: 590, 2198, 624, 2239
sub-image matched at: 605, 817, 639, 858
sub-image matched at: 605, 1020, 639, 1061
sub-image matched at: 606, 618, 640, 659
sub-image matched at: 607, 1225, 641, 1266
sub-image matched at: 616, 1791, 650, 1832
sub-image matched at: 620, 1995, 654, 2036
sub-image matched at: 627, 2197, 661, 2238
sub-image matched at: 816, 1209, 850, 1250
Number of matches: 15
```

Experimental Platform

The experiments documented in this report were conducted on the Red Hawk cluster with the following configuration:

Component	***Details***
CPU Model	Intel(R) Xeon(R) CPU X5550 @ 2.67GHz
CPU/Core Speed	2661.000 Mhz
Main Memory (RAM) size	24591648 kB
Operating system used	Linux mualhpcp01.hpc.muohio.edu 2.6.32-642.el6.x86_64 #1 SMP Tue May 10 17:27:01 UTC2016 x86_64 x86_64 x86_64 GNU/ Linux
Interconnect type & speed (if applicable)	n/a
Was machine dedicated to task (`yes/no`)	Yes (via a qsub job)
Name and version of C++ compiler (if used)	GCC 4.9.2
Name and version of Java compiler (if used)	n/a
Name and version of other non-standard software tools & components (if used)	n/a

Parallelization strategy

Since the data is big and only one task needs to be implemented on data I used data-parallelism and I only used 'parallel for' on the main loop inside the search(...) method which basically has been used for looping through the rows. That is, each thread works on different set of rows and I also tried to parallel other loops which although they got run with multithreads but the performance decreased due to the style of my code for other loops (using pointers) and combination of all sources and threads working together. Having multithreads, since each thread almost needs the same amount of time for calculation, I used static scheduling and the result was a decent amount of CPU usage (around ~770% for 8 threads based on the image). Although sometimes unbalanced scheduling happened in some cases, because finding a match in one region caused jumping to the next region (as I defined to be happened) which caused one thread has less work to do. I also tried dynamic and guided scheduling and the performance did not

change a lot. I used the thread number for saving the results in a vector for preventing race conditioning and at the end merged all results together.

Test Case #1

The following performance statistics were collated by using the supplied sample data files `Mammogram.png` and `Cancer_mask.png` shown in the images further below. The image match was conducted with 75% match with color tolerance of 32

The program was compiled using the following command line:

```
$  c++ -g -Wall -fopenmp -lpng -std=c++14 -O3 PNG.cpp ImageSearch.cpp -o ImageSearch
```

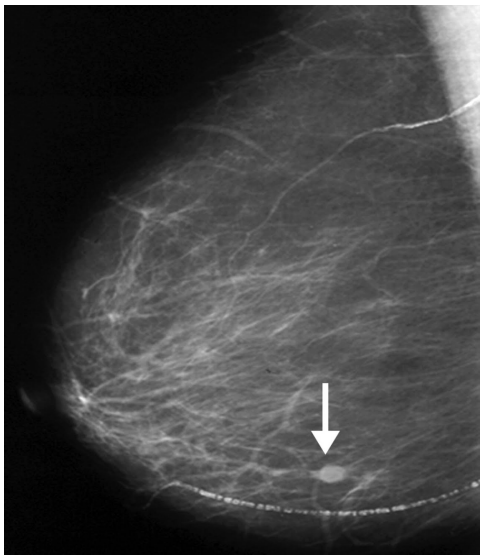
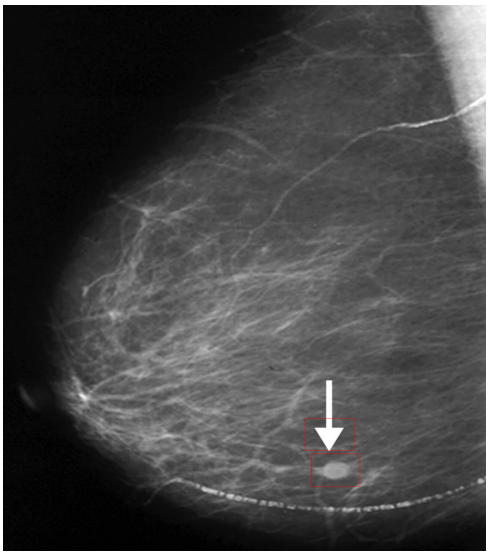
The program was run and timings were collected using the following key lines in the PBS-job script.

```
for threads in 1 4 8;
do
   export OMP_NUM_THREADS=${threads}
   /usr/bin/time -v ./ImageSearch Mammogram.png Cancer_mask.png result_Mammogram_${threads}.png
   true 75 32
done
```

Textual Output verification:

Expected Output	program Output
sub-image matched at: 1211, 714, 1306, 829 sub-image matched at: 1315, 727, 1410, 842 Number of matches: 2	sub-image matched at: 1211, 714, 1306, 829 sub-image matched at: 1315, 727, 1410, 842 Number of matches: 2

Visual output verification:

Search Image	Mask Image	Output Image
		

Performance statistics:

#Threads	User Time	Elapsed Time	%CPU
1	188.80 sec	188.82 ± 0.2423 sec	99%
4	191.33 sec	49.372 ± 0.1682 sec	387.2%
8	200.62 sec	25.944 ± 0.1984 sec	772.6%

Test Case #2

The following performance statistics were collated by using the supplied sample data files `TestImage.png` and `and_mask.png` shown in the images further below. The image match was conducted with 75% match with color tolerance of 16

The program was compiled using the following command line:

```
$ c++ -g -Wall -fopenmp -lpng -std=c++14 -O3 PNG.cpp ImageSearch.cpp -o ImageSearch
```

The program was run and timings were collected using the following command line:

```
for threads in 1 4 8;
do
   export OMP_NUM_THREADS=${threads}
   /usr/bin/time -v ./ImageSearch TestImage.png and_mask.png result_TestImage_${threads}.png true
75
   16
done
```

Textual Output verification:

Expected Output	Program Output
sub-image matched at: 73, 630, 85, 660 sub-image matched at: 120, 310, 132, 340 sub-image matched at: 202, 677, 214, 707 sub-image matched at: 226, 864, 238, 894 sub-image matched at: 274, 67, 286, 97 Number of matches: 5	sub-image matched at: 73, 630, 85, 660 sub-image matched at: 120, 310, 132, 340 sub-image matched at: 202, 677, 214, 707 sub-image matched at: 226, 864, 238, 894 sub-image matched at: 274, 67, 286, 97 Number of matches: 5

Visual output verification:

Search Image	Mask Image	Output Image
		

Performance statistics:

#Threads	User Time	Elapsed Time	%CPU
1	6.478 sec	6.49 ± 0.0111sec	99 %
4	7.456 sec	2.27 ± 0.0248	327.6 %
8	8.42 sec	1.396 ± 0.0601sec	601.4 %

Test Case #3

The following performance statistics were collated by using the supplied sample data files `MiamiMarcumCenter.png` and `WindowPane_mask.png` shown in the images further below. The image match was conducted with 50% match with color tolerance of 64.

The program was compiled using the following command line:

```
$  c++ -g -Wall -fopenmp -lpng -std=c++14 -O3 PNG.cpp ImageSearch.cpp -o ImageSearch
```

The program was run and timings were collected using the following command line:

```
for threads in 1 4 8;
do
   export OMP_NUM_THREADS=${threads}
   /usr/bin/time -v ./ImageSearch MiamiMarcumCenter.png WindowPane_mask.png
   result_MiamiMarcumCenter_${threads}.png true 50 64
done
```

Textual Output verification:

Expected Output	Program Output
sub-image matched at: 567, 818, 601, 859 sub-image matched at: 567, 1021, 601, 1062 sub-image matched at: 568, 619, 602, 660 sub-image matched at: 568, 1226, 602, 1267 sub-image matched at: 578, 1791, 612, 1832 sub-image matched at: 582, 1996, 616, 2037 sub-image matched at: 590, 2198, 624, 2239 sub-image matched at: 605, 817, 639, 858 sub-image matched at: 605, 1020, 639, 1061 sub-image matched at: 606, 618, 640, 659 sub-image matched at: 607, 1225, 641, 1266 sub-image matched at: 616, 1791, 650, 1832 sub-image matched at: 620, 1995, 654, 2036 sub-image matched at: 627, 2197, 661, 2238 sub-image matched at: 816, 1209, 850, 1250 Number of matches: 15	sub-image matched at: 567, 818, 601, 859 sub-image matched at: 567, 1021, 601, 1062 sub-image matched at: 568, 619, 602, 660 sub-image matched at: 568, 1226, 602, 1267 sub-image matched at: 578, 1791, 612, 1832 sub-image matched at: 582, 1996, 616, 2037 sub-image matched at: 590, 2198, 624, 2239 sub-image matched at: 605, 817, 639, 858 sub-image matched at: 605, 1020, 639, 1061 sub-image matched at: 606, 618, 640, 659 sub-image matched at: 607, 1225, 641, 1266 sub-image matched at: 616, 1791, 650, 1832 sub-image matched at: 620, 1995, 654, 2036 sub-image matched at: 627, 2197, 661, 2238 sub-image matched at: 816, 1209, 850, 1250 Number of matches: 15

Visual output verification:

Search Image	Mask Image	Output Image
		

Performance statistics:

#Threads	User Time	Elapsed Time	%CPU
1	98.154 sec	98.296 ± 0.0487	99 %
4	100.114 sec	29.878 ± 0.0927	334.8 %
8	105.11 sec	18.214 ± 0.0879	578 %

Inferences

Implementing explicit parallelism by using OpenMp and multi-threads looks correct since the program returns the same output as running it with the single thread or different threads. Increase in number of threads, increases the CPU usage, almost additional 100% for adding each extra thread. Although we can see difference between theoretical and expected runtime due to NUMA architecture by increasing the number of threads. Speed up increases by adding threads and by calculating 95% CI for observe elapsed times, we can clearly see the improvement in speed up by increasing threads(from 5-7x based on the images). Because the program works with output streams at the end for demonstrating results, we can see slight system time and File System Output which in each case is almost the same time number since this part of program work under only 1 thread. Having the decent amount of CPU usage also indicates the correct implementation of multithreading such as ~387% for 4 threads and ~770% for 8 threads.