EURASIP Journal on Advances in Signal Processing
a SpringerOpen Journal

RESEARCH **Open Access**

Scalable video encoding with macroblock-level parallelism

Sreeramula Sankaraiah*, Lam Hai Shuan, Chikkannan Eswaran and Junaidi Abdullah

Abstract

H.264 video codec provides a wide range of compression options and is popularly implemented over various video recording standards. The compression complexity increases when low-bit-rate video is required. Hence, the encoding time is often a major issue when processing a large number of video files. One of the methods to decrease the encoding time is to employ a parallel algorithm on a multicore system. In order to exploit the capability of a multicore processor, a scalable algorithm is proposed in this paper. Most of the parallelization methods proposed earlier suffer from the drawbacks of limited scalability, memory, and data dependency issues. In this paper, we present the results obtained using data-level parallelism at the macroblock (MB) level for encoder. The key idea of using MB-level parallelism is due to its less memory requirement. This design allows the encoder to schedule the sequences into the available logical cores for parallel processing. A load balancing mechanism is added to allow the encoding with respect to macroblock index and, hence, eliminating the need of a coordinator thread. In our implementation, a dynamic macroblock scheduling technique is used to improve the speedup. Also, we modify some of the pointers with advanced data structures to optimize the memory. The results show that with the proposed MB-level parallelism, higher speedup values can be achieved.

Keywords: Video encoding; H.264; MB-level parallelism; Elapse time; Speedup; OpenMP; Barrier; Multicore architecture; Scalability; Load balancing

1 Introduction

H.264 is an emerging video coding standard developed by the ITU-T Video Coding Experts Group (VCEG) and ISO/IEC Moving Picture Experts Group (MPEG) together with the partnership of Joint Video Team (JVT). H.264 has been developed with the aim of providing good-quality video at lower bit rates compared to previous video compression standards. H.264 also provides flexibility in serving broad range of video applications by supporting various bit rates and resolutions [1]. The improvement on bit rate efficiency of H.264 is at the cost of increased complexity compared to existing standards. The higher complexity of H.264 encoder results in longer encoding time [2]. This creates a need for improving the encoding time of the video for batch processing or real-time applications. Hardware acceleration or parallel algorithm for multicore processor is often needed to increase the processing speed of the encoder. Parallel algorithm is becoming popular with the usage of multicore processors over the years even for mobile devices. Parallel algorithms for H.264 encoder design have been discussed in several papers [2-9]. These papers describe different levels of parallelism that can be applied on H.264 encoder such as GOP level, frame level, slice level, and macroblock level. Out of these, macroblock-level parallelism is often favored for its fine granularity and its ability to prevent any video quality losses from its serial algorithm counterpart [4]. Macroblock-level parallelism provides good scalability and load balancing. Some of the main concerns in designing macroblock-level algorithm are the accessing pattern, the data partitioning, and the load balancing. Macroblock access pattern defines the way in which the data is to be processed in order to reduce the data dependencies. The data partitioning process defines how each macroblock can effectively be assigned to separate processor core. The load balancing mechanism ensures that each processor is loaded with similar amount of workload to prevent the processing core from staying idle or starvation.

*Correspondence: sankar2510@ieee.org
Center for Visual Computing, Multimedia University, Cyberjaya, Selangor 63100, Malaysia

In general, all the threads may not run in parallel and there will be a time difference of a few milliseconds to microseconds among the threads. This problem leads to load imbalancing. In this paper, a dynamic thread scheduling strategy is proposed to solve the load imbalancing problem. Furthermore, a new technique to access data patterns is also proposed to improve the encoding time. Another contribution of this paper is on memory optimization using advanced data structures. This paper proposes a scalable algorithm based on the above strategies that exploits the capability of a multicore processor using macroblock-level parallelism for video encoder. The remainder of this paper is organized as follows: Section 2 gives the description of previous work related to macroblock parallelism. Section 3 gives an overview on the design consideration of the parallel algorithm. In Section 4, the design and implementation of macroblock parallelism of H.264 encoder parallelism are discussed in detail. In Section 5, the experimental results of the design are presented and analyzed. Section 6 consists of the conclusion and the possible future work.

2 Literature review

Many researchers have been working on parallel algorithms. The popular parallel algorithms that are proposed are at the GOP, frame, slice, and macroblock levels. Many researchers have implemented macroblock-level parallelism [3-10], but all the proposed methods so far have scalability issues. In [4], a method using SIMD instructions has been proposed to improve the encoding time of H.264. However, this approach is too complex to implement on personal computers. The parallel algorithm using wave-front technique reported in [5] splits a frame into macroblocks and maps these blocks to different processors along the horizontal axis. This technique requires data communication among the parallel processing blocks (except for the outer blocks of a frame), slowing down the encoding process. The speedup values achieved with this implementation are 3.17 and 3.08 for quarter common intermediate format (QCIF) and common intermediate format (CIF) video formats, respectively [5]. The macroblock region partition (MBRP) algorithm proposed in [6] adopts wave-front technique and focuses on reducing the data communication between processors using a new data partitioning method. This data partitioning method assigns a specific macroblock region for each processor, so that neighboring macroblocks are mostly handled by the same processor. However, in this implementation, the waiting time of the processors before starting to encode a new macroblock is high [6]. The speedup values achieved with this method are 3.32 and 3.33, respectively, for CIF and standard definition (SD) video formats. The MBRP algorithm has not been applied to higher resolutions such as high-definition (HD) and full high-definition (FHD). A new macroblock-level parallelism method has been reported in [7]. In this method, the data partitioning on the macroblocks eliminates the dependency among the macroblocks at the beginning of the encoding process. Encoding the subsequent frames is initiated only when the reconstructed macroblocks constitute more than half of a frame. Thus, this method increases the concurrency of the thread-level parallelism to process multiple frames. The speedup values achieved with this method for CIF, SD, and HD video resolutions are around 3.8×. However, in this implementation, the authors have used only I and P frames and they have not included B frames. The dynamic data partition algorithm proposed in [8] for macroblock-level parallelism reduces data communication overhead and improves concurrency. The dynamic data partition algorithm achieves speedup values of 3.59 for CIF, 3.88 for 4CIF, and 3.89 for HD resolution video formats. Even though good speedup values are obtained, these values are not consistent with different video formats. Various thread-level techniques have been proposed in [9] to effectively utilize a multicore processor. We have adopted some of these techniques in the proposed algorithm to improve the encoding time.

3 Design considerations

3.1 Data dependencies

In general, data dependency remains as one of the major problems in parallel design. Macroblock-level parallelism is also suffering from data dependency problem. There are three major types of data dependencies for the macroblock-level parallelism which are the dependencies introduced by intra-prediction (Intra Pred.), inter-prediction (MV Pred.), and deblocking filter (De-blocking). Some of the neighboring macroblocks need to be encoded before the current macroblock can be encoded. Figure 1 shows the current macroblock and the related neighboring macroblocks used in a wavefront model. In this case, four neighboring macroblocks need to be encoded before encoding the current macroblock.

In the proposed implementation, only three neighboring macroblocks are required to encode the current macroblock as shown in Figure 2. The rate distortion (RD) performance will not be affected very much since the motion vector values in blocks 1 and 3 will be almost identical. The pseudo code for implementing the proposed method is shown in Figure 3.

The macroblock access pattern with the time stamp used when four macroblocks are processed in parallel is shown in Figure 4.

3.2 Load balancing

Scalability and load balancing are the two major concerns when parallelizing a program [11,12]. Scalability implies maximum number of threads that can be created for a

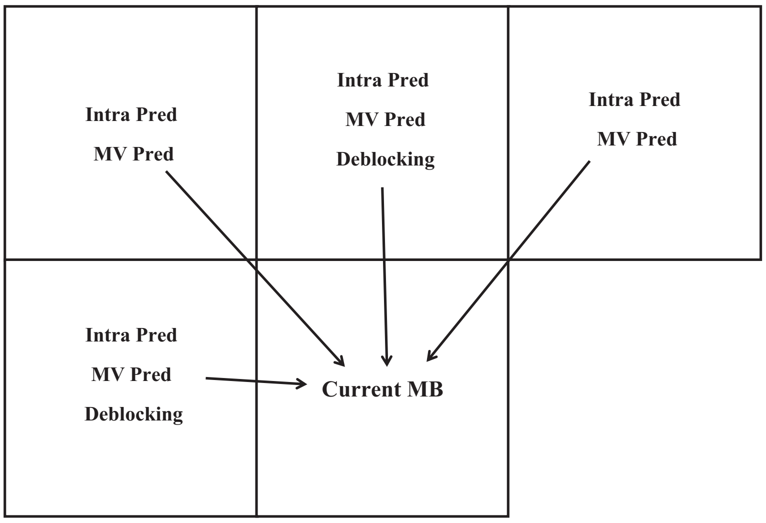


Figure 1 **Macroblock accessing pattern in wavefront method.**

parallel program. Load balancing is concerned with the issue of allocating the same amount of load to all the processing elements and ensuring that the execution times of these processors are nearly the same. The main challenge of implementing macroblock-level parallelism is to reduce the idle time among the processors. Processors should wait until the reference macroblocks are encoded except for the first macroblock. The balancing of tasks in the MB level is performed by determining the execution time of each function dynamically using the profiling. We have written a function call in the program, which monitors the threads' function with their execution times and makes all the threads active without going to idle state by any interruptions during the task execution. Load balancing is further improved by allowing each of the processing to access the structure and load the index of the macroblock that can be encoded by themselves without the need of a coordinator thread. A reference flag is created in the program for each thread to identify the status of the thread. The 0 and 1 statuses of a reference flag will indicate respectively whether the thread is active or not active. Each thread identifies starting and ending positions of its region based on its own thread ID with reference flag. Each thread shall only encode the macroblocks within its own region after all the data dependencies are resolved as shown in Figure 2. However, each thread will perform

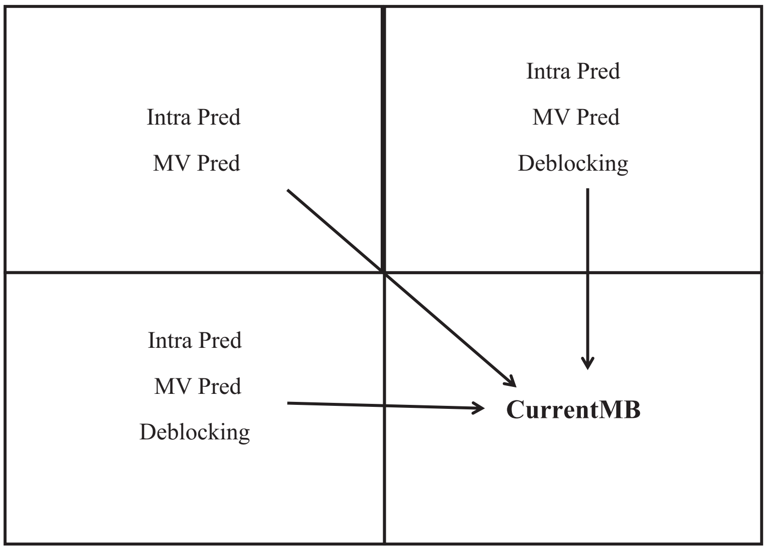


Figure 2 **Accessing pattern in the proposed method.**

```
check if the mode is available;

check if the mode is valid for the current slice;

if(available) {

            execute intra_prediction;

            rdcost_t[ipmode]=rdcost;

           }

 else rdcost_t[ipmode]=UNAVAILABLE;

      //store in the array that the ipmode is not available

/*store the rdcost in the array offset by their thread_id which is also ipmode*/

//end of parallel region

compare each rdcost in the array rdcost_t and save the minimum
```

*Note: This is a generalized pseudo code for concept visualization
and may need additional syntax to be fully functional.*

Figure 3 Pseudo code for the RD cost calculation.

entropy encoding if there are no macroblocks available for encoding. This idea is to effectively balance the workload among the threads, so that no threads will fall into an idle state. Hence, threads do not have to wait for the availability of macroblocks to encode within its region. In this way, the balancing of tasks in the MB level is achieved dynamically without going any thread to idle state, which solves the load balance problem.

The parallel algorithm of encoder is designed in such a way that each thread is independent and all the threads continue their execution by checking the status of the reference flag. All the threads can be independently executed without sharing of the cores by the threads. So, no data race condition occurs, and this will not incur any extra latency and eventually upgrade the overall encoding performance. Another benefit with this configuration is that for any thread, no extra waiting cycle is required to acquire the Mutual exclusion (Mutex) lock. Whenever there is no macroblock available for encoding, each thread will enter the shared region. In the shared region, only one thread is allowed to access at one time. Before entering the shared region, each thread will try to acquire a Mutex lock to gain an access and execute the code inside the shared region. In this way, no thread will fall into a waiting state, since there will be no repeated failures in acquiring the Mutex lock. This will not incur extra latency and eventually upgrade the overall encoding performance. Therefore, this solves the thread synchronization problem without the data race condition and thread locking overhead.

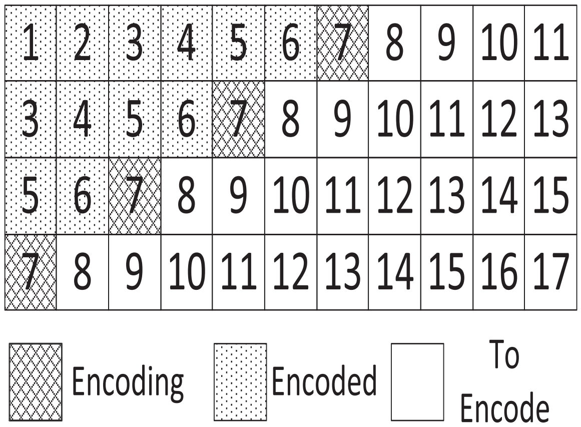


Figure 4 Macroblock access pattern with the time stamp.

3.3 Data partitioning

The data partitioning process assigns certain area of the frame to a particular processor core to reduce the data communication. Data partitioning can be generally separated into three variations, namely, horizontal, vertical, and dynamic variations. Figure 5 shows the horizontal

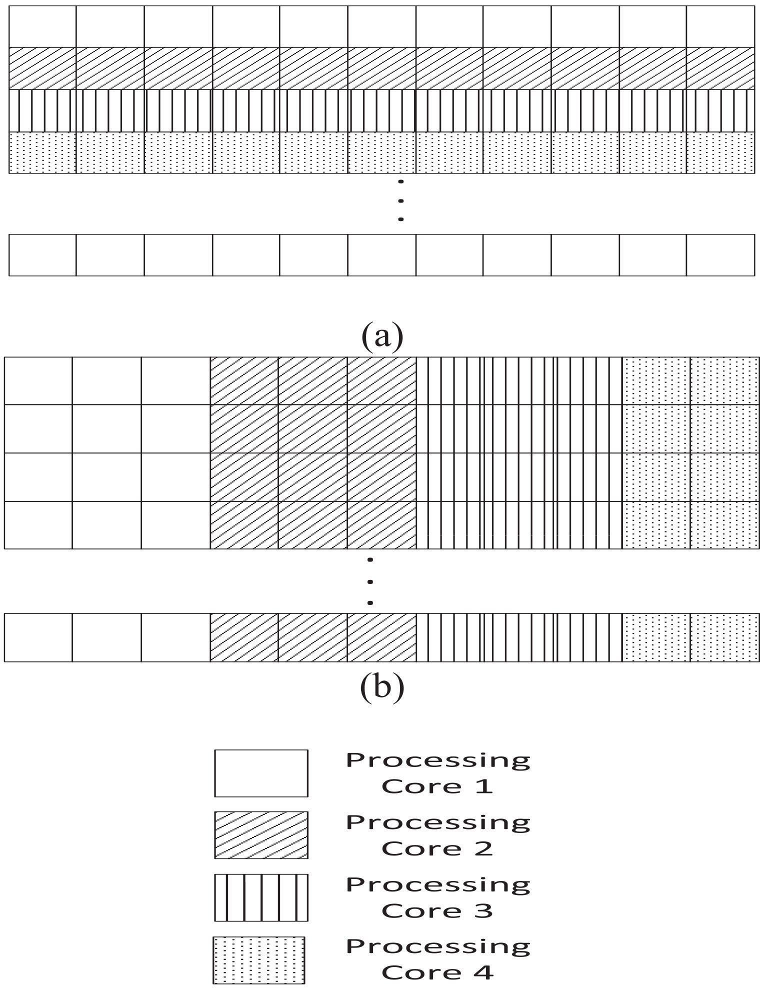


Figure 5 Data partitioning. (a) Horizontal variation. **(b)** Vertical variation.

and vertical variations of the data partitioning process [8]. Data communication is needed when the current and neighboring macroblocks are encoded by different processors.

3.4 Parallel system design

To design a parallel system, each of the processing cores should be able to determine which macroblock it has to encode next. This normally requires a coordinator to assign the macroblocks to each processor core. Figure 6 shows a general structure of macroblock-level parallelism for a single frame. Figure 6 also shows the flow of macroblock encoding process.

4 Design and implementation of parallel video coding

4.1 Macroblock-level parallelism design methodology

The proposed design utilizes a dynamic scheduling algorithm to stores the current encoded macroblock index in a temporary memory. This memory can be accessed by all the processing cores to determine which macroblock is to be encoded next. The processor that finishes first in a given cycle will update the indices of the macroblocks for the next cycle. Based on these indices, the macroblocks will be encoded. The indices of the encoded macroblocks will be automatically deleted from the memory. Figure 7 shows the process flow of the proposed macroblock-level parallelism method.

In our proposed method, we create threads in such a way that all the threads work without any starvation and without any interruption in the thread process even if there are any external stalls. Also, we use data communication among all the threads, so that even if any thread lags or stalls due to starvation or any other reason, it should be immediately notified to the remaining threads. The thread that finishes the task earlier will take care of any stalled thread task dynamically. In this way, we keep all threads running with full functionality without

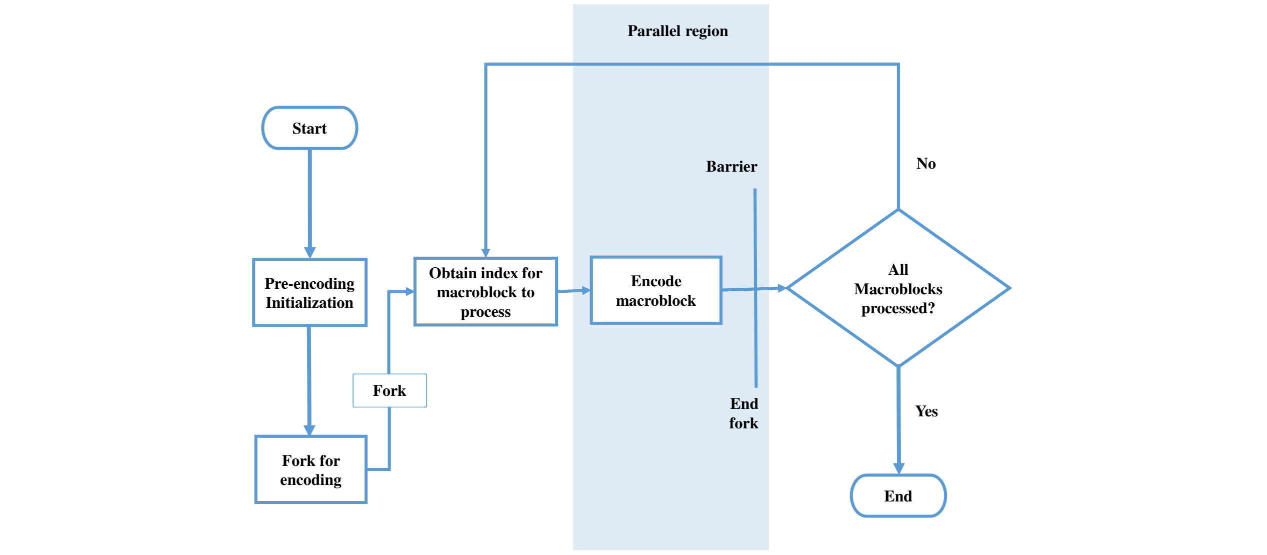

Parallel region
Start
Pre-encoding Initialization
Fork for encoding
Fork
Obtain index for macroblock to process
Encode macroblock
Barrier
End fork
All Macroblocks processed?
No
Yes
End

Figure 6 General macroblock encoding process.

any idle state, so that we get good load balance with all the available threads. In this paper, the macroblock access pattern is implemented with the dynamic data partitioning. In order to keep track of the macroblock status, we define a function in our program structure. The function in the program can identify which macroblock can be encoded after a specific macroblock is processed. By changing this function, it is possible to change the macroblock access pattern according to the encoding pattern. Figure 8 shows the macroblocks that are unlocked when the shaded macroblock is encoded. The idle processing core will load the indices of the unlocked macroblocks into the structure.

Even though the data partitioning is done dynamically, the processing core loads the index of the next encoding macroblock into the structure in a random order. Figure 9 shows the dynamic encoding process of the macroblocks with four threads. It may be noted that in this case, the work loads of the four threads are evenly balanced.

Macroblock-level parallelism is implemented using two structures termed as currSlice and currMB. Figure 10 shows the initialization of currSlice and currMB structures for parallelism using OpenMP. The processing core encodes the current macroblock by obtaining the specific macroblock index using the currMB structure. In our

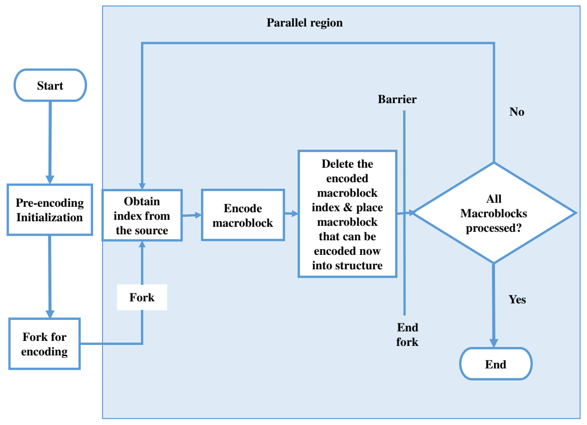

Parallel region
Start
Pre-encoding Initialization
Fork for encoding
Fork
Obtain index from the source
Encode macroblock
Delete the encoded macroblock index & place macroblock that can be encoded now into structure
Barrier
End fork
All Macroblocks processed?
No
Yes
End

Figure 7 Flow chart of the proposed dynamic parallel design.

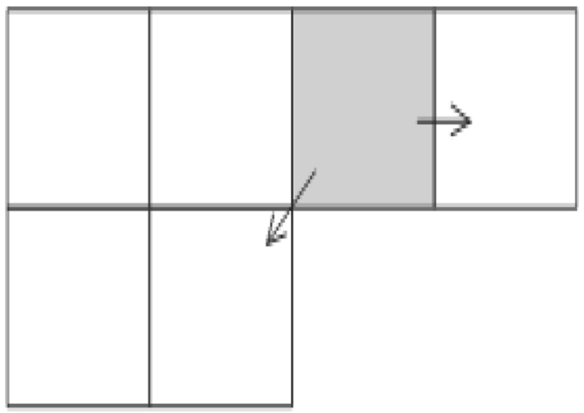


Figure 8 Macroblocks that are unlocked.

implementation, the initialization of the currSlice and currMB as well as the encoding sequence are modified to facilitate dynamic parallelism. The pseudo code for the parallelized currSlice and currMB structure initialization is shown in Figure 10.

4.2 Scheduling

In the proposed dynamic scheduling algorithm, no additional thread (processing core) is used for task scheduling; instead, one of the available thread is assigned to do this scheduling task. Due to the problems such as data dependency and control dependency in the shared memory system, a barrier mechanism is applied to parallel programming to retain programming execution order and to prevent data corruption. Barrier is a type of coordination method where it defines a point in the program where any thread that reaches there first has to wait for the other thread's completion before proceeding on. It is useful to prevent data from being updated prematurely by other thread. The scheduling algorithm attempts to minimize the barrier wait. It is possible to enforce a flush system to monitor the processing times of all the parallel threads during a cycle. Figure 11 shows the structure of parallelism with barrier. The balancing of tasks in the macroblocks is performed by determining the execution time of each function dynamically to resolve the load imbalance problem. The design supports a variable number of processing cores to utilize more macroblocks at a time. Each of the processing cores receives its macroblock index by dynamically detecting the macroblock status.

In order to calculate the index of the macroblock, a new variable termed as time cycle is introduced into the algorithm. This variable stores the count of macroblocks that are processed within the parallel region and outside the parallel region. Figure 12 shows the pseudo code illustrating the process of determining the macroblock index. The macroblock indices are numbered from left to right and then from top to bottom. The calculations for determining the index values are done for the master and slave threads separately. Equations 1 and 2 give the index value (thread status) for the master and slave threads, respectively. The index values are determined dynamically as shown in the pseudo code in Figure 12.

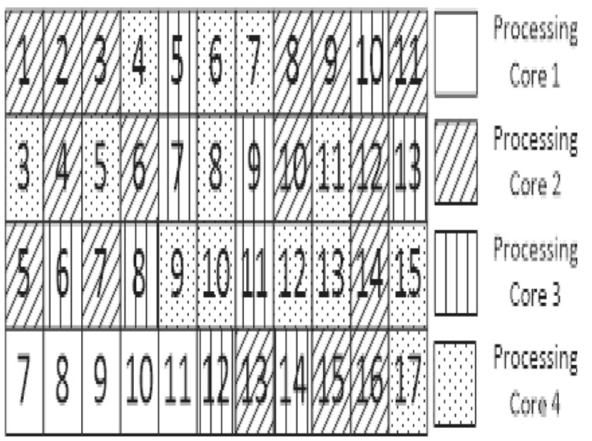


Figure 9 Output of the proposed DMBRP design pattern.

where t represents the time cycle number, t_{id} represents the thread ID, and MB represents the number of macroblocks per row.

4.3 Data structures for memory optimization

The source code of JM 18.0 is implemented with pointers all over the locations [13]. All the structures are tangled together by these pointers which work fine in serial mode. However, these pointers cause problems for memory access 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 memory location pointed by the pointer. There are actually two problems to be taken into account to tackle this issue. The first problem occurs when a structure in the higher hierarchy has a pointer pointing back to another structure in the lower hierarchy. The second problem occurs when a structure is having two pointers one pointing to a structure in the higher hierarchy and the other pointing to a structure in the lower hierarchy. In the original source code of JM 18.0 [13], pointers can be replaced with suitable data structures. To replace a pointer with a suitable data structure, the encoding parameter structure p_Enc (JM 18.0 encoder code) is made as a global variable and all the structures can be directly or indirectly linked to the p_Enc structure.

$$\text{index} = \begin{cases} t & \text{if} \quad t < \text{MB} \\ \frac{t}{\text{MB}} \times n \times \text{MB} + (t \mod \text{MB}) & \text{otherwise} \end{cases} \tag{1}$$

$$\text{index} = \begin{cases} (t_{\mathrm{id}} \times \text{MB}) + (t - (t_{\mathrm{id}} \times 2)) & \text{if} \quad t < (\text{MB} + (t_{\mathrm{id}} \times 2)) \\ \frac{t-(t_{\mathrm{id}}\times 2)}{\text{MB}} \times (n + t_{\mathrm{id}}) \times \text{MB} + (t - (t_{\mathrm{id}} \times 2) \mod \text{MB}) & \text{otherwise,} \end{cases} \tag{2}$$

```
allocate memory space for macroblock address set;
obtain all the macroblock addresses;
allocate memory space for currSlice and currMB for each thread;
#pragma omp parallel
/* the variable clause is not displayed as it is too long */
{
thread_id = omp_get_thread_num();
   initialize currSlice_thread[thread_id]and    currMB_thread[thread_id];
}
/* Take note here; since the currSlice is separated into individual threads, some of
the line require coordination such as critical region to execute properly.*/
```

Figure 10 **Pseudo code of the parallel MB structure initialization.**

The p_Enc structure provides more flexibility to access all the structures, for example, whenever there is a need to make a structure private, the value of that structure is copied to another memory location. In order to store the macroblock's references, the proposed encoder is implemented with advance data structures, which eliminates the need for the extra picture buffer. Using the Intel Parallel Studio's 2011 Vtune amplifier (Intel Corp., Santa Clara, CA, USA), we observed that memset function is taking most of the time to set memory locations for specific values. The memset function in JM 18.0 encoder [13] makes use of calloc function, for allocation of memory locations. We have replaced calloc function by malloc, which allocates memory and also reduces the memory stalls [14,15]. The malloc function reduces the runtime and improves speedup value in a significant manner. To solve the scalability and latency issues that occur for large volumes of data, the 'non-temporal stores' function [14] is used which stores data straight to the main memory without going through cache allocation which makes faster memory access. The use of 'non-temporal stores' function provides better scalability. It is expected that when a core

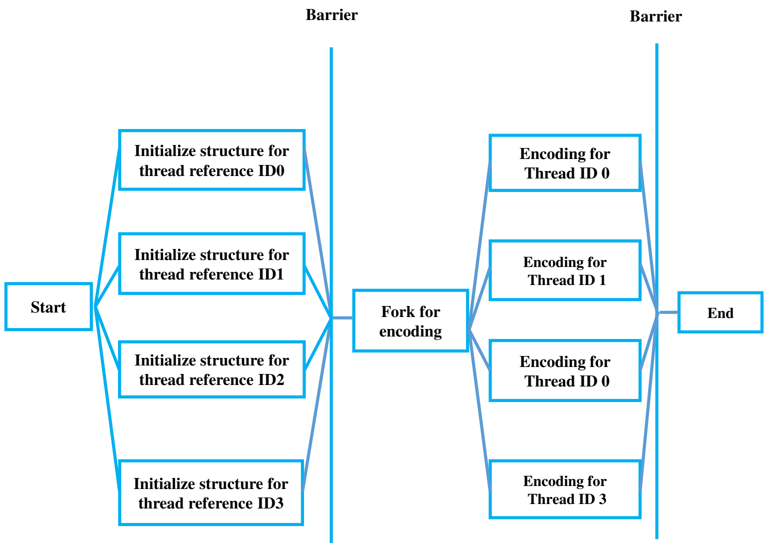

Barrier
Barrier
Start
Initialize structure for thread reference ID0
Initialize structure for thread reference ID1
Initialize structure for thread reference ID2
Initialize structure for thread reference ID3
Fork for encoding
Encoding for Thread ID 0
Encoding for Thread ID 1
Encoding for Thread ID 0
Encoding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

Figure 11 **Structure of the proposed parallel design with barrier.**

```
if (macroblock index is in the second column of the frame)
{
        load (index + 1);
        load (index + macroblock per row - 1);
}
else if (macroblock index is at the last column of the frame)
{
}
else if (half of the macroblocks are encoded)
if(deblocking filter and quarter-pixel interpolation done)
      load (next frame’s first macroblock);
else
        load (index + 1);
```

Figure 12 Pseudo code of macroblock index calculation.

processes a macroblock, its cache will be filled quickly, so there is a possibility of more number of cache misses taking place. In order to resolve cache miss issue, each processor core dynamically keeps track of its cache status. Whenever a cache is full, its data will be flushed to the secondary-level cache. In this way, the cache usage can be optimized effectively. Table 1 shows the number of cache accesses and cache misses per frame with respect to data cache (L1) for first processor core. The cache accesses and misses shown in Table 1 represent the values obtained when encoding high-motion (Rush_hour) video sequences with QCIF, CIF, SD, and HD resolutions. The results shown in Table 1 indicates that the miss rates are much lower compared to those obtained by other researchers [5-8].

MB-level parallelism is performed on all parallel threads without any dependencies by dynamically detecting the threads’ status using reference flag so that the subsequent frames processed by each thread would not depend on the results of other threads. Each thread will access only a specific portion of the memory (of the core) without altering the existing memory mapping structure. Each thread writes the results prior to reading the results. This would not affect the processor core’s results to access the external DRAM and will not affect the memory bandwidth bottleneck issues. For multicore architectures, this is one main benefit to enable the flexible shared memory subsystem. This minimizes the data exchanges between pipeline stages and enables non-blocking handshaking between tasks of a multicore architecture.

Table 1 Cache performance metrics

Video sequences	Cache accesses	Cache misses
Rush_hour_QCIF	358×10^6	3.8×10^6
Rush_hour_CIF	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

4.4 Encoding

In our implementation, we use looping structure for macroblock encoding. Once the last macroblock in a slice is encoded, the end of slice flag will be activated to end the loop. The motivation for this method of looping is to support the flexible macroblock order (FMO) structure in the H.264 standard. Figure 13 shows the pseudo code of the loop with the macroblock encoding.

In the parallelized loop, the macroblock index cannot be incremented sequentially, since each thread needs immediate access to specific macroblock address. To solve this problem, the macroblock address has to be pre-initialized. Thus, each thread can fetch its macroblock according to the macroblock index. The end_of_slice detection mechanism is also changed slightly to accommodate the pre-computed macroblock address scheme. The pseudo codes for the parallelized macroblock encoding are shown in Figure 14. It may be noted that the time cycle is updated outside the parallel region, since the macroblock index is a function of time cycle. It is impossible for a thread to

```
while(end_of_slice==FALSE)
{
          load the currently encoding macroblock into  currMB structure;
         encode the macroblock;
end macroblock encoding;
if(recode_macroblock==FALS
{
       load the next macroblock address;
       if(last macroblock) end_of_slice=TRUE;
       {
          updates statistic;
          prepare next macroblock
       }
else recode macroblock;
}
}//end of while loop
```

Figure 13 **Pseudo code of macroblock encoding scheme.**

obtain its next macroblock address unless the time cycle is incremented. This will act as an explicit barrier in addition to the implicit barrier inside the OpenMP structure. This is due to the fact that the OpenMP implicit barrier requires that all threads must exit the parallel region before proceeding to the next cycle.

4.5 Simulation Environment

In this implementation, an Intel i^7 platform is used for simulating a four-physical-core system and as an eight-logical-core system using hyper-threading technology. It is assumed that each core has an independent data cache (L1) and data can be copied from additional

```
while(end_of_slice==FALSE)
{
#pragma omp parallel /*the variable
clause is not displayed as it is too long*/
       {    //start of parallel region
              obtain the thread identifier;
obtain the macroblock index for the
thread;
 if(macroblock index =macroblock in
picture-1)
     end_of_slice = TRUE;
```

```
if(macroblock index != IDLE)
{
        load the macroblock content;
              encode the macroblock;
              end macroblock encoding;
            if(recode_macroblock==FALSE)
           {
                updates statistic;
                prepare next macroblock
           }
/*take note that the statistic that are being updated will be
accessed by all thread, and mostly at the same time, to
prevent this, a critical segment must be used*/
         else recode macroblock;
      }
    }//end of parallel region
    time_cycle++;
}//end of while loop
```

Figure 14 **Pseudo codes for the parallelized macroblock encoding.**

caches (L2 and L3) through four channels. To record the encoder's elapse time, all existing native services and processes in the cores are 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 [13] and compiled with Microsoft Visual Studio 2010 using Intel i7 platform (Redmond, WA, USA) as described below: Intel Core™ i^7 CPU 930, running at 2.8GHz 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 testing 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 are disabled.
- Visual effect is set to get better performance.
- Power setting is changed to "Always on" for all devices.
- All extra windows features are removed with the exception of Microsoft.net framework.

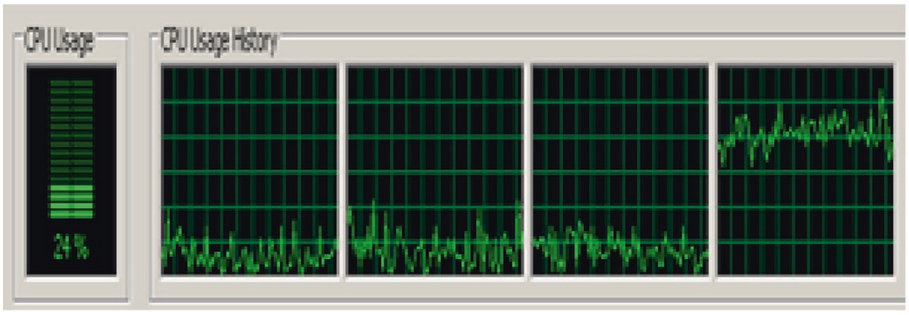
All simulations are performed under this controlled environment and the encoder's elapsed time is recorded using Intel Parallel Studio's 2011 Vtune Amplifier and AMD code analyst. The memory leaks are analyzed using Intel Parallel Inspector 2011. The parallel programming is implemented using OpenMP technique. The resolutions of the video sequences used in the simulation are QCIF, CIF, SD, and HD resolutions. The scalability is tested by increasing the number of processing cores and applying homogeneous software optimization techniques to each core.

5 Experimental results

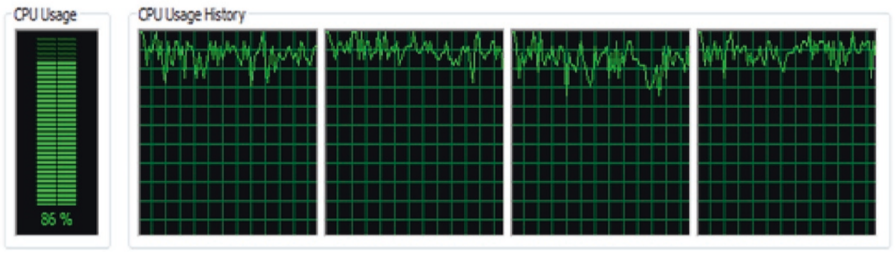
The H.264 reference software JM 18.0 is implemented in sequential with C language. After modifying the JM 18.0 with some optimized C language data structures, JM 18.0 is parallelized by using OpenMP. The simulation is performed using a high-motion video sequence (rush_hour) with different resolutions such as QCIF, CIF, SD, and HD. In this implementation, 300 frames are encoded for all sequences. For each of this resolution, a variable number of threads from 2 to 8 are tested.

5.1 CPU performance

Figure 15A,B shows the CPU usage graphs before and after parallelization, respectively. It is observed that all the four cores are equally balanced after implementation of the MB-level parallelism.



(A) CPU usage before parallelization of JM18.0



(B) CPU usage after parallelization of JM18.0

Figure 15 Performance of CPU usage (A) before parallelization and (B) after DMBRP implementation.

5.2 Speedup performance

Figure 16A,B,C,D shows the speedup values achieved for different resolutions.

From Figure 16A,B,C,D, it is clear that speedup values close to the number of threads created are achieved for all resolutions. For example, for HD, a speedup value of 1.973 is achieved using multicore system with two threads, which is very close to the maximum speedup value of 2 for a two-core system. A speedup value of 3.95 is achieved using a multicore system with four threads, which is very close to the maximum speed up of 4 for a four-core system. A speedup value of 7.71 is achieved using multicore system with eight threads. This value is not very close to the maximum possible speed up value of 8 even though

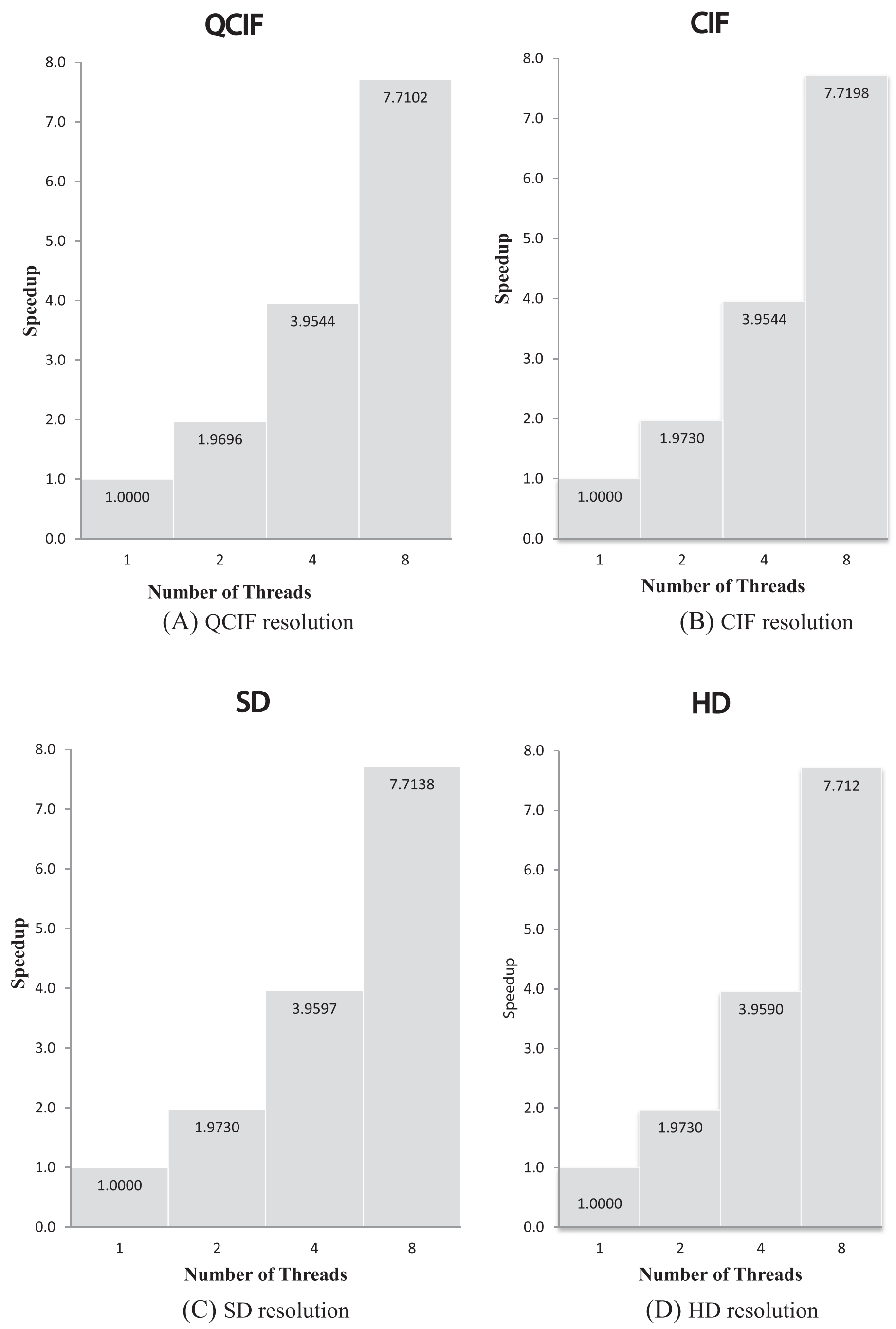


Figure 16 Performance of speedup with different resolutions. (A) QCIF, **(B)** CIF, **(C)** SD, and **(D)** HD resolutions.

Table 2 Speedup comparison with different HD video sequences with different configurations

Video sequences (HD resolution)	Two threads	Four threads	Eight threads
Life	1.98	3.95	7.8
Factory	1.97	3.94	7.72
Riverbed	1.978	3.953	7.75
Station2	1.976	3.955	7.76

Table 3 Speedup comparison

Implementation methods	QCIF	CIF	SD	HD
Wavefront method	3.17	3.08	No	No
MBRP parallelism	3.32	3.32	3.32	No
MBRP with data partitioning	3.8	3.7	3.68	3.54
Dynamic data partitioning	3.89	3.87	3.8	3.59
Our proposed method	3.954	3.954	3.959	3.959

dynamic scheduling is used to reduce the barrier. This is because that though Intel i7 platform has eight logical cores, it has only four physical cores. We note from Figure 16A,B,C,D that without parallelism, a speedup value of 1 is achieved for all multicore processors, since all are using a single thread only. We also observe from Figure 16A,B,C,D that it is possible to achieve significantly higher speedup values (closer to theoretical values) using the macroblock-level parallelism. The results obtained on the multicore system with two threads, four threads, and eight threads (hyper-threading) demonstrate significant speedup improvements. The speedup values given in Table 2 are calculated using Amdahl's law shown in Equation 3 [13]:

$$\text{Speedup} = \frac{1}{r_s + \frac{r_p}{n}}, \tag{3}$$

where r_p is parallel ratio, r_s is serial ratio $(1 - r_p)$, and n is the number of threads. Figure 17 shows the speedup values obtained using dynamic MB-level parallelism for four different video sequences (life, Factory, riverbed, Station 2) with HD resolution.

The speedup values obtained using different video test sequences with HD resolution using threads, four-thread and eight-thread configurations, are shown in Table 2.

The speedup values obtained by using different methods are shown in Table 3. The Table 3 values are shown only for four-thread configuration to compare the other methods implemented, since the other methods implemented only for four threads. From Table 3, it is observed that the proposed method achieves significantly higher speedup values compared to those obtained by other researchers [6-9]. We can also note from Table 3 that the results obtained by the proposed method are consistent for all the resolutions unlike the varying results obtained by other researchers.

Figure 18 shows the results of the peak signal-to-noise ratio (PSNR) values obtained using the proposed method for various threads using multicore system for QCIF, CIF, SD, and HD resolutions. It is clear from Figure 18

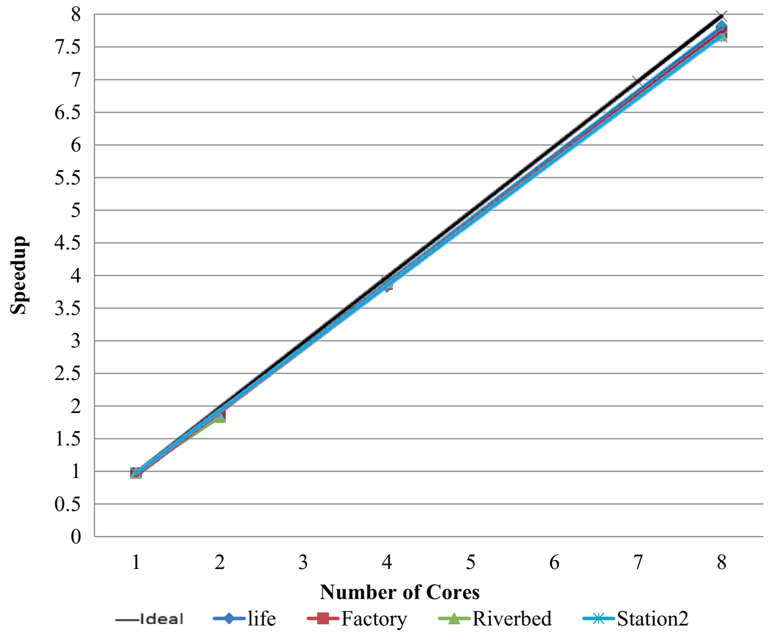


Figure 17 Speedup for HD resolution with four different video sequences.

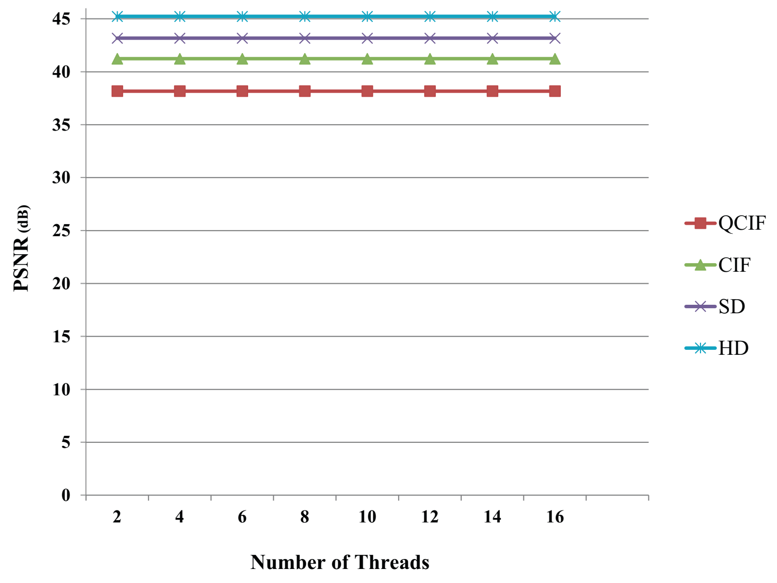


Figure 18 PSNR vs. number of threads.

that there is no loss of video quality for all resolutions when the number of threads is increased from 2 to 16 (i.e., the PSNR value remains constant). For PSNR calculation, we created more threads (up to 16) to see the effect of the video quality compared to the effect of speedup.

6 Conclusions

A new scalable method based on macroblock-level parallelism has been presented. The proposed method has advantages such as good load balancing, scalability, and higher speedup values compared to the existing methods. Unlike the existing methods where one thread is specifically used for the purpose of assigning macroblock indices, the proposed method makes use of all the threads to encode the macroblocks leading to good load balancing. This is achieved by using a dynamic scheduling technique. In order to obtain better scalability, the proposed method makes use of a dynamic data partitioning method. Experimental results show that speedup values close to theoretical values can be obtained using the proposed method. Speedup values of 1.97, 3.96, and 7.71 have been obtained using two, four, and eight threads, respectively. Furthermore, it has been found that the speedup values remain constant for QCIF, CIF, SD, and HD resolutions. These values are very close to the theoretical speedup values without degradation in the video quality. Although, the focus of this paper is on the use of H.264 encoder, the proposed technique can be applied to other video codecs and computationally intensive applications to speedup the process.

Competing interests
The authors received funding from Intel Technology Sdn.Bhd for this research.

Acknowledgements
The authors would like to thank Intel Technology Sdn.Bhd for their sponsored funding for this research.

Received: 3 January 2014 Accepted: 1 September 2014
Published: 19 September 2014

References
1. IE Richardson, *The H.264 Advanced Video Compression Standard*, 2nd edn. (Wiley, London, 2010)
2. C Luo, J Sun, Z Tao, The research of, H.264/AVC video encoding parallel algorithm, in *Second International Symposium on Intelligent Information Technology Application*, vol. 1 (Shanghai, China, 21–22 Dec 2008), pp. 201–205
3. S Sankaraiah, HS Lam, C Eswaran, J Abdullah, *GOP Level Parallelism on H.264 Video Encoder for Multicore Architecture*, vol. 7. (IACSIT, Singapore), pp. 127–132
4. J Lee, S Moon, W Sung, H.264 decoder optimization exploiting SIMD instruction. Proc. IEEE Asia Pac. Conf. Circuits Syst. **2**, 1149–1152 (2004)
5. Z Zhao, P Liang, A highly efficient parallel algorithm for H.264 video encoder. ICASSP. **5**, 489–492 (2006)
6. S Sun, D Wang, S Chen, A highly efficient parallel algorithm for H.264 encoder based on macro-block region partition. High Perform. Comput. Commun. Lect. Notes Comput. Sci. **4782**, 577–585 (2007)
7. J Kim, J Park, K Lee, J Tae Kim, Dynamic data partition algorithm for a parallel H.264 encoder. World Acad. Sci. Eng. Technol. **72**, 350–353 (2010)
8. S Sankaraiah, HS Lam, C Eswaran, ed. by T Herawan, M Mat Deris, and J Abawajy, Junaidi Abdullah Parallel full-HD video decoding for multicore architecture, in *Lecture Notes in Electrical Engineering (LNEE)*, vol. 285 (Springer, Singapore), pp. 317–324
9. Y Chen, EQ Li, X Zhou, S Ge, Implementation of H.264 encoder and decoder on personal computers. J. Vis. Commun. Image Representation. **17**(2), 509–532 (2006)
10. S Ge, X Tian, YK Chen, Efficient multithreading implementation of H.264 encoder on Intel hyper-threading architectures, in *Proceedings of the 2003 Joint Conference of the Fourth International Conference on Information, Communications and Signal Processing, 2003 and the Fourth Pacific Rim Conference on Multimedia*, vol. 1 (IEEE, Piscataway, 2003), pp. 469–473

11. S Sankaraiah, HS Lam, C Eswaran, Junaidi Abdullah, Performance Optimization Of Video Coding Process On Multi-Core Platform Using GOP Level Parallelism. International Journal of Parallel Programming (Springer). **42**(6), 931–947 (2014)
12. YIL Kim, JT Kim, S Bae, H Baik, HJ Song, H.264/AVC decoder parallelization and optimization on asymmetric multicore platform using dynamic load balancing, in *the IEEE international conference on multimedia and expo* (Hannover, Germany, 26 April–23 June 2008), pp. 1001–1004
13. Fraunhofer Heinrich Hertz Institute, JM 18.0 (2014). http://iphome.hhi.de/suehring/tml/download/old_jm/jm.18.0.zip. Accessed 30 Nov 2011
14. S Taylor, *Optimizing Applications for Multi-core Processors: Using the Intel Integrated Performance Primitives*, 2nd edn. (Intel Press, Santa Clara, 2007)
15. R Gerber, AJC Bik, KB Smith, X Tian, *The Software Optimization Cookbook: High-Performance Recipes for IA-32 Platforms*, 2nd edn. (Intel Press, Santa Clara, 2005)

doi:10.1186/1687-6180-2014-145
Cite this article as: Sankaraiah *et al.*: **Scalable video encoding with macroblock-level parallelism.** *EURASIP Journal on Advances in Signal Processing* 2014 **2014**:145.