International Journal of Engineering Research and Development
e-ISSN: 2278-067X, p-ISSN: 2278-800X, www.ijerd.com
Volume 11, Issue 07 (July 2015), PP.08-15

Exploiting Multi Core Architectures for Process Speed Up

Deepika Roy[1], N. Saikumari[2], Ch Umakanth[3], M. Manju Sarma[4], B.Lakshmi[5]
National Remote Sensing Centre, Indian Space Research Organization, India

Abstract:- As a part of the Indian Space program, Indian Space Research Organization has been developing and deploying Indian Remote Sensing (IRS) satellites. National Remote Sensing Centre (NRSC) is responsible for acquiring, processing and disseminating the data products for different users. Pre-processing of satellite data is one of the important steps in satellite data processing. It involves decoding, decompressing and reformatting of the satellite data so that it can be given in an appropriate form to next level of processing. The pre-processing step is time consuming and needs to be optimized to deliver the data products with minimum turnaround time. This paper presents the design and performance analysis of a dynamically scalable multithreaded process

Keywords:- Multi CPU, Multi Core, Multithreading, Data Parallelism, Performance Speedup

I. INTRODUCTION

The growing application needs of remote sensing data demand higher spatial, spectral and temporal resolutions, thus increasing the volumes of data. Added to this, the demand for near real-time supply of data has increased the need for faster and faster processing. Hence improving process execution times has become an important concern.

Earlier the servers used for data processing were single core machines. Saturation of clock speed of single core processors, made CMP (Chip Multi Processors) as the main stream in CPU designs [1]. The multiple cores in a processor run at a lower frequency but result in speedup as all cores can be executed in parallel. To exploit the full capacity of multiple cores the software should be multithreaded which is achieved by splitting the workload to different cores [2]. This introduced the multithreaded programming paradigm.

The applications which use massive data sets are termed as Data Intensive [5]. Due to large amount of data to be processed, the data intensive applications are I/O bound and hence there is considerable amount of CPU idle time which can be utilized by overlapping processing and I/O threads. The number of threads per processor are to be optimum so that the advantages of multithreading are not overshadowed by the overhead involved in threads creation and synchronization.

The applications where complex computations are involved are termed as Compute intensive [5]. Due to heavy utilization of processor these are mostly CPU bound. In case of such applications, it is apt to run one thread per processor. More than one thread per processor will result in unwanted overhead where time will be wasted in scheduling the threads and context switching.

The independent frame/block wise format of satellite downlink data makes it a suitable candidate for multithreaded implementation using data parallelism.

II. PARALLEL COMPUTING DESIGN METHODOLOGY

Parallel computing is a form of computation in which calculations are carried out simultaneously. There are several different levels of parallel computing: bit-level, instruction level, data, and task. The form of parallelism implemented in this paper is SIMD (single instruction multiple data).

The sequential programming method can utilize only one core resulting in underutilization of the multi core architecture.

In Multi Threaded application, a large data set is broken into multiple subsets and each is assigned to an individual core. After processing is complete, these subsets are recombined into a single output. This implementation framework yields high performance by efficiently utilizing multiple cores in multi CPU multi core architecture.

III. MULTITHREADING MODEL

The data parallelism model employed here is a variant of Master – Slave (Boss-Worker) model [3]. In this design, two threads function as boss, one to assign tasks and the other to control the worker threads. The first boss thread determines the number of worker threads, the data/work partitioning and assigning the jobs to each worker thread [4].

The number of worker threads are determined during run time based on number of free processors/cores and memory. As the application is designed to run in time sharing environment memory usage is to be optimum to avoid swapping.

The first boss thread also sets up various parameters for the worker threads. It initializes information for each thread like the thread number, data subset to be processed, the output file and location.

Each worker thread performs the same task on a different data partition. Instead of keeping total subset data in the memory, the worker thread runs in a loop for predefined blocks of the subset. This ensures optimal utilization of memory resource. The second boss thread waits for the completion of all the worker threads, consolidates the output result and performs the final cleaning up.

The execution of the worker thread is controlled using semaphores. One semaphore is used to control the group of worker threads to reduce the overhead. Semaphore is initialized to the number of worker threads in that group and decremented each time a new worker thread is created. The other semaphore is used to control the joining of the worker threads.

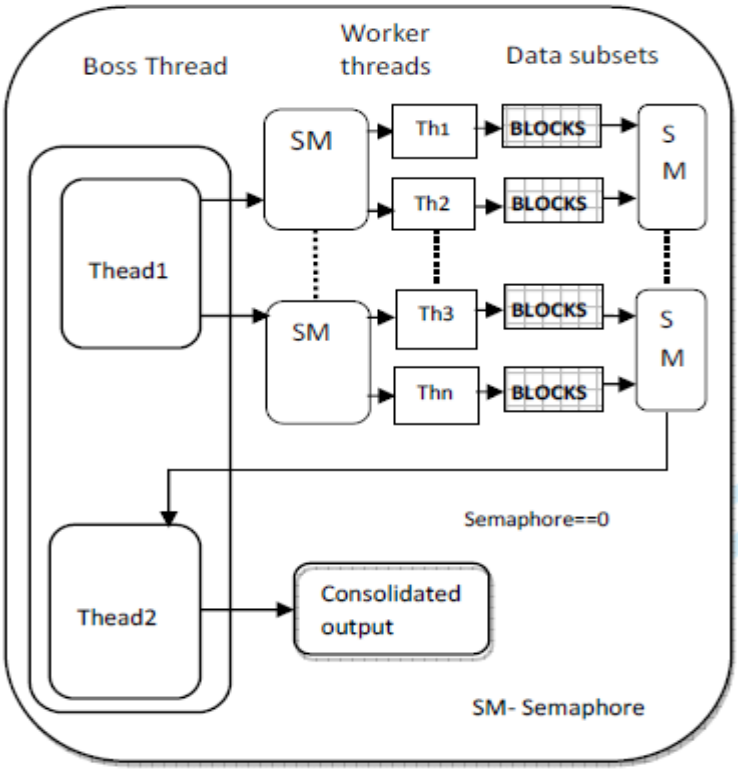

Fig 1: Boss –Worker Thread Model

IV. APPLICATION

Parallel computing based on Data Parallelism has been applied for Level-0 data processing of recent IRS satellites such as Oceansat-2, Cartosat-2B, Resourcesat-2 and RISAT-1. The Level-0 processing of IRS satellite data involves various steps for pre-processing of raw data like extraction of auxiliary data, decoding, decompression and reformatting of raw satellite data to make it suitable for data product generation.

A typical application would involve reading of a data partition from disk(I/O operation), processing it(CPU bound processing) and then writing that processed data back to the disk(I/O operation). Thus we have to decide the number of threads per processor such that the I/O operation of one thread can be interleaved with the CPU bound processing of other thread. So minimum number of threads assigned to each processor is 2. The number of threads per processing unit can be increased based on the CPU idle time.

The following processes are implemented using multithreaded approach for faster processing.

A. Auxiliary Data Extraction Process

IRS data is received as raw frames each consisting of auxiliary data followed by video data. Auxiliary data gives the details to interpret the actual image data. It contains information about compression modes, imaging modes, sensor modes, information about payload health and position. It is necessary to extract this auxiliary data for further processing of image data.

This process extracts the auxiliary data from every frame of the raw data and writes it into a separate file. The similarity in format of raw data frames makes it convenient to partition the data among threads. Each thread then performs similar processing on the data partition assigned to it.

Each thread is provided with the start and end line of its data partition. All threads simultaneously access the raw file for reading at a predefined offset. The extracted auxiliary data is also simultaneously written into the output file at a predefined offset position. This simultaneous reading and writing to the same raw file is done using the system calls provided by the POSIX threads library.

B. Reed Solomon Decoding

The IRS satellite data is encoded on board using Reed Solomon Encoding scheme. Encoding helps in detecting and correcting errors that occur during data transmission and acquisition. RS Encoding scheme used here is RS (255,247). In this type of encoding, for every 247 symbols 8 parity symbols are appended and sent as a block of 255 symbols identifying it as a RS block. On ground the 8 parity symbols are used for error detection and correction of the 247 symbols.

Each RS block requires similar operation to be performed. Each frame of raw data contains many RS blocks. This gives scope for parallel computing. So data partitioning between threads is done based on frames.

The data partitioning scheme has to ensure that there is no dependency between the threads while handling different satellite data. All threads decode their data subset and write the decoded data simultaneously in the output file at their designated offsets. Each thread also calculates the number of errors found. In the end the joining boss thread combines these counters of each thread and gives a single value for determining the errors in the raw data.

C. Decompression

IRS satellites apply compression to the payload data to reduce the downlink data rate. Many types of compression schemes have been used based on the imaging and payload type. Some examples of these compression schemes are Differential Pulse Code Modulation (DPCM), Multi Linear Gain (MLG) and JPEG like compression. The concept of data partitioning and multithreading is employed here for exploiting multi-core architectures.

V. IMPLEMENTATION

The application is implemented using POSIX library for threads and C programming language on LINUX platform. The POSIX thread library, IEEE POSIX 1003.1c standard, is a standards based thread API for C/C++ [6]. Pthreads is a shared memory programming model where parallelism takes the form of parallel function invocations by threads which can access shared global data. Pthreads defines a set of C programming language types, functions and constants. It allows one to spawn a new concurrent process flow. It is most effective on multi-processor or multi-core systems where the process flow can be scheduled to run on another processor thus gaining speed through parallel or distributed processing.

There are other multithreading models available like compiler based OpenMP and Cilk. Our application uses POSIX threads as it gives a low level control of thread creation, scheduling, joining, synchronization, communication and more flexibility in program design. Even though programming with Pthreads needs considerable threading-specific code, the finer grained control over threading applications provided by its greater range of primitive functions makes POSIX threads the natural choice for our software[7]. Synchronization between threads is achieved using POSIX semaphores.

This multithreaded design is generic in nature and can be easily extended to other applications for IRS satellite level-0 processing where there is a scope of data parallelism.

This design will work on any system with multi core architecture and will be scaled up/down dynamically based on the system architecture which in turn will determine the level of parallelism that can be achieved.

VI. DEPLOYMENT

The application is deployed on different configurations and it scales itself up or down dynamically based on the processing units and memory available.

Deployment is done on Intel and AMD processors without any modifications in the software as it is POSIX complaint and hence easily portable. It is deployed on architectures such as: Intel Xeon 4 CPU 4 Core server with 16GB of memory and Red Hat Enterprise Linux version 5.3, Intel Itanium (IA-64) 4 CPU single core server with 4GB of memory on RHEL version 4, Intel Xeon 4 CPU 4 Core server with 16GB of memory and Red Hat Enterprise Linux version 5.3 and AMD Opteron 4 CPU 8 Core server with 32 GB of memory and RHEL 5.5.

VII. PERFORMANCE EVALUATION

A. Timing Analysis

The timing analysis was done on two different types of applications. Time taken to execute the sequential and parallel applications was examined on 4 CPU 8 cores server (32 cores in total). The timing analysis for these two test cases is as follows.

1) Test case 1- Reed Solomon Decoding

In the first test case, Reed Solomon decoding was run on 6.3 GB of raw data.
The sequential application took 403 seconds to execute.
The following table lists the time taken by the multithreaded data parallel application with varying number of threads per core

TABLE I: Time taken to run test case 1 with different number of threads

Number of threads per core	Total number of threads	**Time Taken(seconds)**
1	32	72
2	64	70
3	96	65
4	128	63
5	160	63
6	192	60
7	224	50
8	256	60
9	288	68
10	320	70

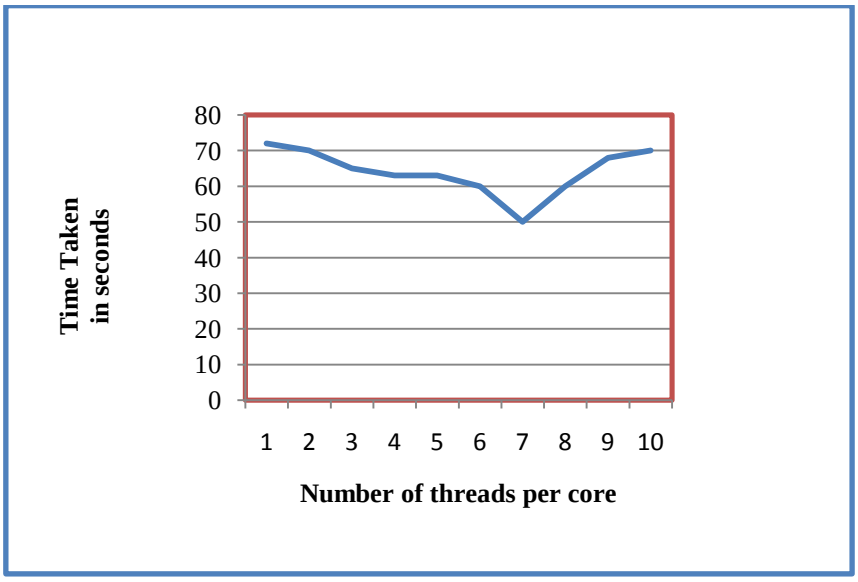

Fig 2: Number of threads per core vs. time for Test Case 1

2) *Test Case 2- Reed Solomon Decoding with DPCM Decompression*

In second test case, the application performed both Reed Solomon decoding and Differential Pulse Code Modulation Decompression on 9.6 GB of raw data.
The sequential application took 985 seconds to execute.

The following table lists the time taken by the multithreaded data parallel application with varying number of threads per core

TABLE II: TIME TAKEN TO RUN THE TEST CASE 2 WITH DIFFERENT NUMBER OF THREADS

Number of threads per core	Total number of threads	**Time Taken(seconds)**
1	32	720
2	64	362
3	96	343
4	128	264
5	160	192
6	192	138
7	224	132
8	256	317
9	288	358
10	320	390

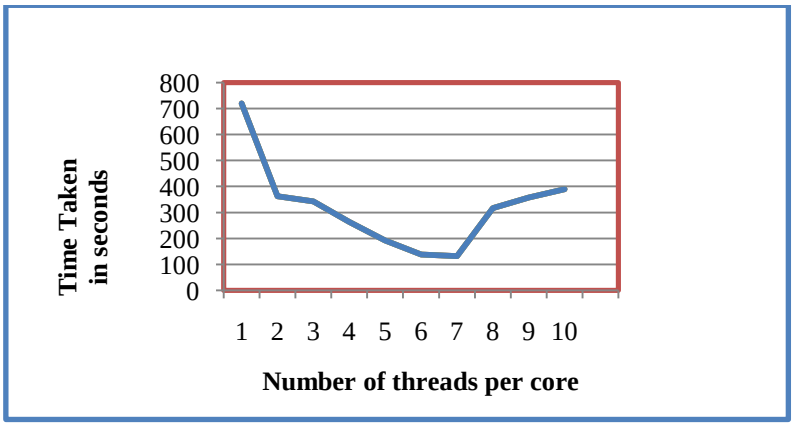

Fig 3: Number of threads per core vs. time for Test Case 2

It can be observed from the graphs shown in Figure 2 and Figure 3 that the maximum speedup is achieved when 7 threads are assigned to each core i.e. when 224 threads are created in total. It is also observed that if the number of threads per core increases beyond 7 the execution time starts increasing. This can be attributed to the fact that more the number of threads, more time is spent in thread overheads like thread creation, synchronization, scheduling and swapping and thus performance speedup reduces. Hence the number of threads should be created based on the nature of the application. Increasing the number of threads may not lead to increasing the speed up after certain limit and may also cause performance degradation.

B. Speedup Calculation

The speedup achieved by parallelizing the application can be calculated as:

$$Speedup = \frac{T(1)}{T(N)} \qquad (1)$$

Where
T(1) : is the time required to run the serial code on a single processor
T(N): time required to run the parallel code on N processors

Here T(1) contains the time for the serial version T_s and the parallelizable portion T_p of the code. Hence T(1) = $T_s + T_p$.

Theoretically, if a task is split among N processors, it should complete in 1/N of the serial time, giving a speedup of N. However in real world scenario, there will always be some tasks that need to run serially and cannot be parallelized. These serial tasks will not run any faster on a multi core system and will take the same time as taken on a single processor system. Thus the time spent in executing these serial tasks will limit the total speedup of the parallel application.

These limitations are taken into account to some extent by Amdahl's Law, which states that the speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program. So speedup according to this law can be calculated as:

$$S(N) = \frac{T(1)}{T(N)} = \frac{Ts+Tp}{Ts+\left(\frac{Tp}{N}\right)} \qquad (2)$$

Where

T_s : Time taken by the serial part of the code

T_p : Time taken by the parallelizable part of the code

N: Number of processing units (currently 32)

For test case 1,

T(1) = 403 seconds

T_s = 7 seconds

T_p = 396 seconds

N = 32

Substituting these values in the equation 2, we get the speedup factor according to Amdahl's Law for test case 1 as 20.

For test case 2,

T(1) = 985 seconds

T_s = 14 seconds

T_p = 971 seconds

N = 32

Substituting these values in the equation 2, we get the speedup factor according to Amdahl's Law for test case 2 as 22.

However the speedup factors thus calculated for both the test cases according to Amdahl's Law gives far too optimistic values. This is because it ignores the overhead incurred due to parallelizing the code.

A more realistic calculation of speedup would be to include the parallelizing overhead too [8]. So the following terms are also included in calculating the speedup:

- T_{is}: The (average) additional serial time spent doing things like inter thread communications, thread creation and setup, and so forth in all parallelized tasks. This time can depend on N, the number of processors, in a variety of ways, but the simplest assumption is that each thread has to expend this much time, one after the other, so that the total additional serial time is N* T_{is}. In our application the overhead time taken for thread creation, setting up of initial environment and inputs for threads and thread synchronization takes 9.8 seconds for test case 1 and 38 seconds for test case 2.
- T_{ip}: The (average) additional time spent by each processor doing just the setup and work that it does in parallel. This may well include the idle time, which is often important enough to be accounted for separately. This time amounts to 2 usecs per thread.

By considering the above factors, the expected speed up can be calculated as:

$$Speedup = \frac{Ts + Tp}{Ts + N * Tis + \left(\frac{Tp}{N}\right) + Tip} \qquad (3)$$

For test case 1, substituting the above two overhead times in equation 3, we get the calculated speedup as 13. The real maximum speedup achieved for test case 1 at 7 threads per processor using equation 1 and Table 1 is 8.

For test case 2, substituting the above two overhead times in equation 3, we get the calculated speedup as 12. The real maximum speedup achieved for test case 2 at 7 threads per processor using equation 1 and Table 2 is 7.

It is observed from both the test cases that the actual maximum speedup achieved is close to but less than the calculated speedup. This can be attributed to the fact that some time is also spent in merging the outputs of threads. Some variable delays also happen due to thread scheduling which is dependent on the operating system and beyond the control of the application. Hence the achieved speedup is in accordance to the expected values for our application.

C. CPU Performance Analysis

CPU performance was analysed on an Intel Xeon 4 CPU single core server with Hyper-Threading enabled. Figure 4 shows the CPU utilization by the sequential application. It is observed that only one CPU shows maximum performance at one instance of time. The other processing nodes are mostly idle. The memory utilization is also very less as only 174 MB of 4GB is used.

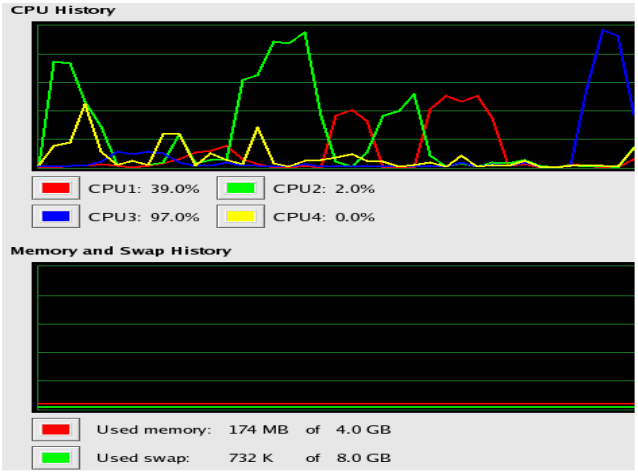

Fig 4: CPU & Memory utilization - sequential processing

Figure 5 shows the performance of multithreaded application on the same server.

Fig 5: CPU & Memory utilization - multithreaded processing

As the application starts and threads are created, all the CPUs show peak (100%) performance. In this application 7 threads are running in parallel. This indicates that all the cores are being used to their maximum performance. The memory profile also shows that 1.9 GB of 4GB available memory is being used. Thus the multithreaded application efficiently utilizes all the available resources of processing nodes and memory to speed up the data processing.

VIII. CONCLUSION

The speedup achieved for test case 1 with raw data size of 6.3 GB is 8, the theoretical calculated speedup factor being 13. For test case 2 with 9.6 GB of raw data, the achieved speed up factor is 7 and the theoretical calculated speedup factor was 12.The achieved speedup in both the test cases is very close to the expected values taking into account some variable delays in threading overheads when 7 threads are assigned to each core of a 4 CPU 8 core server.

The speed up achieved has resulted in considerable reduction of Turn Around Time for data products generation thereby realizing the required delivery timelines. The application design is generic and can be easily configured for new applications for future IRS satellite missions thereby reducing the implementation and testing times.

ACKNOWLEDGEMENTS

The authors extend their sincere gratitude to Deputy Director, SDAPSA and Director, NRSC for their constant encouragement and support.

REFERENCES

[1]. Bryan Schauer, "Multicore Processors – A Necessity", ProQuest Discovery Guides, September 2008, http://www.csa.com/discoveryguides/multicore/review.pdf

[2]. M. Aater Suleman, Moinuddin K Qureshi , Onur Mutlu, , Yale N. Patt, "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures", January 2010

[3]. Bradford Nichols, Dick Buttlar, and Jackie Farrell, "Pthreads Programming", First Edition, O'Reilly & Associates, September 1996

[4]. Ian J. Ghent ,"Faster results by multi-threading data steps", SAS Global Forum, 2010, http://support.sas.com/resources/papers/proceedings10/109-2010.pdf

[5]. Richard T. Kouzes, Gordon A. Anderson, Stephen T. Elbert, Ian Gorton, and Deborah K. Gracio, "The Changing Paradigm of Data-Intensive Computing", IEEE Computer Society, 2009

[6]. D. Butenhof, "Programming With POSIX Threads", Addison-Wesley, 1997

[7]. Andrew Binstock, "Threading Models for High-Performance Computing: Pthreads or OpenMP?", October 2010, http://software.intel.com/en-us/articles/threading-models-for-high-performance-computing-pthreads-or-openmp

[8]. Robert G. Brown, "Maximizing Beowulf Performance", USENIX Association Proceedings of the 4th Annual Linux Showcase & Conference, October 2000, https://www.usenix.org/legacy/publications/library/proceedings/als00/2000papers/papers/full_papers/brownrobert/brownrobert.pdf