Proceedings of the 4th International Conference on Open-Source Systems and Technologies (ICOSST 2010)

Parallel Implementation of Imaging Filters on Multi-Core Processors for Win32 Platform

***Abstract*—This work aims to propose a parallel implementation of imaging filters using a 3x3 kernel on multi-core processors in Win32 environment. This paper presents architecture for parallel processing on two cores by using win32 threads and collection of mutex objects for synchronization. It is shown that for large images, the parallel implementation approaches Amdahl's ideal curve.**

I. Introduction

Current multi-core CPU architectures demand drastic change in programming methods, as a single-thread application uses only a fraction of available processing power. This implies that CPU intensive applications/simulations must be parallelized on multiple cores for full CPU utilization.

One processor-intensive operation is image filtering such as image smoothing or sharpening. With recent advances in digital imaging technology, digital cameras with a resolution of 12 megapixels are commercially available. This paper addresses the problem of parallel image filtering or convolution operations for windows-based platforms. The major contributions of this paper are 1) Solution to thread synchronization problem in Win32, 2) Benchmark of parallel implementation with increased data scalability or O(n) and 3) Optimal number of cores to be used for given data set.

II. Related Work

Many authors have contributed in parallel image processing, especially on Unix/Linux based platforms. In [5], authors implemented parallel multi-threaded implementations of k-means and mean-shift. A JAVA based application [4] is available online for image processing on any JVM.

III. Parallel Programming

Any sequential algorithm targeted on a multi-processor must be re-written so that parts of algorithm executes in parallel. In this work, the paradigm used is recursive decomposition or divide and conquer approach [2], as operations such as filtering are separable by definition. Target image is split into n (no of processor cores) smaller images which are processed in parallel, and concatenated at the final step. This implementation is easily scalable for quad-core and multi-core processors. For quad-core implementation, the image will be split into four smaller images, 3 new threads will be created and the four threads will work on four sub-images in parallel.

Firstly, we deal with synchronization problem i.e. how to alert other threads that one thread has finished successfully. The proposed solution is to use a collection of mutex [1] objects with one mutex for every thread. A mutex is a variable that can be in one of two states: unlocked or locked. The win32 API CreateMutexA is used by a thread locking the mutex object and other threads try to get mutual exclusion after finishing their tasks. In our case, a thread after finishing job releases its mutex by calling CloseHandle API. Then it waits for other threads to release their mutexes using API WaitForSingleObject**,** which returns 1 to indicate some thread has unlocked its mutex. The instant last thread release its mutex, all threads terminate almost instantly except the parent thread. Normally, one or two threads have tendency to finish much earlier so, in theory, they must wait for other threads and finish together. This is explained in pseudo code for synchronization problem given below:

```
mutex sync_obj[no of processors];

void thread(int threadID )
{
  CreateMutexA sync_obj[threadID];
  // threadID is a local thread identifier = 0, 1, …3,…
  //   .
  //   .   actual job here !
  //   .
  // job finished !
  CloseHandle (sync_obj[threadID];
 int check;
 while (1) {
  check=0;
  for (int i=0; i<no of processors; i++)
   if (i!=threadID) check+=WaitForSingleObject(i);
   if  (check== (no of processors-1)) break;
 }
  TerminateThread ( GetCurrentThreadID () );
}
```

IV. TEST BED

For benchmarking purpose, Win32 API GetTickCount is employed which returns system's uptime in milliseconds. A call is made at the start of experiment and another one is made at the end to calculate the computation time. In multi-threaded parallel implementation, the process creates multiple threads and then all threads work in a parallel fashion. If we ignore overheads such as setting up threads i.e. assuming the fraction of code that cannot be parallelized (α) be 0, then by Amdahl's law [3], we can calculate Amdahl's Ideal time for implementation on 'n' multi-cores as follows:

$$Amdahl's\ Ideal\ time = \frac{Time\ for\ single\ thread}{Speedup\ factor} - - (1)$$

$$= \frac{Time\ for\ single\ thread\ T_0}{1/\left(\alpha + \frac{1-\alpha}{no\ of\ processors\ (n)}\right)} = \frac{T_0}{n}$$

This shows that in ideal case, for dual-core implementation, time for parallel implementation should be approximately halved as compared to single-thread implementation.

For image filtering experiments, (table 1) we have some interesting results. For relatively small images (A and B), single-thread implementation is more efficient than parallel implementation. In fact it is shown that parallelism is overkill for small batch jobs. Results are poor for images C and D as well, although parallel implementation takes lesser time but at the cost of full CPU usage. For image E, we have slightly better results and for image F with large dataset (18megapixels), we finally have close to ideal results (figure 1) and difference from Amdahl's ideality is only 3.37 %. This shows that on multi-core machines, we need to make a tradeoff between single-threaded, partially multi-threaded and fully multi-threaded versions. By partially multi-threaded version, it is meant that number of threads running is less than number of cores available.

If we assume, total time T (sec) is a function of input dataset size A (total pixels) and processor speed C (average pixels processed/sec), it can be empirically described as:

$$T = A/(C.n) + b.n$$

, where n is number of processor cores used and b is a constant to model extra time to delegate independent sub-problems to different cores. Find dT/dn and setting to 0 yields, $n = \sqrt{A/Cb}$ which suggests roughly how many processor cores should be used to solve this problem.

A quick solution can be formulated by inspecting table 1. Image C gives us approximately a borderline case when performance of single-thread is approximately equal to that of multi-threaded one. So a condition is introduced in the program to delegate the job to 2 cores only if image size is greater than 2160x1082=2.3 million pixels!

Table 1. Results for image filtering

	Mean Time for 10 trials (sec)					
Image #	A 610 x 391	B 960 x 312	C 2160 x 1082	D 3200 x 1200	E 4096 x 1536	F 8000 x 2248
Single-Thread implementation (Monolithic Sequential Programming)	0.484	0.593	1.275	1.891	3.844	10.328
Parallel implementation (Multi-threaded programming)	0.516	0.595	1.250	1.609	2.687	5.344
Amdahl's Ideal time (time for single thread/2)	0.242	0.297	0.638	0.946	1.922	5.164
% difference between parallel implementation and Amdahl's time	53.10	50.08	48.96	41.21	28.47	3.37
Optimal Number of Cores to use	1	1	2	2	2	2

The platform for experimentation is Microsoft VC++ 6.0 IDE on Win XP SP3, 1.66GHz Core2Duo machines.

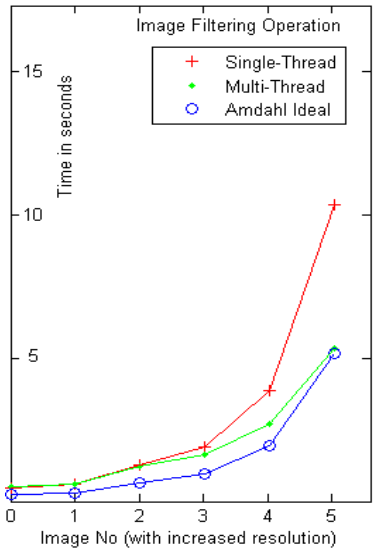

Figure 1. Plot of Computational Performance

CONCLUSION

Parallel implementation using multi-threads result in a close-to-ideal implementation only for large data scalability. For small batch jobs, parallelism should be avoided. However the number of processor cores to be used, as a function of dataset size remains an open problem.

REFERENCES

[1] Andrew S. Tanenbaum and Albert S. Woodhull. Operating Systems Design and Implementation, 3rd edition, USA/ Prentice Hall. 2006.

[2] G. Wilson. Parallel Programming for Scientists and Engineers. MIT Press. Cambridge. MA. 1995.

[3] Amdahl, Gene. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. AFIPS Conference Proceedings. Pg 483-485.

[4] ImageJ. http://rsb.info.nih.gov/ij/.

[5] Honggang Wang, Jide Zhao, Hongguang Li, Jianguo Wang. Parallel Clustering Algorithms for Image Processing on Multi-Core CPUs. International Conference on Computer Science and Software Engineering. 2008.