Int J Parallel Prog (2014) 42:931–947
DOI 10.1007/s10766-013-0267-4

Performance Optimization of Video Coding Process on Multi-Core Platform Using Gop Level Parallelism

S. Sankaraiah · Lam Hai Shuan · C. Eswaran · Junaidi Abdullah

Received: 30 March 2013 / Accepted: 13 August 2013 / Published online: 17 September 2013

Abstract High definition video applications often require heavy computation, high bandwidth and high memory requirements which make their real-time implementation difficult. Multi-core architecture with parallelism provides new solutions to implementing complex multimedia applications in real-time. It is well-known that the speed of the H.264 encoder can be increased on a multi-core architecture using the parallelism concept. Most of the parallelization methods proposed earlier for these purposes suffer from the drawbacks of limited scalability and data dependency. In this paper, we present a result obtained using data-level parallelism at the Group-Of-Pictures (GOP) level for the video encoder. The proposed technique involves each GOP being encoded independently and implemented on JM 18.0 using advanced data structures and OpenMP programming techniques. The performance of the parallelized video encoder is evaluated for various resolutions based on the parameters such as encoding speed, bit rate, memory requirements and PSNR. The results show that with GOP level parallelism, very high speed up values can be achieved without much degradation in the video quality.

Keywords Video encoding · H.264 · Data level parallelism · GOP · Speed up · OpenMP · Barrier · Multi-core architecture · Scalability · Load balancing

S. Sankaraiah (✉)· L. H. Shuan · C. Eswaran · J. Abdullah
Center for Visual Computing, Multimedia University, 63100 Cyberjaya, Selangor, Malaysia
e-mail: sankar2510@ieee.org; sreemula.sankaraia10@student.mmu.edu.my

L. H. Shuan
e-mail: hslam@mmu.edu.my

C. Eswaran
e-mail: eswaran@mmu.edu.my

J. Abdullah
e-mail: junaidi@mmu.edu.my

1 Introduction

H.264 video codec is an advanced video codec and popularly used for low bit-rate, multiple profiles and high video quality compression [1]. H.264 is used in many video services such as YouTube, Blu-ray discs, HDTV, live video conferencing etc. The H.264 design includes a network abstraction layer (NAL) and flexible macro-block ordering (FMO). NAL provides a network-friendly bit stream of video by placing the video data in a rational data packet called NAL unit (NALU) [2]. FMO provides the capability to divide the picture into areas called slice clusters, which significantly reduce the data corruption due to data losses. Additional functionality offered in H.264 codec includes scalable video coding and multi-view coding (MVC). Scalable video coding (SVC) provides the ability to create sub-bit streams which however, leads to lower spatial resolution or lower temporal resolution [3]. MVC provides multiple views of the same scene as a digital 3D image. It is difficult to implement H.264 in practice with all its features due to its complexity [4]. H.264 encoder consumes a lot of resources during the encoding process. It is often highly desirable to reduce the encoding time especially in applications where timing is an important factor such as real-time encoding for live streaming.

In general, a single-core processor will perform the reading and execution of instructions in a serial manner and hence its performance cannot be improved without increasing the clock speed. However, there is a limit on the achievable clock speed due to power and heat dissipation problems. Hence, an alternative solution to achieve higher speed would be to use multi-core processors [5]. If the H.264 reference software with JM18.0 [1] is executed on a multi-core processor, the CPU capacity of the processor will not be fully utilized because the encoder is not parallelized. To exploit the capability of a multi-core processor, it is necessary to introduce multi-threading and parallel programming methods [6]. Multi-threaded programming splits the serial program into multiple threads so that the threads can be processed by different cores simultaneously. Ideally, this will speed up a program by a factor, which will be equivalent to the number of cores.

In this paper, a parallel encoder is suggested and the results obtained using GOP parallelism for H.264 encoder using JM18.0 with high profile is presented. In implementing the parallelism concept, the software JM 18.0 is optimized, restructured and modified using OpenMP programming. The performance of the proposed technique is evaluated by conducting experiments with different video sequences of resolutions Quarter Common Intermediate Format (QCIF), Common Intermediate Format (CIF), Standard Definition (SD) and High Definition (HD). The experimental results show that with the proposed GOP parallelism, higher speedup values which are closer to the theoretical values can be achieved.

The rest of the paper is organized as follows: Earlier works on parallelization of H.264 are explained in Sect. 2. The methodology and experimental setup used in this paper for simulation are described in Sect. 3. This section also gives a description on the dynamic parallelism with the GOP access pattern. Section 4 presents a detailed explanation of the H.264 parallelism implementation with the GOP access pattern on a multicore architecture. The simulation results are presented in Sect. 5. Conclusions and future work are described in Sect. 6.

2 Literature Review on H.264 Parallelism

The scalability and the load balancing are the two major concerns when parallelizing a program. Scalability implies the maximum number of threads that can be created in a parallel program. The implementation of load balancing is important to ensure that all processing cores are sharing an equal amount of work. Ideally, all cores must have equal loads implying that all of them have equal processing times. In a paper reported earlier [7], it was shown that high speed- up values can be achieved using GOP level parallelism for QCIF and CIF resolutions. In this paper we extend our work for higher resolution video sequences such as SD and HD. Further we also make use of some advanced data structures such as non-temporal and *MALLOC* functions for minimizing the latency and memory stalls. Parallel execution of encoding and decoding has been reported in several papers [7–11]. These approaches have limitations with regard to scalability, latency and load balancing for large number of cores.

A parallel implementation of H.264 encoder based on the hybrid version of GOP and frame-level parallelism on clustered workstations using MPI has been reported by Rodriguez et al. [8]. In [8], the authors have used MPI, which has large communication overhead whereas in this research work OpenMP is used which has low latency, high bandwidth and allows run-time scheduling with dynamic load balancing. The advantage of GOP-level parallelism is that it does not have data dependencies since each GOP starts with a new video sequence. Thus, there is no communication overhead and no synchronization is needed between processors or cores. Besides, GOP-level parallelism has a high scalability due to the large number of GOPs present in the video. Frame and slice-level parallelisms suffer from the problem of scalability due to the fact that only a limited number of B-frames will normally be present in a video sequence and similarly only a limited number of slices will be present in a frame.

However, GOP-level parallelism requires a large amount of memory for storing all decoded frames especially with higher resolutions [16]. Though real -time operation is possible with this setup, it has the problem of higher latency. Antonio et al. [9], have implemented macro-block level and slice level parallelisms and achieved good speedup values. The results of the implementation methods reported in [10,11] leads to many memory stalls. This problem is solved in the proposed approach by dynamically detecting the dependencies and exploiting the parallelism automatically. We show that by using the dynamic scheduling technique and some advanced data structures, we can minimize the cache trashing thereby reducing the memory stalls. VanderTol et al. [12], proposed MB level and static frame level parallelism and explained the utilization of spatial MB-level parallelism. In this approach, MB-level parallelism is merged with frame-level parallelism. In [13], the frame-level parallelism is implemented in a static manner using motion vectors. Gurhanl et al. [14,15], have implemented H.264 decoder with GOP parallelism. The technique used by these authors in implementing GOP parallelism for decoder has problems such as latency, load balancing and memory limitation for higher resolution video sequences. Alvvarez et al., introduced dynamic frame level parallelism associated with MB level parallelism. Techniques for analyzing the scalability in terms of parallelism are suggested by Cor Meenderinck et al. [16, 17].

Parallelism of the H.264 encoder can basically be divided into two categories, namely, task domain decomposition and data domain decomposition [18,19]. The main idea in task domain decomposition is to split the encoder's tasks among the threads. Decomposition of task is identified by the functions in the program. The data domain decomposition method, also known as the loop level parallelism, explores the idea of distributing the data into multiple threads for processing [20,21].

3 Methodology and Simulation Environment

The main motivation for implementing on GOP level parallelism for video encoding is that GOP's are independent of each other and a single master thread controller can be used to distribute GOP's to available threads. However, GOP level parallelism requires more memory [8]. In the proposed method, the problem of high memory requirement is solved using a dynamic scheduling approach in which information pertaining to only three frames is stored in the main memory. The remaining frames in each GOP are stored in a temporary buffer. It is assumed that only two frames in each GOP are used for reference. Furthermore, some changes are also implemented in the data structures for optimizing the memory required.

The reference software JM 18.0 of H.264/AVC codec comprises encoder (lencod) and decoder (ldecod). JM 18.0 is based on serial programming, which encodes and decodes the set of frames serially making the encoding and decoding times longer. In this paper, the emphasis is on the speedup of H.264 encoder. The video sequences are divided into a set of frames which are called the GOP's and the GOP's are then encoded in parallel on multi-core platform. Before applying the parallelization concept, the redundancy or unused codes are removed and modified in order to efficiently utilize the memory cache access. With this modification, the data access parameters can be dynamically stored and removed leading to significant reduction of memory usage.

In this section, GOP-level parallelism method with dynamic scheduling is described. The proposed technique checks the thread status in a dynamic manner, so that the latency problem is significantly reduced and it also improves the scalability. The computations of processing cores are modeled based on the number of cycles. In this implementation, an Intel i7 platform is used for simulating a 4 physical cores system and as an 8 logical cores system using hyper-threading technology. It is assumed that, each core has an independent data-cache (L1) and data can be copied from additional caches (L2 and L3) through four channels. In this approach, one thread controller and 2–8 slave threads are created depending on the number of threads required. To record the encoder's elapse time, all existing native services and processes in the cores should be closely monitored and controlled. It is also important to ensure that the computer is not running any additional background tasks during encoding as it will incur additional overhead to the processor. The experimental results are obtained based on H.264 high profile using I, P and B frames. The experiments are conducted using JM 18.0 reference software and compiled with Microsoft Visual Studio 2010 using an Intel i7 platform as described below:

Intel Core$^{\text{TM}}$ i7 CPU 930, running at 2.8 GHz with four 32 KB D-Cache (L1), four 32 KB I-Cache (L1), four 256 KB cache (L2) with 8-way set associative and 8 MB

L3 cache with 16-way set associative and 8 GB RAM. The operating system used is Windows 7 64 bits Professional version.

The following are some of the additional settings that are used to create the test environment:

- All external devices are disconnected from the computer excluding the keyboard and mouse
- All drivers for network adapters are disabled
- Windows Aero, Gadget, Firewall is disabled
- Power setting is changed to “Always on” for all devices
- All extra windows features are removed with the following exceptions:
  - Microsoft .net framework

All simulations are performed under this controlled environment and the encoder’s elapsed time is recorded using Intel Parallel Studio’s 2011 Vtune Amplifier and AMD code analyst. The memory leaks are analyzed using Intel Parallel Inspector 2011. The parallel programming is implemented using OpenMP technique. The resolutions of the video sequences used in the simulation are mainly (SD) and (HD) resolutions. This paper presents an enhanced parallelization approach that provides good scalability. GOP-level parallelism has many advantages over other methods in terms of scalability, independency, load balancing and utilization of the processing cores. The scalability is tested by increasing the number of processing cores and applying homogeneous software optimization techniques to each core.

4 Implementation of Scalable Gop Parallelism on H.264 Video Encoder

4.1 Master Thread Controller

In order to attain better data-level parallelism, the processing elements should be independent with regard to the data. In the proposed GOP-level parallelism, each GOP is processed with a separate thread and closed GOPs are used. Though the threads are processed independently, dependency exists within the GOP. The dependency that exists among Intra, Prediction and Bi-prediction frames in a GOP, which may not affect the process. Since every GOP initiates with an Intra frame to encode, followed by the prediction frame and the bi-prediction frames. Figure 1 shows the GOP data access pattern with two threads on an Intel i7 platform. In the proposed approach, all the frames of a GOP are stored in a temporary buffer and sequentially transferred to the corresponding threads for processing. Using the Intel i7 4-core configuration, the structure of scalable GOP level parallelism with 4threads is shown in Fig. 2. In a similar way, 8 threads can be created for the Intel i7 platform using Hyper-threading for 8-logical core configuration. In this implementation, video sequences are divided into GOPs according to the preset number of frames. The data of each frame in each GOP are arranged in the sequence of IBBP…. BBPBB and are stored in a temporary buffer. All GOPs with threads are scheduled using the scheduler and controlled by a master thread control unit as shown in Fig. 2. In this implementation, L2 and L3 caches are used effectively. In Intel i7 platform, either 4-core or 8-core configuration, the L2 or L3 caches are connected to the main memory with a separate bus.

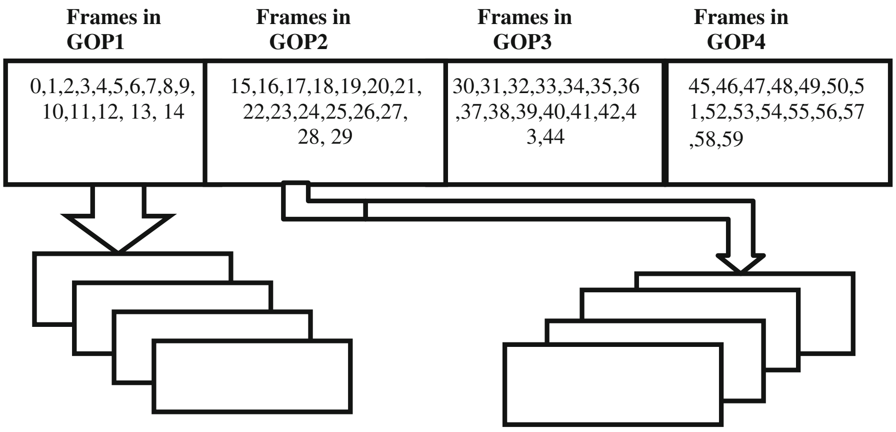


Fig. 1 First two GOPs access pattern on threads

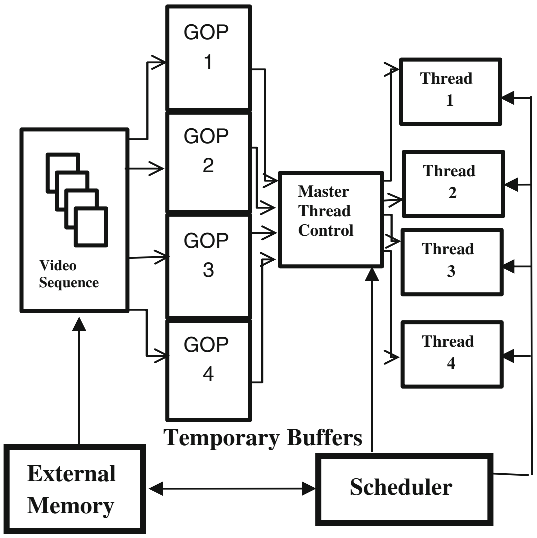


Fig. 2 Implementation of scalable parallelism at GOP level with four threads

4.2 Dynamic Scheduling Algorithm

A Dynamic Scheduling Algorithm is designed to assign raw video sequences into temporary buffer. The steps involved in this scheduling are as follows:

1. Storing of video sequences in external memory
2. Dividing the video sequences into GOPs and assigning a reference id to each GOP
3. While looping for (not the end of the sequence)
   (i). Check the Rate Distortion (RD) cost of the frames in the GOP
   (ii). Check the encoding parameters (search pattern of motion estimation)
4. Encode the GOP according to the reference id

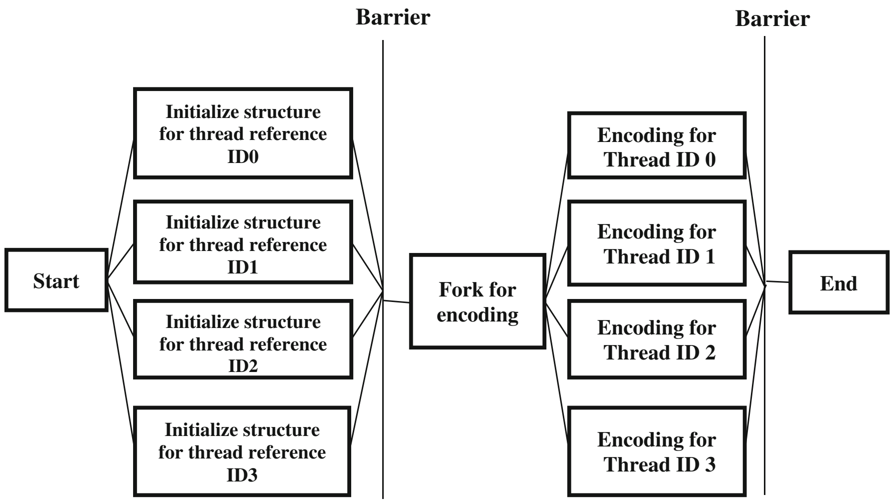

Barrier
Barrier
Start
Initialize structure for thread reference ID0
Initialize structure for thread reference ID1
Initialize structure for thread reference ID2
Initialize structure for thread reference ID3
Fork for encoding
Encoding for Thread ID 0
Encoding for Thread ID 1
Encoding for Thread ID 2
Encoding for Thread ID 3
End

Fig. 3 Structure of parallelism with barrier

5. Write the encoded frames into the external memory
6. After completely encoded a GOP, reference id will be increased
7. Exit if reference id is equal to the total number of GOPs
8. Else repeat steps 4 to 7

For the calculation of the total elapsed time, step 5 is omitted as it is a common overhead incur for both parallel and sequential encoding.

In the proposed dynamic scheduling algorithm, no dedicated thread is used for task scheduling. Instead, any available thread can be used for performing. The scheduling algorithm attempts to minimize the barrier wait [22,23]. It is possible to enforce a flush system for all the parallel threads, which updates the encoded GOP information. Figure 3 shows the structure of parallelism with barrier. The controller continuously keeps track of the threads and sends frames to the idle threads. One thread is assigned for each GOP for execution. Each thread executes the same operation on different frames, thus ensuring that all threads take the same processing time. The balancing of tasks in the GOPs is performed by determining the execution time of each function dynamically thus solving the load balance problem. Using the RD cost, the type of frame (I, B, or P) is determined and the referencing is performed for encoding accordingly [2].

4.3 Data Structures for Memory Optimization

The source code of JM18.0 is dispersed with pointers all over the location. Each structure has a pointer that will point to all the structures. The entire structure is tangled together by these pointers. These pointers work fine serially. However, it causes pointer and memory access issues when parallelized. This is due to the fact that when making a structure private to a processing core, the pointer's value is made private but not the pointed memory location by the pointer. There are actually 2 parts to take note when resolving this issue. The first part is the fact that the structures in the lower hierarchy having pointer pointing to any structure that is above it in the hierarchy. The second part is the fact that some structures are using pointer to another structure in lower hierarchy. In the original source code of JM18.0, the pointers are replaced with suitable data structures. To resolve the pointer to the structure, the encoding parameter structure p_Enc (JM 18.0 encoder code) is made into a global variable and all structures are directly or indirectly pointed to this structure [1]. This provides flexibility in pointer structures. For example, whenever there is a need to make a pointer private, the value of the pointer is copied to another memory location and then de-referenced from its variable. Finally, within the parallelized region, the private variable will be updated directly into the structure p_Enc. In order to store the GOP's references, the proposed encoder is implemented with advanced data structures, which eliminates the need for the extra picture buffer. Using Intel Parallel Studio's 2011 Vtune Amplifier, it is observed that *the memset function* is taking most of the time to set memory locations for specific values. In JM 18.0 encoder, most of the calls to *memset* is done by *calloc* function, for allocation of memory locations. All *calloc* functions are replaced by *malloc*, which allocates memory and also reduces the memory stalls [22,23]. The *malloc* function reduces the runtime and improves speedup value in a significant manner. In JM reference software, for transferability or compactness purposes, all the memory allocations are performed with an initialization. To solve the scalability and latency issues that occur for large volumes of data, the 'non-temporal stores' function [22] is used which stores data straight to the main memory without going through cache allocation which makes the memory initialization faster. The use of 'non-temporal stores' function provides better scalability, since all the cores will work independently. Figure 4 shows the flow chart illustrating the steps involved in the proposed GOP based encoding process.

5 Experimental Results and Discussions

5.1 Load Balancing

In order to improve the parallel performance, the proposed algorithm starts the GOP encoding and reading processes at the earliest possible instants and continues without any overlap between the cores. This process minimizes the communication delays between the processors without affecting the parallel execution time. In the proposed implementation, the processors handle the GOPs dynamically and this improves the load balance. The CPU usage graphs before and after parallelization are shown in Figs. 5a, b, respectively. From these graphs, it can be observed that all the four cores are equally balanced after the implementation of GOP parallelism.

5.2 Memory Utilization

The main bottleneck for GOP level parallelism implementation is the memory issue since each GOP processing deals with a large amount of data resulting in many memory

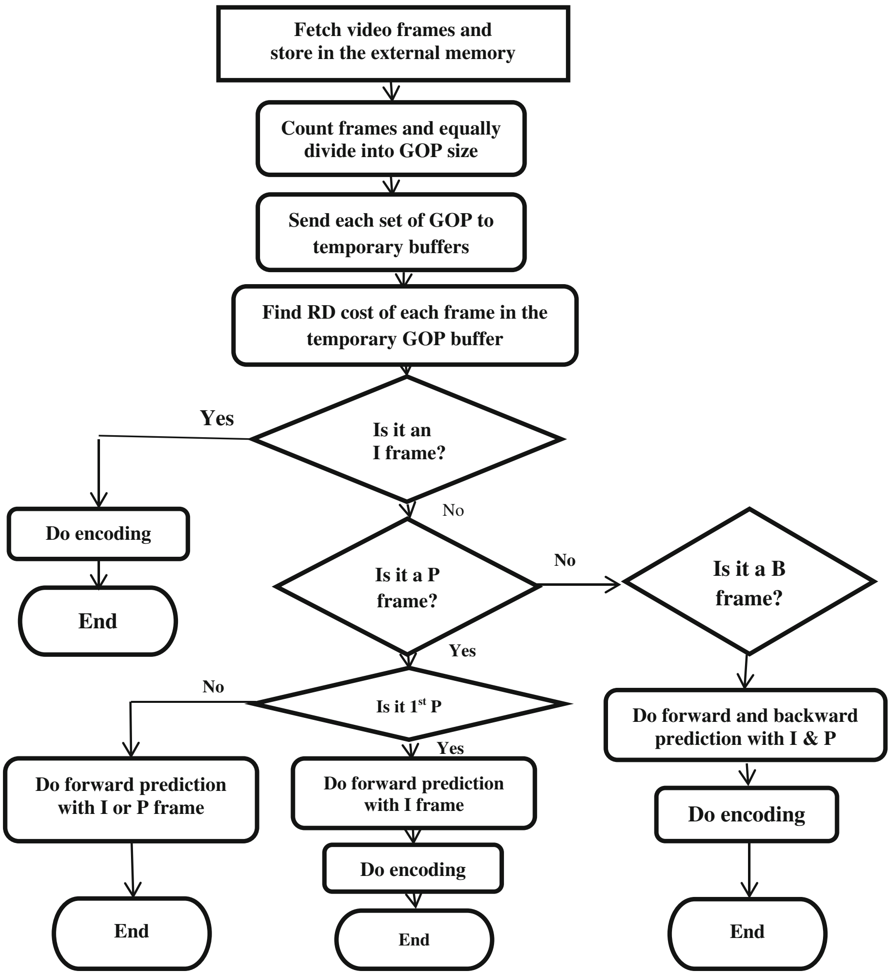
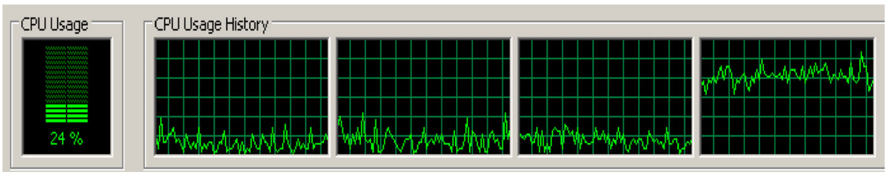
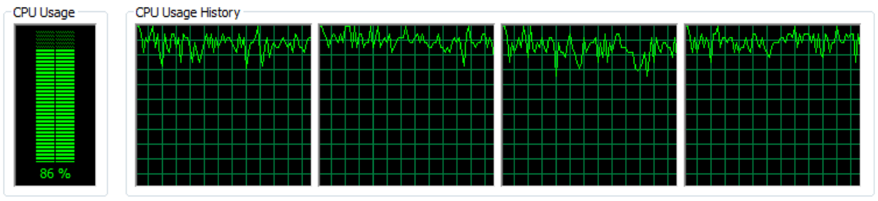

Fetch video frames and store in the external memory
Count frames and equally divide into GOP size
Send each set of GOP to temporary buffers
Find RD cost of each frame in the temporary GOP buffer
Is it an I frame?
Yes
Do encoding
End
No
Is it a P frame?
No
Is it a B frame?
Yes
Is it 1st P
No
Yes
Do forward prediction with I or P frame
End
Do forward prediction with I frame
Do encoding
End
Do forward and backward prediction with I & P
Do encoding
End

Fig. 4 Flow chart of GOP parallelism encoding process

load and store operations. Especially, when each core doesn't have a direct path to the memory for read and write operations, the bandwidth becomes insufficient for parallel working cores. In order to resolve these issues, the proposed method utilizes the memory effectively by using advanced data structures such as *malloc* and non-temporal functions. Table 1 shows the number of cache access and cache misses per frame with respect to the data cache (L1) during the encoding process. The results shown in Table 1 indicate the miss rates are much lower compared to those obtained by other researchers' results [18–21]. This is due to the fact that GOP parallelism performs more processes per frame with the better data neighborhood in the data cache. From the analysis of different frames with the cache access and misses, it is observed that P and B frames have more data misses than I frame. This is due to the fact that B frames refer to multiple frames. This problem occurs only in higher resolutions (HD and FHD). To



(a) CPU usage before parallelization with original JM18.0



(b) CPU usage after parallelization of JM 18.0

Fig. 5 Performance of CPU load balance

Table 1 The results of parallel encoding of high motion video sequence (rush_hour_HD)

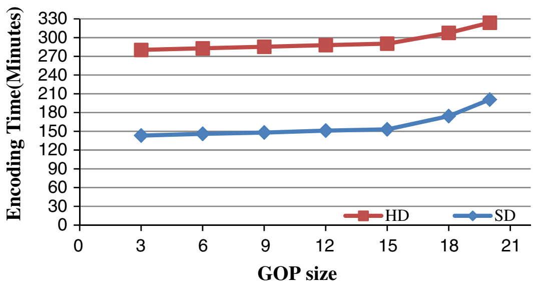
Sequences	Cache access	Cache misses
Sunflower_SD	358×10^6	3.8×10^6
Sunflower_HD	562×10^6	7×10^6
Rush_hour_SD	363×10^6	4×10^6
Rush_hour_HD	575×10^6	7.4×10^6

minimize this effect in the proposed method, only two B frames are used in between I or P to minimize referencing. In this implementation referencing is done dynamically. It is also observed that the bus activities do not increase significantly when the number of threads is increased. The elapse time is reduced by better utilization of the system by exploiting the optimum thread-level parallelism.

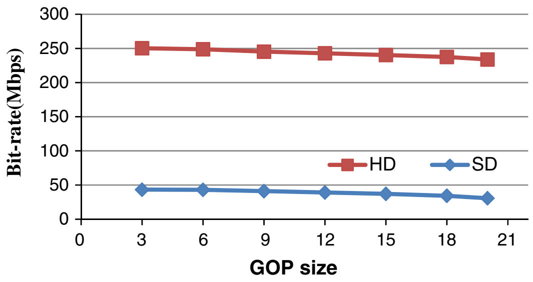
To find the optimum number of frames in a GOP, the GOP size varies from 3 to 20 and the corresponding encoding time, bit rate and PSNR values are recorded. The results obtained for Intel i7 platform with 4 threads configuration is shown in Fig. 6. The results for the case of 2 and 8 threads (Hyper-threading) are almost similar to those shown in Fig. 6. As seen from Fig. 6a, b optimum encoding time and bit-rate is achieved for GOP size 15 and hence further experiments are performed using this GOP size.

5.3 PSNR, Encoding Time and Bit-Rate Values

The results obtained for PSNR, total encoding time and bit-rate using GOP size of 15 are presented in Tables 2, 3, 4, 5 for two types of video sequences (with slow and high motions) for SD (720 × 576) and HD (1280 × 720) resolutions. The results have



(a) Encoding time versus GOP size



(b) Bit-rate versus GOP size

Fig. 6 GOP size with different parameters

Table 2 The results of parallel encoding of low motion video sequence (sunflower_SD)

Parameters	Configuration			
	Original JM	2 threads	4 threads	8 threads
Average PSNR (dB)	43.24	43.24	43.24	43.24
Total encoding time (min)	562.18	286.24	146.4	73.77
Bit-rate (Mbit/s)	42.35	42.35	42.35	42.35

been obtained by performing tests with 300 frames for a sequence using 2, 4 and 8 threads and compared to the original JM results on the same platform. Table 2 shows the results of slow motion (sunflower_SD) video sequence. Table 3 shows the results of high motion (rush_hour_SD) video sequence. Table 4 shows the results of slow motion (sunflower_SD) video sequence. Table 5 shows the results of high motion (rush_hour_HD) video sequence.

It is seen from the results shown in Tables 2, 3, 4, 5 that a significant reduction in the encoding time has been achieved using GOP parallelism. Also we note that the implementation of this method has not resulted in any change in the bit rate of the output file.

The encoding results (PSNR and bit rate) presented in Tables 2, 3, 4, 5 show clearly that the GOP parallel implementation scheme does not affect the PSNR and bit-rate values. These results have been achieved without any extra computational overhead. For the sake of subjective comparison, samples of video frames obtained using parallel

Table 3 The results of parallel encoding of high motion video sequence (rush_hour _SD)

Parameters	Configuration			
	Original JM	2 threads	4 threads	8 threads
Average PSNR (dB)	43.06	43.06	43.06	43.06
Total encoding time (min)	586.59	298.67	152.65	76.98
Bit-rate (Mbit/s)	45.19	45.19	45.19	45.19

Table 4 The results of parallel encoding of low motion video sequence (sunflower_HD)

Parameters	Configuration			
	Original JM	2 threads	4 threads	8 threads
Average PSNR (dB)	44.19	44.19	44.19	44.19
Total encoding time (min)	1, 062.18	540.82	276.6	139.39
Bit-rate (Mbit/s)	259.19	259.19	259.19	259.19

Table 5 The results of parallel encoding of high motion video sequence (rush_hour _HD)

Parameters	Configuration			
	Original JM	2 threads	4 threads	8 threads
Average PSNR (dB)	44.25	44.25	44.256	44.25
Total encoding time (min)	1, 092.24	556.13	284.43	143.33
Bit-rate (Mbit/s)	273.56	273.56	273.56	273.56

and sequential encoding schemes are shown in Fig. 7. Figure 7a, c represent the video samples obtained using sequential encoding scheme and Fig. 7b, d represent the video samples obtained using parallel encoding scheme. It is clear from Fig. 7 that the quality of the parallel encoded video frame is identical to that of the sequentially encoded video frame.

Figure 8 shows the PSNR results obtained for different number of threads using i7 platform for SD and HD resolutions.

5.4 Speedup Performance

According to Amdahl's law, the speedup that can be obtained with no threads is given by [21]:

$$speedup = \frac{1}{r_s + \frac{r_p}{n}}$$

where $\mathbf{r_p}$ is parallel ratio, $\mathbf{r_s}$ is serial ratio $(\mathbf{1} - \mathbf{r_p})$ and $\mathbf{n}$ is the number of threads. Using Intel Parallel Studio Vtune Amplifier the following results are obtained with different threads.

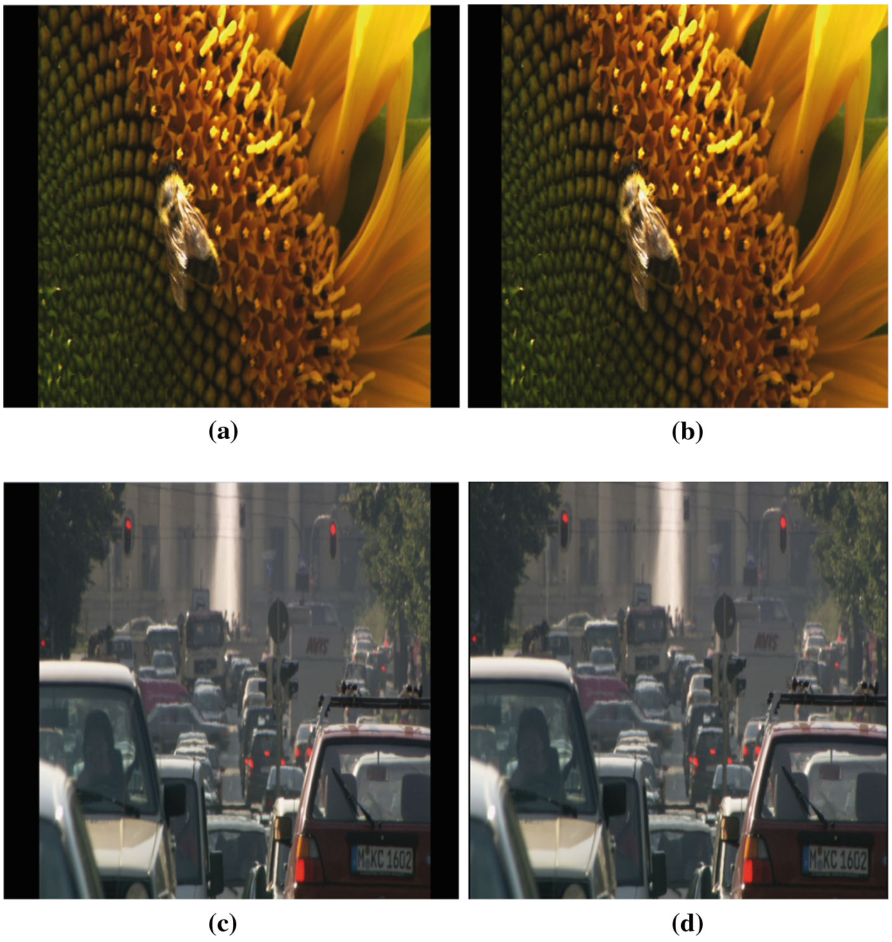


Fig. 7 The quality of parallel GOP encoded video versus non parallel encoded video

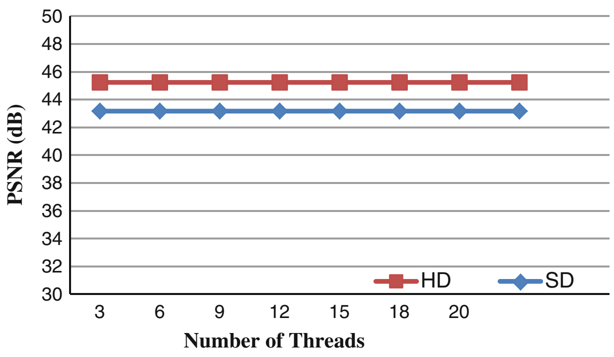


Fig. 8 PSNR versus the number of threads

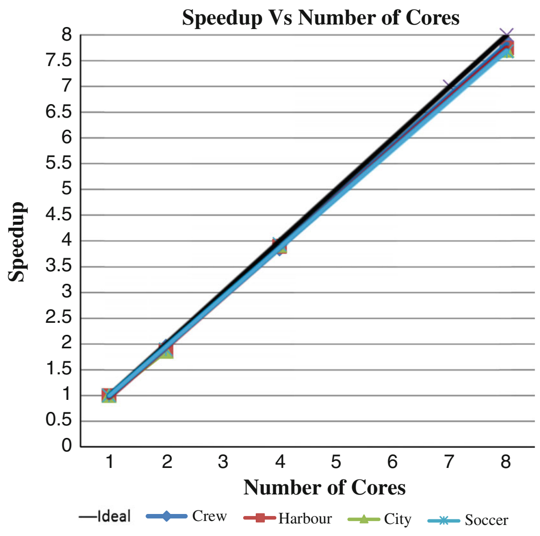


Fig. 9 Speedup using different video sequences with HD resolution

1. With 2 threads configurations, we get the parallel ratio of $\mathbf{r_p} = 0.98$ and $\mathbf{r_s} = 0.02$, respectively. With these values, the speedup that is obtained according to Amdahl's law is 1.96, which is very close to the maximum speed up to 2 for a 2 core system. This shows that the 2 processing cores are utilized effectively when the other 2 cores are kept idle.
2. With 4 threads configurations, the parallel ratios are $\mathbf{r_p} = 0.994$ and $\mathbf{r_s} = 0.006$. With these values, the speedup achieved is 3.93, which is very close to the maximum speed up of 4.
3. With 8 threads configurations, the parallel ratio of $\mathbf{r_p} = 0.991$ and $\mathbf{r_s} = 0.009$ are obtained. In this case, the value of speed up obtained is 7.62. This value is not very close to the maximum speed up of 8 even though dynamic scheduling is used to reduce the barrier. This is due to the fact that the system is only having 8 logical cores and only 4 physical cores.

Figure 9 presents the speedup values obtained according to Amdahl's law using GOP level parallelism for four different video sequences (Crew, Harbour, City and Soccer) with HD resolution. These values are higher compared to those obtained by using other types of parallelism [6–8, 17–21]. Figure 10 shows the speedup versus the number of cores for the soccer video sequence with QCIF, CIF, SD and HD resolutions. In this case also, it can be observed that the resulting speed up values is very close to the ideal or theoretical values. In Figs. 9 and 10, the same speedup value is obtained for a particular video sequence irrespective of its resolution (QCIF, CIF, SD and HD).

Figure 11 shows the results obtained using different number of thread configurations for HD videos. It is observed that the speedup is almost constant (or slightly lower) when the number of threads exceeds the number of cores. This is due to the fact that additional overheads are required to preempt, schedule and perform context switching.

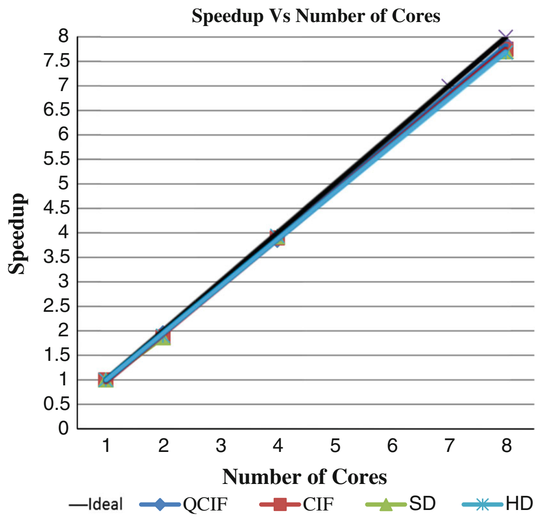


Fig. 10 Speedup for different number of cores with QCIF, CIF, SD and HD resolutions

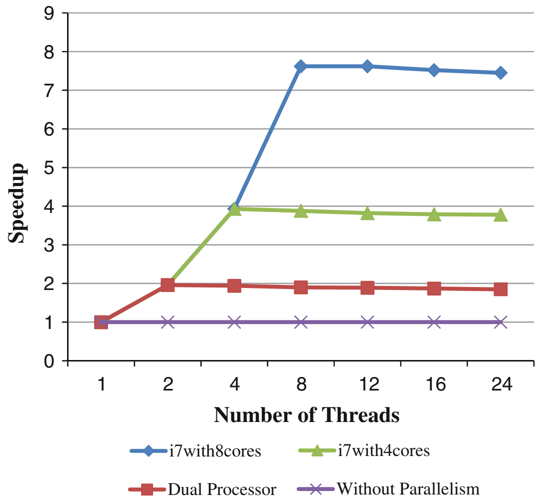


Fig. 11 Speedup attained for GOP level parallelism with HD resolution (rush_hour video)

Additional buffers are also needed to hold the information or to process the extra threads.

In Fig. 11, it is shown that without parallelism, identical speedup values are achieved for multi-core processors, since all are using single thread configuration only. The results obtained demonstrate significant speedup improvement on the Intel i7 platform with 2 threads, 4 threads and 8 threads configurations. These results are better compared to those obtained using parallelization at macro-block level, frame level and slice-level parallelism [6,8,17,18]. Furthermore, the experimental results show clearly that the proposed technique does not degrade the quality of the video which may not be the case when other parallelization techniques are adopted. Though the proposed approach leads to less latency values compared to other known techniques, for applications involving Full HD video real-time implementation is still a problem.

6 Conclusion

In this paper, we have presented a method based on GOP level parallelism for H.264 video encoder and tested its performance using Intel i7 platform with 2 threads, 4 threads and 8 threads configurations. The proposed parallelization strategy can overcome many shortfalls of the other known methods such as scalability issues and data dependency constraints. The experimental results shown that the GOP-level parallelism strategy efficiently exploits the capabilities of the multicore processors. The speedup values obtained using 2 threads, 4 threads and 8 threads configurations are 1.96, 3.93 and 7.62, respectively. These values are higher compared to those achieved using other levels of parallelism. Although, the focus of this paper is on the use of H.264 encoder for video processing, the proposed technique can be applied to other video codecs as well as for computationally intensive multimedia applications.

References

1. International Standard of Joint Video specification (ITU-T Rec. H. 264 ISO/IEC) (2009)
2. Ostermann, J., Bormans, J., List, P., Marpe, D., Narroschke, M., Pereira, F., Stockhammer, T., Wedi, T.: Video coding with H.264/AVC: tools, performance, and complexity. IEEE Circuits Syst. Mag. **4**(1), 7–28 (2004)
3. Kwon, S-k, Tamhankar, A., Rao, K.R.: Overview of H.264/MPEG-4 part 10. J. Vis. Commun. Image Represent. **17**(2), 186–216 (2006)
4. Schwarz, H., Marpe, D., Wiegand, T.: Overview of the scalable video coding extension of the H.264/AVC standard. IEEE Trans. Circuits Syst. Video Technol. **17**(9), 1103–1120 (2007)
5. Hoogerbrugge, J., Terechko, A.: A multithreaded multicore system for embedded media processing. Trans. High Perform. Embed. Archit. Compil. III, LNCS **6590**, 154–173 (2011)
6. Gschwind, M.: The cell broadband engine: exploiting multiple levels of parallelism in a chip multiprocessor. Int. J. Parallel Program., Kluwer, Norwell, MA, USA **35**, 233–262 (2007)
7. Sankaraiah, S., Shuan, L.H., Eswaran, C.: GOP level parallelism on H.264 video encoder for multicore architecture. IPCSIT, IACSIT Press **7**, 127–132 (2011)
8. Rodriguez, A., Gonzalez, A., Malumbres, M.P.: Hierarchical parallelization of an h.264/avc video encoder. In: Proceedings of the International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06), Bialystok, pp. 363–368 (2006)
9. Rodrigues, A., Roma, N., Sousa, L.: p264: open platform for designing parallel H.264/AVC video encoders on multi-core systems. In: 20th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2010), Proceedings. ACM, Amsterdam, The Netherlands, June, pp. 81–86 (2010)
10. Chen, Y.-K., Li, E.Q., Zhou, X., Ge, S.: Implementation of H.264 encoder and decoder on personal computers. J. Vis. Commun. Image Represent. **17**(2), 509–532 (2006)
11. Azevedo, A., Juurlink, B.H.H., Meenderinck, C.H., Terechko, A., Hoogerbrugge, J., Alvarez, M., Ramirez, A., Valero, M.: A highly scalable parallel implementation of H.264. Trans. High Perform. Embed. Archit. Compil (HiPEAC) **4**(2), 1–25 (2009)
12. Van Der Tol, E.B., Jaspers, E.G.T., Gelderblom, R.H.: Mapping of H.264 decoding on a multiprocessor architecture. In: Image and Video Communications and Processing, Proceedings of SPIE 5022 Santa Clara, CA, pp. 707–718 (2003)
13. Kim, M., Song, J., Kim, D.H., Lee, S.: H.264 decoder on embedded dual core with dynamically load-balanced functional paritioning. In: 17th IEEE International Conference on Image Processing (ICIP), Hongkong, pp. 3749–3752 (2010)

14. Gürhanli, A., Chen, C.C.-P., Hung, S.-H.: GOP-level parallelization of the H.264 decoder without a start-code scanner. In: Proceedings of 2nd International Conference on Signal Processing Systems (ICSPS), Dalian, China,3, pp. 627–630 (2010)
15. Gurhanli, A., Chen, C.C.-P., Hung, S.-H.: Coarse grain parallelization of H.264 video decoder and memory bottleneck in multi-core architectures. Int. J. Comput. Theory Eng. **3**(3), 375–381 (2011)
16. Meenderinck, C., Azevedo, A., Alvarez, M., Juurlink, B., Ramirez, A.: Parallel scalability of H.264. In: Proceedings of the first workshop on programmability issues for multi-core computers (MULTPROG-1), Goteborg, Sweden, pp. 1–12 (2008)
17. Meenderinck, C., Azevedo, A., Juurlink, B., Mesa, M.A., Ramirez, A.: Parallel scalability of video decoders. J. Signal Process. Syst. **57**(2), 173–194 (2009)
18. Alvarez Mesa, M., Ramirez, A., Azevedo, A., Meenderinck, C., Juurlink, B., Valero, M.: Scalability of macroblock-level parallelism for H.264 decoding. In: Proceedings 15th International Conference on Parallel and Distributed Systems (ICPADS'09), Shenzhen, China, pp. 236–243 (2009)
19. Chen, Y.-K., Tian, X., Ge, S., Girkar, M.: Towards efficient multilevel threading of H.264 encoder on intel hyper-threading architectures. In: Proceedings of the 18th International Parallel and Distributed Processing Symposium, Santa Fe, New Mexico, pp. 63–67 (2004)
20. Ge, S., Tian, X., Chen, Y.-K.: Efficient multithreading implementation of H.264 encoder on intel hyper-threading architectures. In: Proceedings of the Joint Conference of the Fourth International Conference on Information, Communications and Signal Processing and Fourth Pacific Rim Conference on Multimedia, Singapore, pp. 469–473 (2003)
21. Kim, Y.-I, Kim, J.-T., Bae, S., Baik, H., Song, H.J.: H.264/AVC decoder parallelization and optimization on asymmetric multicore platform using dynamic load balancing. In: IEEE International Conference on Multimedia and Expo, Hannover, pp. 1001–1004 (2008)
22. Taylor, S.: Optimizing Applications for Multi-Core Processors: Using the Intel Integrated Performance Primitives, 2nd edn. Intel Press, USA (2007)
23. Gerber, R., Bik, A.J.C., Smith, K.B., Tian, X.: The Software Optimization Cookbook: High-Performance Recipes for IA-32 Platforms, 2nd edn. Intel Press, USA (2005)