Practical 2 - Static pThreads

Warren 'fletch' Fletcher[†]
EEE4084F Class of 2017
University of Cape Town
South Africa
FLTWR002[†]

***Abstract*—This paper details an investigation into increasing the performance of a fixed-sized medium filter. It focuses on the implementation of a known solution, a 'golden measure' and the possible improvements that can be made through the use of different algorithms, thread count, and space partitioning.**

sectionIntroduction

The aim of this document is to report on the results of Practical 2 for the EEE4084F Course. In this practical, we are tasked with investigating the performance (In this scenario the runtime of the process) of a fixed 9x9 medium filter. We begin by creating a reference point. It was chosen to use an optimized bubblesort algorithm (running a single thread).

The metrics for investigation are:

- Algorithms Used
- Number of Threads used
- Segmentation (shaping) of data

From here we split the report into 2 sections. Section A details with the speeds up due to using alternate algorithms, however fixed to a single thread of execution. Section B details with the improvements that come from using multi-threaded versions of many algorithms. In this section, shape size, thread count and others are discussed. During the investigation attention is given to when and for how long each thread must enter a critical section.

Note the assumptions made for this experiment:

- Performance is measured in terms of speed up against the golden measure.
- No memory considerations are taken into account.
- A minimum performance of improvement of 5% is required for it deemed to be relevant

I. Methodology

A. Hardware

Intel Core i5-5200U ; 8GB RAM ; 256GB SSD

B. Software

Linux Mint 18.1. Kernel 4.4. gcc version 5.4.0

C. Implementation

Below are descriptions of the equations, scripts and tools used to implement each individual experiment. A notable performance improvement is one that is 1.1 better than an alternate version.

D. Test Data

Test data for the medium filters is provided by 4 images of varying sizes. 3 of them where the original images provided within the practical and the final was a 4000x4000 pixel image taken from the popular website reddit. This image represents a perfect square image.

E. General Sorting Methodology

Bubblesort is an algorithm known to be poor in performance. In fact, it is known to be an algorithm who's sole purpose is to be a metric of a bad sorting algorithm. Before we dive into the alternate sorting algorithms that were used. It is important to discuss the implementation of the sorting algorithms used, specifically in reference to an image.

In the JPEG format. a NxM pixel image is unwrapped into an $Nx3M$ integer array. For any pixel located at position (N,M), the Red, Green, and Blue channels can be extracted as $(N, M+0)$, $(N, M+1)$ and $(N, M+2)$ respectively. These values are stored directly after each other in memory. We can leverage the fact that the pre-fetcher will fetch these values when a cache miss occurs. It was actually noted that using this method we can sort all 3 channels in one passing.

F. Golden Measure

The Golden measure was chosen to be an optimized bubblesort. Optimized bubblesorts decrease the running time by 2 compared to the original bubblesort algorithm. This puts the runtime complexity at $O(N^2)/2$ but constant time decreases are ignored in traditional big-oh notation and thus we can still consider the worst case complexity $O(N^2)$.

G. Alternate Sorting Algorithms

The following three sorting algorithms were used.

- Bubble Sort
- Insertion Sort

- std::sort Sort

Insertion sort is a sort that works well on small datasets. For a 9x9 mask, 81 values is small enough. Although, insertion sort is also a worst case $O(N^2)$ algorithm. It's amortized implementation leads to a constant time decrease. It is expected that insertion sort will perform the best when the data set is small.
std::sort is expected to work better on the larger datasets but show more consistent results across the board due to its $O(NlogN)$ scalability

H. Data Partitioning and Critical Section

The design of a medium filter only relies on knowledge of the immediate pixel and the required mask pixels around it. This leads to an embarrassingly parallel method as there is no data dependency on other pixels being processed first or last. From this we completely **remove the need for a critical section** as no thread is dependent on information that another thread is using/will produce.
Secondly, we know that the pre-fetcher will fetch the target location plus an additional page of continuous memory. This is very useful, as the next data will immediately be fetched and will be able to be accessed much quicker than RAM when it is required.
C++ stores 2-Dimensional arrays as Row-Major ordering, this will allow us utalize the ability of the pre-fetcher fully if we traverse the data one row after another as opposed to one column after another.
A third partitioning method was testing by allocating each thread an equal 'block' of the image. This works well for 1:1 images and it is theroized that this will be the most fare distribution of work and will show a performance improvement in 1:1 images.

I. Threaded Implementation

Finally, we repeat each experiment using threaded versions of the original algorithm and compare these values against the golden measure. It is expected that a 2-3x speed up will be experienced and that threading improvements will being to decrease once the number of threads exceeds the number of execution cores available.

II. Results

This section presents and discusses the experiments and their results.

1) Single Thread Execution: Table 1 shows the results of the median filter speeds and Figure 1 shows the images and figure 2 shows the outputs

It was successfully predicted that the insertion sort would be the quickest. This is due to insertion sort operating extremely well in cases where the dataset is small. If the mask was increased and thus the dataset grew. the sort is expected to take exponential longer due it's intrinsic N^2 growth rate. std::sort is under-performing due to the dataset being too small, as the mask gets larger, this sort is expected to dominate in performance.

2) Multi-Threaded Execution: For multi-threaded execution, we get interesting results. for the smaller image (small.jpg) we see that insertion sort performs the best, but that all algorithms got at least a 2x performance increase. As the data set gets larger, we see that std::sort is out performing bubblesort by a significant margin and insertion sort to a lesser extent. This was predicted initially due to the nature of std:sort.m
As we increase the Thread count, we start to notice that the overhead of threads is already becoming significant and that adding more power to the problem is being bottlenecked by the time it takes to create these threads. Again, this was as expected as the time to create the threads when the data set is small (as in small.jpg) becomes significant. In the larger data sets we see sight improvements, but they are negligible. This is due to the fact that the test system is limited to 2 Physical cores at any given time and that new threads will not add any benefit if they are also just waiting.

3) Partitioning Method: Looking at Table V, we see that some improvements are there but are negligible in the long run. For extremely large images, square partitioning shows improvements. but these images are unlikely in normal use.

4) Speed-up vs Golden Standard: Regardless of number of threads, partitioning or algorithm used. It is clear that any multi-threaded implementation offered significant speed up. Looking at Table IV, we see that for small datasets, the insertion sort is faster. It should also be noted that insertion sort works best on partially sorted data or data that does not deviate much which is true with regards to small.jpg. std::sort dominates when the data sets get larger and outperforms Insertion sort constantly.

III. Conclusion

From the experimental results, we can conclude that given a constant size 9x9 median filter. Threading provides a significant improvement in performance. However, this improvement is not linear and simply adding K more threads to the problem will not give K performance gain.
No critical section needs to be considered in this case as the threads act independently to one another.
In selecting thread counts, it is best to match the number of execution cores (Physical + Logical) your system has. Any more threads adds minimal gain and in certain cases can lead to degradation in performance.
Data partitioning was shown to only produce improvements if the partitioning is done intelligently to ensure that each thread receives equal amounts of work to do. Badly portioning data can lead to some thread finishing early and this leads to wasted time. It also resolves to being algorithm specific.
In terms of golden-measure improvements and a fixed 9x9 mask size. The most optimal sort to use is the std::sort with 4 threads. This was shown to produce a speed up of 3.6x on

TABLE I

SINGLE THREADED PERFORMANCE

Image	Bubble Sort(ms)	Insertion Sort(ms)	Std::sort(ms)
small.jpg	2534	1173	2388
fly.jpg	38041	23071	26813
greatwall.jpg	204476	153149	158474
place.jpg	645848	389644	446351

TABLE II

4-THREAD PERFORMANCE

Image	Bubble Sort(ms)	Insertion Sort(ms)	Std::sort(ms)
small.jpg	1498	580	883
fly.jpg	17941	10941	9444
greatwall.jpg	99545	66851	56937
place.jpg	293426	185942	174023

TABLE III

9-THREAD PERFORMANCE

Image	Bubble Sort(ms)	Insertion Sort(ms)	Std::sort(ms)
small.jpg	1145	676	1006
fly.jpg	17866	12493	10413
greatwall.jpg	113309	76838	61375
place.jpg	293426	190034	174023

TABLE IV

SPEED UP TO GOLDEN MEASURE (4 - THREAD)

Image	Bubble Sort(ms)	Insertion Sort(ms)	Std::sort(ms)
small.jpg	1.70	4.37	2.87
fly.jpg	2.12	3.48	4.03
greatwall.jpg	2.05	3.06	3.6
place.jpg	2.2	3.473	3.71

TABLE V

PERFORMANCE ON GREATWALL.JPG FOR DIFFERENT SHAPES

Shape	Bubble Sort(ms)	Insertion Sort(ms)	Std::sort(ms)
Column Order	1497.66	580	883
Row Order	1327	695	1010
Square Order	1269	606	882

average.

Further investigation should be done into the timing of functions by running the experiments in a perfectly unobstructed system with no system cache and only the mandatory minimum background processes running (Systemd, etc). Experiments should also be conducted with varying filter sizes.

Ailsa is a bad ass bitch

Fig. 1. Sample Input Images

Fig. 2. Sample Output Images