

# Implementation of a Multicore Computing strategy for Image Smoothing Filters using POSIX Threads

Rahul Sharma (Graduate Student, UNC Charlotte), Member, IEEE  
(Submitted in completion of requirements for ECGR 6090 – Multicore Computing)

**Abstract** – In an attempt to extract maximum performance from processor cores, various methods such as pipelines, superscalar architecture, instruction level parallelism, better cache management, higher clock speeds have been implemented to great effect and success. However due to saturation in such areas, the latest approach in extracting processor performance has focused on increasing the number of processor cores, which in return has required scientists and engineers to look for functional and data parallelism in conventional engineering problems. This paper attempts to analyze one such problem in image processing, where a program was implemented to test the principles of the discipline using low level parallelism concepts as specified by the POSIX threads library. It also documents an implementation of this image processing application on parallel architectures.

**Index Terms:** Multicore Computing, POSIX, Spatial smoothing, Image processing, multi-threading

## I. INTRODUCTION

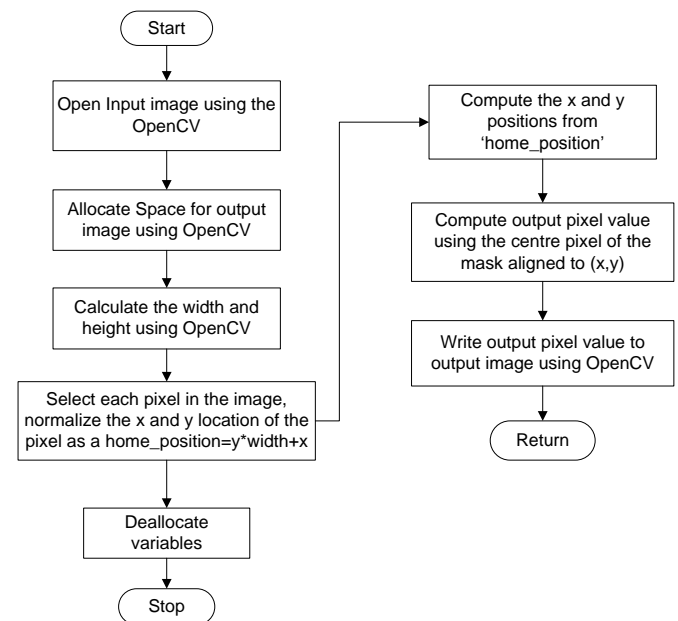
Image smoothing filters have traditionally found use in noise removal from high resolution astronomical and satellite imagery which tend to be affected by extra-terrestrial noise and interference. Image Processing, by nature depends on heuristics to make educated guesses and trial and error judgments to achieve the optimal output effect. For example, an Image Smoothing operation on an astronomical image must take into account aspects of such nature that the image processing filter does not mistake an object of interest such as a star or an asteroid to be an effect of spatial noise. This requires prior knowledge and experience and often the output may be a compromise requiring a tradeoff.

From a high performance computing standpoint, the images tend to be extremely large with lossless compression or no compression applied to it, so that data losses are kept to a minimum. Also the data sizes tend to be huge requiring a large amount of storage space. This tends to tax the computing system with large amounts of data being passed around. When processing this data, the typical problems

expected to be faced are cache management (avoiding/reducing cache misses), keeping CPU utilization to the maximum which can particularly be difficult since a single memory has to feed multiple cores.

Typically, the best parameter to test concepts in Multicore high performance applications has been the measurement of execution time. The execution time succinctly covers all aspects of computing such as setup time, the computing, collection of data, memory reads and writes, cache misses and data size. Hence execution time has been used extensively to profile the problem and solution code in its sequential and parallel forms.

## II. THE SEQUENTIAL PROGRAMMING APPROACH



**Fig1: Flowchart of the sequential approach**

An analysis of the problem at hand reveals that the best memory access times can be achieved by reading the entire image to the RAM given that most modern machines have at

least 1 Gigabyte of RAM memory. It is also required that array accesses be made in a row major format to reduce cache misses. An open source computer vision library OpenCV developed by Intel was used for overcoming the memory access problem.

The image smoothing operation is done by picking up a pixel value. Depending on the size of the mask, the center pixel of the mask is aligned to the x and y locations of the image. In an all integer ALU operation, the average value with respect to the entire neighborhood is calculated. Using the OpenCV library function, the pixel value of the output is written back to the output matrix. This operation is performed for each pixel in the image.

### III. THE PARALLEL PROGRAMMING APPROACH

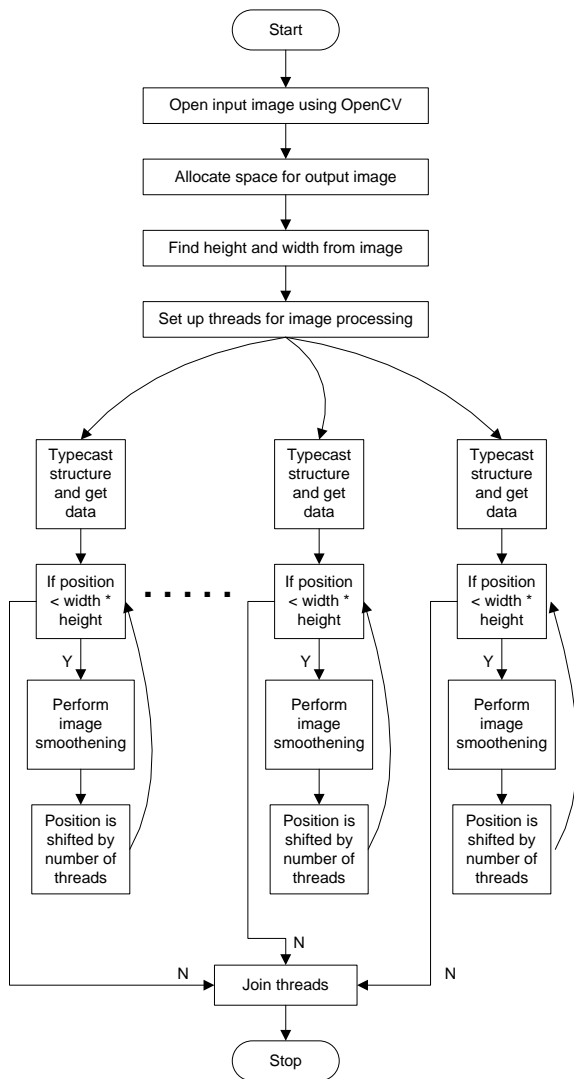


Fig 2: Flowchart of the Parallel Programming approach

In the parallel programming approach, it is imperative to keep memory accesses as row-major as possible to avoid cache misses. Hence the functional approach towards accomplishing this task is geared to conform to this requirement.

When the threads are set up, the starting position for each thread operation is set up in a row major manner.

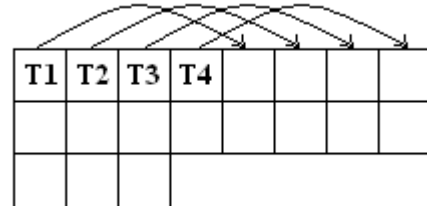


Fig 3: Pictorial representation of the thread-pixel assignment conforming to a row major memory access focused on cache miss reduction for four threads. (This strategy can be applied to larger number of threads)

The initial assignment of the thread is represented in figure 3. The x and y values are normalized to a variable which is calculated position.

$$position = y * width + x$$

This value is assigned to the thread which obtains back the values of x and y as follows:

$$x = position / width \quad \text{-- Integer division}$$

$$y = position \% width \quad \text{-- Modular division}$$

Using these values, the mask is aligned to this position of the image and output value is calculated and assigned to the same x and y position in the output image. The purpose of these conversions of positional values is so that when ‘position’ value reaches the end of one row, the position value will point to the first value in the next row.

The strategy employed in the parallel program also avoids the use of global variables which in turn does away with mutexes (mutual exclusion components that prevent thread conflicts when accessing global resources). The reduction in global values and exclusive resources for all of the threads also contributes to thread independence and a non requirement of thread communication. The above mentioned factors reduce execution time of the program.

### IV. PERFORMANCE GAINS

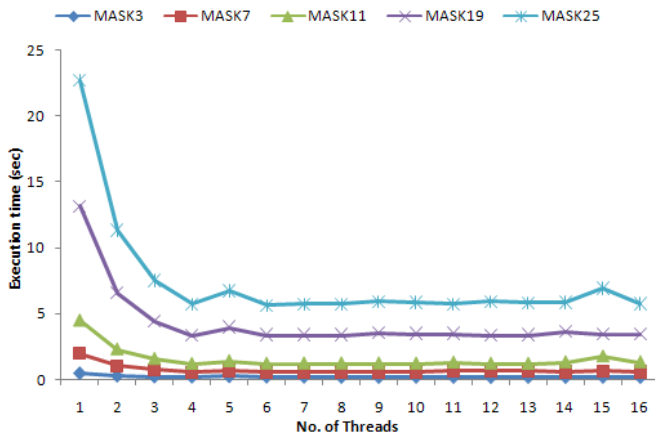
The parallel programming strategy discussed in section III shows an improvement over the traditional sequential version of the process. The strategy was compared to the traditional

version over various Windows Bitmap BMP sample input images. The smallest image in the sample set was 481 kilobytes and the largest was 6.2 Megabytes.

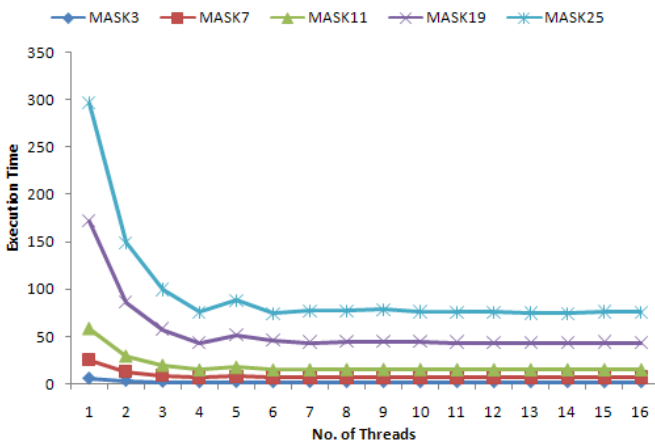
The program was tested on a multiprocessor server machine consisting of 4 Intel Xeon CPUs running Red Hat Linux at University of North Carolina at Charlotte during non-peak load conditions.

Data-size	481 Kilobyte	6.2 Megabytes
<b>MASK3 - 3*3</b>	0.39	5.3
<b>MASK7 - 7*7</b>	1.84	23.97
<b>MASK11 - 11*11</b>	4.42	57.5
<b>MASK19 - 19*19</b>	12.84	170.27
<b>MASK25 - 25*25</b>	22.14	294.72

**Fig 4: Execution times in seconds for sample sizes with various sized masks using the Sequential Approach**



**Fig 5: Effect of number of threads on execution time on a 481 Kilobyte sample image**



**Fig 6: Effect of number of threads on execution time on a 6.2 Megabyte sample image**

The sequential version shows an almost exponential increase in execution time as the size of the mask increases. This observation can be explained by the fact that the dataset is a two dimensional entity with two dimensioned mask acting on it. The number of arithmetic operations can be estimated as follows:

For an  $n*n$  mask, the number of additions per location is  $n^2$ .  
 For an image of size width\*height,

No. of additions is: width\*height\* $n^2$   
 No. of divisions is: width\*height

Since these operations can be performed simultaneously over all the pixels, the amount of functional parallelism that can be extracted from the problem is high.

This was evident from the fact that when the parallel strategy was applied to the problem, the execution time almost reduced linearly with number of threads as shown in figures 5 and 6. This trend however levels when the number of threads increases beyond what the hardware can manage and tends to become an unnecessary strain on machine resources.

It is observed that the best results are achieved for large size images which are typical of satellite imagery. The speedup achieved can be increased if processors with better thread management mechanisms are used. It was also observed that since the images were stored to RAM and accessed in a row-major manner, the CPU utilization was kept high during the operation. When the mask size was increased, the similar speedup results were achieved.

### V. CONCLUSION AND AREAS OF FUTURE WORK

In any engineering problem, there exists a fine tradeoff which determines whether the extra effort required for solving a problem in a different way warrants the research and work in the area. Failing to answer this question tends to relegate the solution as a purely academic effort.

In the problem under consideration, it was observed that when the image sizes get larger, the execution time advantage extracted from a Multicore computer was higher than what a single core computer can offer. It is hence observed that given the setup time and quantity of resources that require being dedicated to the task, the best results in the form of an execution time advantage is achieved when the data sizes are large. This will warrant the effort required and drastically cut down on unavoidable overheads such as setup time. Also it has been observed in the image processing problem that larger mask sizes tend to take much longer on a single core computer. As the number of threads are increased, the execution time required by the process is reduced but only to a point that the hardware to manage the threads. In advanced hardware computer architectures that superscalar pipelines

and hyper-threading, it would be possible to reduce the execution time to a greater extent allowing for a better performance gains. This serves to prove the fact that no matter how well a parallel computing mechanism is devised, the underlying architecture determines the maximum possible speedup and performance.

Future work in the area must focus on reducing cache misses by specifically profiling the problem to solve memory access delays. Also even larger data sizes must be used to test the strategy. Even more functionally innovative methods for data access can be attempted.

## VI. REFERENCES

1. Class lecture notes of ECGR 6090 – Multicore Computing by Dr. Arun Ravindran (University of North Carolina at Charlotte)
2. POSIX Threads Programming (<https://computing.llnl.gov/tutorials/pthreads>)
3. Pthread Tutorial by Peter C. Chapin (<http://www.morris.umn.edu/~gaoj/Teaching/Spring2007/CSci3401ModelsofComputingSystems/pthreadTutorial.pdf>)
4. Image Smoothing Notes – Technische Universiteit Delft (<http://www.ph.tn.tudelft.nl/Courses/FIP/noframes/fip-Smoothin.html>)
5. Intel OpenCV – Open Source Computer Vision Library (<http://opencvlibrary.sourceforge.net/>)
6. Computer Architecture – A Quantitative Approach (4<sup>th</sup> Edition) by John L. Hennessy and David A. Patterson – Morgan Kaufmann Publishers.