International Journal of Computer Applications (0975 – 8887)
Volume 110 – No. 1, January 2015

Effective use of Multi-Core Architecture through Multi-Threading towards Computation Intensive Signal Processing Applications

Prathmesh Deshmukh
BRACT'S Vishwakarma Institute of Information Technology
Kondhawa, Pune-48
Maharashtra (India)

Akhil Kurup
BRACT'S Vishwakarma Institute of Information Technology
Kondhawa, Pune-48
Maharashtra (India)

Shailesh.V.Kulkarni
BRACT'S Vishwakarma Institute of Information Technology
Kondhawa, Pune-48
Maharashtra (India)

ABSTRACT
With the advent of Multicore architecture availability, exploiting parallelism is posing certain trends and tides for application deployment. Earlier approaches to explore parallelism in applications were limited to either instruction level parallelism (ILP) or use of architectural redundant resources. In this paper, we attempted to use multicore processor to demonstrate the speedup in compute intensive tasks such as Convolution, primitive to most Digital Signal Processing algorithms.

Further result of multithreaded application on Multicore processor compared with single core is demonstrated for lower and upper limit of granularity for application fragmentation. This work suggests the need for design of memory manager for Multithreading to exploit it more effectively.

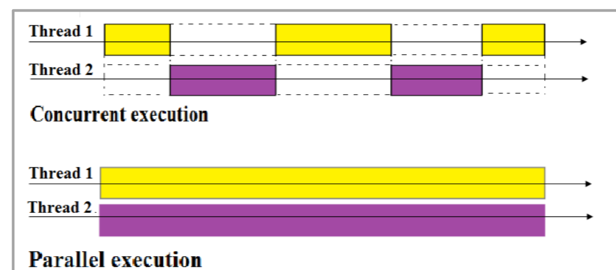
General Terms
Multithreading multicore architectures

Keywords
Multithreading, multicore, granularity

1. INTRODUCTION
In the present Signal processing scenario, large computations are required that makes number crunching processes to be cumbersome. As signal processing algorithms continues to evolve, faster and more capable systems have to be developed. Increasing the clock speed to get a boost in performance seems to be an easy solution to this problem. However, to facilitate timely executions, more than one task may need to be performed simultaneously. In 1965 Gordon Moore stated that the number of transistors on a chip will double roughly every two years. Due to advances in circuit technology and performance limitation in wide-issue, super-speculative processors, Chip-Multiprocessors (CMP) or multi-core technology has become the mainstream in CPU designs [1]. Application should be designed and deployed to explore the inherent parallelism present in hardware. In 2007 Intel released a statement stating that software also has to start following Moore's Law; the software has to double the amount of parallelism that it can support every two years [2], so as to bridge the gap between hardware acceleration and software development. Since increase in silicon estate on chip increases, number of cores per processor is expected to scale. This demand for software to strive in order to attain maximum core utilization. Three commonly used methods for execution performance improvement reported in [8] are number of core increase, cache increase and multithreading. Multithreaded applications will fundamentally advances throughput of multi-core processors. Presently developers are in a trend to exploit multithreading on multi core architecture [9]. Multi-Core processors allow multiple executions of tasks to transpire at the same time which enhances the efficiency in terms of computation time in a multi-tasking, multi-processing environment. Multi-threading facilitates and parallel algorithm enhances the performance of multiprocessor architecture. Fragmenting an application into threads so as to run them in parallel is known as Threading for Performance [3]. Parallelism involves the use of more than one processor core to put into effect the desired result. A thread is a sequential flow of control within a process that enhances its performance and responsiveness [3]. However, if used erroneously, the overhead of threading can degrade the performance and introduce instability and unpredictability [4].

By compiler intelligence, a sequential execution can be made to run concurrently. Concurrency facilitates more than one execution to be in progress, however not simultaneously. Parallelism on the other hand, is the execution of threads at the same time on different cores of the same CPU. Figure 1 illustrates the timeline execution of the above methods. In order to explore multi-threading, we have experimented and presented the results in subsequent sections. Our results can prove perennial for those willing to march onto the path of multi-threading approach on multicore system.

1.1 Threads: Benefits and Pitfalls
Threads are lightweight processes. As a result of lightweight process, thread based deployments is benefited with less overhead requirement as opposed to the launch and termination of processes. Managing a thread requires fewer system resources as opposed to a process [5]. The implementation of threaded programming logic can be influential on many fronts. Following are few of the merits that call for multi-threaded programming.



"Figure 1": Concurrent vs. Parallel execution of threads

- All threads share a common address space and hence potential threat of memory leakages can be avoided.
- Better control over Parallel execution of a task.
- Significant performance gain for large volume of data as compared to small data set.
- Increased efficiency in terms of system resource utilization.

In a processor with multiple cores, each core should perceive the memory as a monolithic array, shared by all the cores. Failing to do so will give rise to the Cache Coherence Problem which will result into data inconsistency across the core caches [5]. Some of the other common pitfalls are:

- Potential threat of landing into a deadlock.
- Starvation of threads.
- Context Switch timing is non-deterministic.
- Generalized increase in programming complexity and difficulty in debugging.

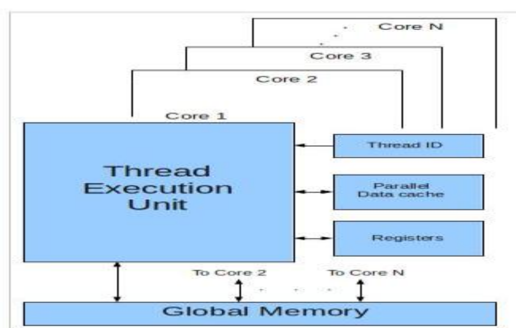
1.2 Multithreading on Multicore

The use of multiple cores on a single chip provides an upper hand in terms of raw processing power. Multi-Threaded Multi-Core Programming (MTMCP) is the way to achieve computing speedup [6]. Concurrent threads can execute on a single processor but true Parallelism requires multiple cores to be used in synchronized manner.

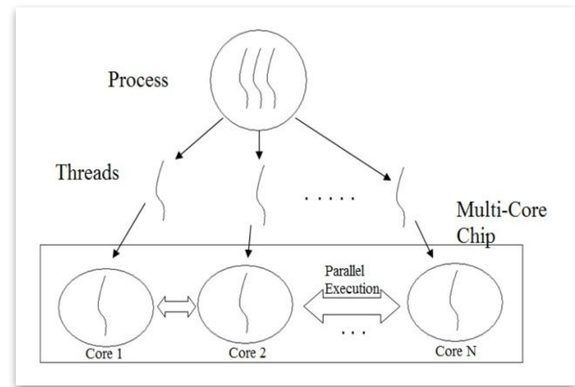
Figure 2 shows the architecture of a generic multi-core CPU. Multiple cores are present in one single package, with each core having its own execution unit. This unit can execute a single thread at one time. In addition to this execution unit, it also has a data cache that works in parallel with the other core caches and the related registers. All these cores share a common pool of data memory known as the Global Memory

Figure 2 shows the architecture of a generic multi-core CPU. Multiple cores are present in one single package, with each core having its own execution unit. This unit can execute a single thread at one time. In addition to this execution unit, it also has a data cache that works in parallel with the other core caches and the related registers. All these cores share a common pool of data memory known as the Global Memory.

As illustrated in Figure 3, a process can be fragmented into smaller, more easily operable modules. A thread can be assigned to each of these modules which can be scheduled to execute on different cores. After the desired output is achieved, they can be recombined, hence enhancing the performance by completing the tasks simultaneously.



"Figure 2": A Multi-Core CPU



"Figure 3": Parallel processing of Threads on multiple cores of the same chip

2. MULTITHREADED APPROACH

The world today depends upon Digital Signal Processing (DSP) algorithms in almost every domain. DSP is fast becoming one of the highest-volume applications, demanding high computation power. It is easier to develop a framework on General Purpose Processor (GPP's) as compared to DSP's because of the availability of prototyped and well tested software and Integrated Development Environment. As processor technology improves, number of cores in a processor keep increasing, which makes their use in Number-Crunching Signal-Processing applications more time efficient by segregating the task into small fragments and allocating them onto different cores.

2.1 POSIX P- Threads

Portable Operating System Interface (POSIX) includes a standard library that defines Application Programming Interface (API) for thread-based distributed processing on multi-core systems to achieve fine grain parallelism. A programmer has to create threads manually and explicitly assign tasks to each thread. The programmer decides how many threads will be spawned, how they will be invoked and how they will terminate. This ensures that the programmer, at all times, has control over the flow of execution.

2.2 Granularity

In the execution of a program, each routine is serially completed first before starting the second, and completely finishing the second before starting the third. However, if the order in which the first two routines execute doesn't affect the third, the third routine can be executed. This property of a program that statements can be executed in any order without changing the result is called potential parallelism. Tasks such as Overlapping of I/O's, printing (display) of results, addition/ multiplication can be carried out simultaneously.

Granularity is the degree to which a process can be fragmented into threads. There is data-dependency incurred in sub-divisions of a process that introduces overheads. Programmer should pay enough heed to write a balanced application such that the overheads are accounted for and there is merit in parallel computation.

3. IMPLEMENTATION DETAILS

Linear Convolution mathematically describes the relationship between input and output signals of Linear Time Invariant (LTI) systems [7]. It involves multiplication of the first signal with the shifted version of the inverted second signal. The total number of multiplications to be performed equals the

product of the size of the input data sets and the number of summations to be performed equals the product of the size of the input data sets minus two. This makes linear convolution computation intensive for a large input data set, which is why we have chosen convolution operation to exploit multi-threading into the signal processing primitive such as convolution.

Algorithm implemented is as follows:

START

Step 1: Generating two input data sets and storing them as matrices.

Step 2: Padding zero's to the input matrices so as to satisfy the dimensions of the resultant convolution matrix.

Size of convolution matrix = (size of first input matrix + size of second input matrix – 1) ... (1)

Step 3: Performing matrix multiplication of the two matrices.

Step 4: Addition of the time-shifted matrix elements.

Step 5: Displaying the resultant matrix.

END

Implementation was performed on the following platform:

System: Dell XPS L502X with Intel® Core™ i52410M CPU @ 2.30GHz x 4

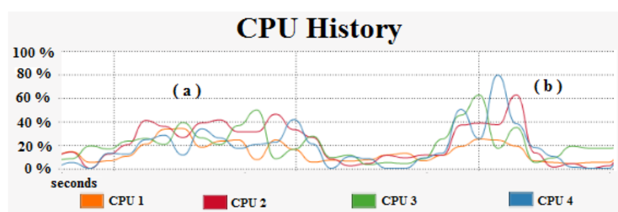
Operating System: Ubuntu 12.04.3 LTS 64-bit

Kernel: 3.8.0-29-generic

Compiler: **gcc** v**4.6.3**: to compile and link '.c' file

Thread model: POSIX p-thread

Generation of input data sets, Padding of Zero's and performing matrix-multiplication were explicitly assigned different threads taking into account their dependencies. This resulted into parallel execution of the above steps which can be monitored from the CPU usage.Figure 4 (a) shows the CPU usage during sequential execution and Figure 4 (b) shows parallel execution of the same algorithm.



"Figure 4": CPU usage during execution of code displayed in a graphical format

During sequential execution the CPU requires a long time and the core utilization is less (45 – 50 %).Whereas, in parallel execution the time required is less and the core utilization is also greater (~ 80 %). This evidences parallel usage of cores and hence demonstrates faster execution.

The command 'time' was appended to the executable on the Linux shell prompt to return the execution time for the code under evaluation. The execution time of various trials has been tabulated in table 1.

4. RESULT DISCUSSION

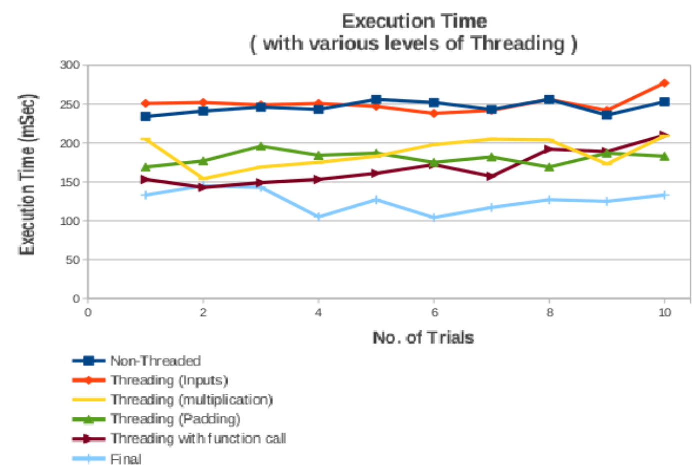
The synthetic data set generated consisted of 145 elements. The execution time for various levels of granularity for 10 trials can be observed in Table 1.

"Table 1": The Execution time observed for various levels of threading

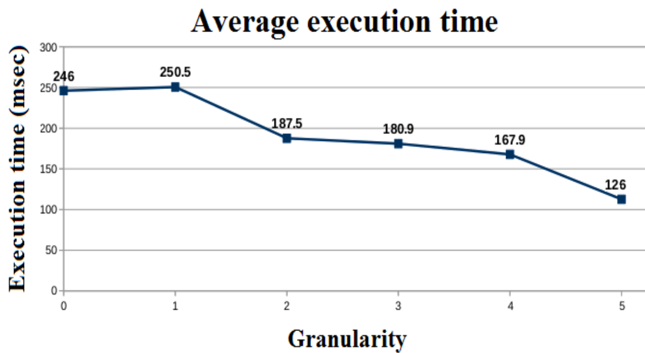
Trials	Non-Threaded	Threading (Inputs)	Threading (Multiplication)	Threading (Padding)	Threading (Function call)	Final
1	234	251	205	169	153	133
2	241	253	154	177	143	145
3	246	249	169	196	149	143
4	243	251	175	184	153	105
5	256	247	183	187	161	127
6	252	238	198	175	172	104
7	243	242	205	182	157	117
8	256	256	204	169	192	127
9	236	242	173	187	189	125
10	253	277	209	183	210	133
Avg:	**246**	**250.5**	**187.5**	**180.9**	**167.9**	**125.9**

"Table 1"also shows that the average time required for execution keeps on reducing as the granularity increases. A plot of these average execution times is shown in "Figure 6". A plot of these values depicts a scenario as shown in Figure.5, wherein the final execution time is minimal for almost every trial.A plot of these values depicts a scenario as shown in Figure.5, wherein the final execution time is minimal for almost every trial.

Initially, the average time taken to execute non-threaded code was observed to be 246msec. Upon increasing the number of threads, the execution time shows a gradual decrease with the exception of first instance of thread usage. This is because the overheads overwhelm the merit of parallelism. The final execution time was a mere 125.9msec which is 1.95 times faster than the sequential execution, which took 121msec more.



"Figure 5": Graphical representation showing the variation in execution time with levels of threading.



"Figure 6": Average execution time for various levels of granularity.

4.1 Analysis towards Speed Gain

The results from the Table 1 were used for calculation of efficiency in terms of execution time.

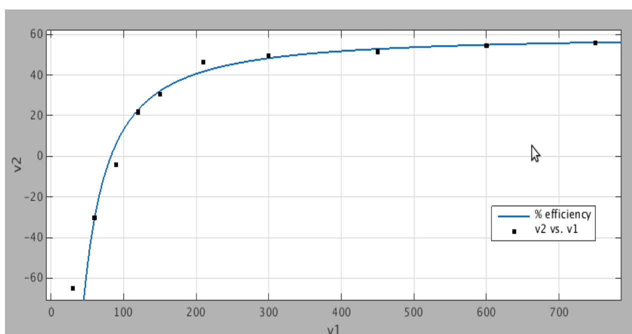
$$\% \text{ efficiency} = \frac{\text{Speedup}}{\text{Time for execution of sequential code}} \times 100\ \% \quad \dots (2)$$

Where:

$$\text{Speedup} = \text{Time for execution of sequential code} - \text{Time for execution of threaded code} \quad \dots (3)$$

A plot of these efficiencies with respect to size of input data set is shown in "Figure 7".The data set was increased from 30 elements all the way up to 750 elements. Unfortunately, as the data size was increased above 750 elements, the program continually crashed. This could be indicative of the fact that memory manager was not able to allocate as much memory required for its operation.

A small data set proved to be inefficient because the time required to execute parallel code was greater than that of sequential code. This shows that overheads are high for a small data set suggesting that threads should be used only for a large data so as to make parallelism fruitful. As the data set was increased, the efficiency also kept increasing until the efficiency plot saturates at about 58%. A mathematical model illustrated by the best fit curve using the 'cftool 'of MATLAB® is shown in figure 6. The curve is governed by a power function as shown in (4) which calculates efficiency.



"Figure 7": A Plot showing variation of efficiency.

$$\% \text{ efficiency} = -18276 \times (x^{-1.298} + 59.367) \quad \dots (4)$$

(Where ' x ' is the size of input data set)

The root mean square error (RMSE) while fitting the curve was observed to be 5.98.

The ability of the above stated model to achieve parallelism can be obtained using equation (4). The user can specify the size of input data set and the % efficiency can be calculated with a tolerance of 4% from which the user can decide whether or not to use this model.

5. CONCLUSION

In this paper, we have demonstrated the use of multithreading on multicore processor to achieve speedup in compute intensive tasks. Using implementation of linear convolution, increase in core utilization of almost 80% was achieved and subsequently time required for execution found to be reduced. The extent to which an application can be granulated to benefit the multithreading for maximum efficiency of resources is also demonstrated and upper and lower limit of fragmentation granularity is tested.

This lower limit can be associated to the increased overheads incurred in threading, which led to a decrease in efficiency. The upper limit is indicative of the limitation of the current memory manager to incorporate multiple threads urging the need for a memory manager designed towards multithreading.

Above experimentation and result discussion is made considering convolution as the DSP primitive operation. However, we aim to apply above framework for more compute intensive signal processing applications such as motion-estimation algorithms in the video processing domain.

6. ACKNOWLEDGMENT

The authors express their profound gratitude to the department of Electronics & Tele-communication, VIIT Pune, for its continued interest and support in this research. The authors are also thankful to Harshal Waghmare, VIIT Pune for discussion on curve-fitting using MATLAB®.

7. REFERENCES

[1] L. Peng et al, "Memory Performance and Scalability of Intel's and AMD's Dual-Core Processors: A Case Study", IEEE, 2007.

[2] T. Holwerda, "Intel: Software Needs to Heed Moore's Law",http://www.osnews.com/story/17983/Intel-Software-Needs-to-Heed-Moores-Law/

[3] Student Guide, "Multi-Core Programming For Windows", Intel Corp 2006.

[4] Student Handout, "POSIX* Threading API Quick Reference", Intel Corp 2006

[5] Ananth Grama, George Karypis, Vipin Gupta, Anshul Kumar, "Introduction to Parallel Computing, 2nd edition", Pearson publication

[6] Georgios Kornaros, "Multi-Core Embedded Systems", CRC Press

[7] Simon Haykin, Barry Van Veen, "Signals and Systems, 2nd edition", John Wiley and Sons.

[8] Massimiliano Meneghin, et.al,"Performance evaluation of inter-thread communication mechanisms on multicore/ multithreaded architectures", IBM technical paper

[9] Jun Yan, Wei Zhang, **"**Hybrid Multi-Core Architecture for Boosting Single-Threaded Performance", ACM SIGARCH Computer Architecture News, , Vol. 35, No. 1, March 2007, pp 141-148.