INSTITUTO TECNOLOGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

MONTERREY CAMPUS
ENGINEERING SCHOOL AND INFORMATION TECHNOLOGIES

TECNOLÓGICO DE MONTERREY®

PARALLELIZATION OF THE MOTION DETECTION STAGE IN THE H.263 ENCODER

THESIS

PRESENTED AS A PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE WITH MAJOR IN ELECTRONIC ENGINEERING
(ELECTRONIC SYSTEMS)

BY

ISRAEL CASAS LOPEZ

MONTERREY, N. L., MEXICO, OCTOBER, 2011

INSTITUTO TECNOLOGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

MONTERREY CAMPUS
ENGINEERING SCHOOL AND INFORMATION TECHNOLOGIES

PARALLELIZATION OF THE MOTION DETECTION STAGE IN THE H.263 ENCODER

THESIS

PRESENTED AS A PARTIAL FULLFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE WITH MAJOR IN ELECTRONIC ENGINEERING
(ELECTRONIC SYSTEMS)

BY

ISRAEL CASAS LOPEZ

MONTERREY, N. L., MEXICO, OCTOBER, 2011

INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

MONTERREY CAMPUS
ENGINEERING SCHOOL AND INFORMATION TECHNOLOGIES

PARALLELIZATION OF THE MOTION DETECTION STAGE IN THE H.263 ENCODER

THESIS

PRESENTED AS A PARTIAL FULLFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE WITH MAJOR IN ELECTRONIC ENGINEERING (ELECTRONIC SYSTEMS)

BY

ISRAEL CASAS LÓPEZ

MONTERREY, N.L., MÉXICO. OCTOBER, 2011

INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

MONTERREY CAMPUS
ENGINEERING SCHOOL AND INFORMATION TECHNOLOGIES

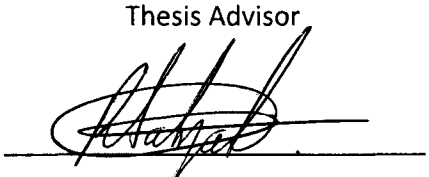
The members of the thesis committee hereby approve the thesis of Israel Casas López as a partial fulllment of the requirements for the degree of

Master of Science with major in Electronic Engineering (Electronic Systems)

Thesis Committee

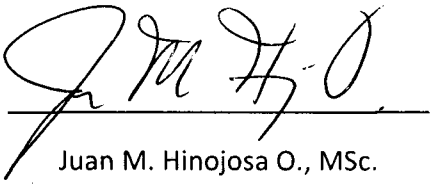
Alfonso Ávila Ortega, Ph.D.

Thesis Advisor



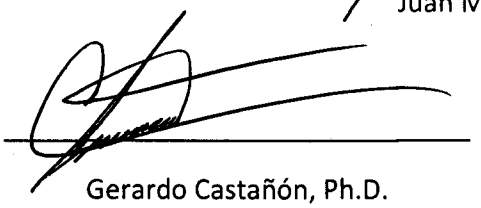
Luis Ricardo Salgado Garza, MSc.

Synodal



Juan M. Hinojosa O., MSc.

Synodal



Gerardo Castañón, Ph.D.

Director of the Master of Science Programs in Electronics and Automation

October, 2011

PARALLELIZATION OF THE MOTION DETECTION STAGE IN THE H.263 ENCODER

BY

ISRAEL CASAS LÓPEZ

THESIS

PRESENTED TO THE ENGINEERING SCHOOL AND INFORMATION TECHNOLOGIES

THIS THESIS IS A PARTIAL FULLFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE WITH MAJOR IN

ELECTRONIC ENGINEERING

(ELECTRONIC SYSTEMS)

INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

MONTERREY CAMPUS

OCTOBER, 2011

Dedico este trabajo de tesis a mis padres por ser ellos quienes me guiaron en este camino, y de quienes recibí una excepcional educación, misma que ah sido la base en la formación del hombre que hoy en dia soy.

Esto es el fruto de lo que me han enseñado.

Gracias.

Acknowledgements

To my sister, Samantha, for her admirable wisdom and attention on me.

To my thesis advisor, Alfonso Avila, Ph.D., for sharing with me his expertise and orientate me in all academic affairs and who became a figure to follow.

To the professors Sergio O. Martinez Ph.D, Juan Hinojosa MSc., Ricardo Salgado MSc., and to all of the professors in the computer department for always pursuing students to wide and achieve their goals.

Special thanks to Juan Alberto Gonzalez, current PhD student, for his unconditional support on academic and personal matters and to Ernesto Leiva for his unconditional frienship.

To Miss Adriana Sanchez for her support over these years, and to Chava, Jimmy, Hugo, Marco and Arturo for their technical attention.

And special thanks to all of my classmates and partners who became my friends thank you all Alberto, Alfredo, Carolina, Cindy, Christian, Francisco, Julian, Marco, Octavio, Ruben, Salvador and to everyone that made this a special time in my life.

ISRAEL CASAS

Instituto Tecnologico y de Estudios Superiores Monterey

October, 2011

Abstract

Multiprocessing has become an irreversible trend in embedded systems, but research on this area has not achieved efficient methods nor techniques to fully strength multicore capabilities. Multicore computers are present on a variety of applications such as video games, multimedia, military applications, servers, supercomputers, etc., but systems running these applications require to develop new strategies to overcome the principal concerns: power dissipation, memory usage, scheduling and mapping of application.

An ideal multicore system would decrease computation time by half when doubling the number of CPUs. This is the goal of every study on parallelization of applications, but reality is different. Accessing memory creates long delays in computers and the number of accesses and time delays depend on the nature of each application. The applications, suitable for multicore systems, are divided on two principal groups: (1) applications with tasks naturally independent of each one such as requests to servers and (2) applications with strong loop nesting and/or high computation demands such has multimedia devices and modeling for scientific purposes.

This work studies the second group of applications mentioned above. The target is to find the most effective way to distribute the application into multicore systems with 2, 4, 8, 16 and 32 CPUs systems. The selected multimedia application is the H.263 standard. First, a profiling work reveals H.263 code sections that contribute with the biggest demand of computer work. Next, two different loads are created varying video quality parameters. Finally, the parallelization approaches are applied to the loads just created. This approaches highlight unique features of codes that permit improvement on parallelization job, these characteristics are: size of task, operation per task and memory accesses.

A profiling work showed that the Sum of Absolute Differences function consumes 55.6% off all program execution. Experiment results showed an improvement of 30% with the light load over the single CPU run for the first parallelization approach, and an improvement of 43% over a single CPU run for the second approach on the heavy load

using two processors. For the heavy load level an improvement of 63% is present with the thirty-two processors system.

Keywords

Static analysis, profiling, data partitioning, functional partitioning, slice partitioning, macroblock partitioning, parallelization, H.263, multimedia, mapping, scheduling.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The motivation of this study is the project Design Space Exploration Of Memory Intensive Embedded Systems, a research work performed by The University of California at Irving and Tecnologico de Monterrey at Monterrey lead by Dr. Nikil Dutt and Dr. Alfonso Avila respectively. The origin of this thesis dated back on mid 2010 meetings with research leaders at the UCI promoted by the UCI-MEXUS financial support. The principal goals of this international project are to study memory accesses on embedded systems with scratch pad and hierarchy memories, power consumption and memory mapping.

This thesis focuses on the development of a parallelization approach for multimedia applications where motion estimation and compensation play an important role on codification. The selected target-application is the H.263; this codec has an appropriate complexity to validate the proposed parallelization approach.

This parallelization approach first locates code sections that contribute with the greatest workload on the computer program. Second, the parallelization extracts and analyzes those sections in order to identify loops, and parallelization opportunities. Then, the parallelization decomposes the sections into different targets to be distributed into the available resources. The parallelization has three approaches. Every one of these approaches has its important characteristic as: using the full memory space of SPMs, reusing data and reducing CPU idle time.

This work starts with a code profiling to identify the code sections that contribute with the major time during program execution, as in [4]. This thesis also goes over the problem of determining the number of cores needed to execute a given task and it leaves intact those sections that have no significant contribution to execution time.

Additionally, this work covers macroblock grouping as in [4] to efficiently use available cores. The load imbalance problem is also treated on this study as in [11]. This work is a study that separates the two dependency types (intra and inter) and assigns to each core the necessary data to compute the task with a lower need for memory transfers.

Another important relation with previous work is on the evaluations for motion estimation as in [2]. This study centralized the parallelization on the SAD (Sum of Absolute Differences) function. This related work introduces the concept of search range, which consists on identifying the necessary macroblocks for the computation of each SAD operation.

Furthermore, this thesis work uses data decomposition and does parallelization at macroblock level. Data decomposition consists on separating data on different sections and computes all necessary functions over them. Parallelization at macroblock level is a data parallelism where data sections have the size of a macroblock (16 x 16 pixels).

The studies discussed above have provided evidence that data parallelization at macroblock level is the best alternative to perform a distribution for multimedia parallel systems. Task size and data decomposition for multicore systems depend directly from the particular behavior of the application. Selecting the correct size of task and data splitting results in a high cost-effective parallel system; an incorrect selection lets to a poor system inefficiently using resources. This thesis does a contribution based on the specific aspects of the H.263 codec.

1.1 Problem Statement

We want to take advantage of multicore technology in multimedia applications, a challenging and underdevelopment area, by efficiently distribute data and tasks among available resources. An incorrect data and task partitioning usually result in performance degradation instead of improvement after increasing the number of cores. We will use data partitioning, an efficient and validated decomposing criteria, to correctly distribute the H.263 codec in the system.

1.2 Objectives

This work proposes an approach to determine the optimal partitioning of code and data of a H.263 algorithm using data decomposition. This approach was applied to the H.263 encoder particularly to the SAD, a metric to evaluate motion estimation and compensation stages.

This work centralized on the problem to select the amount of information to be computed for each processor, to correctly select the size of each of these data groups based on the search window size for the SAD operations.

1.3 Contribution

The main contributions of this thesis effort are: (1) the identification of H.263 code sections suitable for parallelization, (2) a parallelization framework for the H.263 (3) introduce a criteria for better evaluation of the parallelization scheme based on two load sizes: light and heavy load sizes and (4) analyze the results based on the relation between task execution time, task size and number of CPUs.

1.4 Related Work

Multiprocessor systems are now a target of large number of studies for the effective use of multicore systems. Parallelization can be made using different tools and methodologies. Figure 1-1 shows the big picture of parallel system distribution, parallelization can be engaged from different perspectives: function decomposition, data decomposition, mapping and scheduling algorithms and compilation process. The problem of parallelization can start from different methodologies and all of this solution converges at the end.

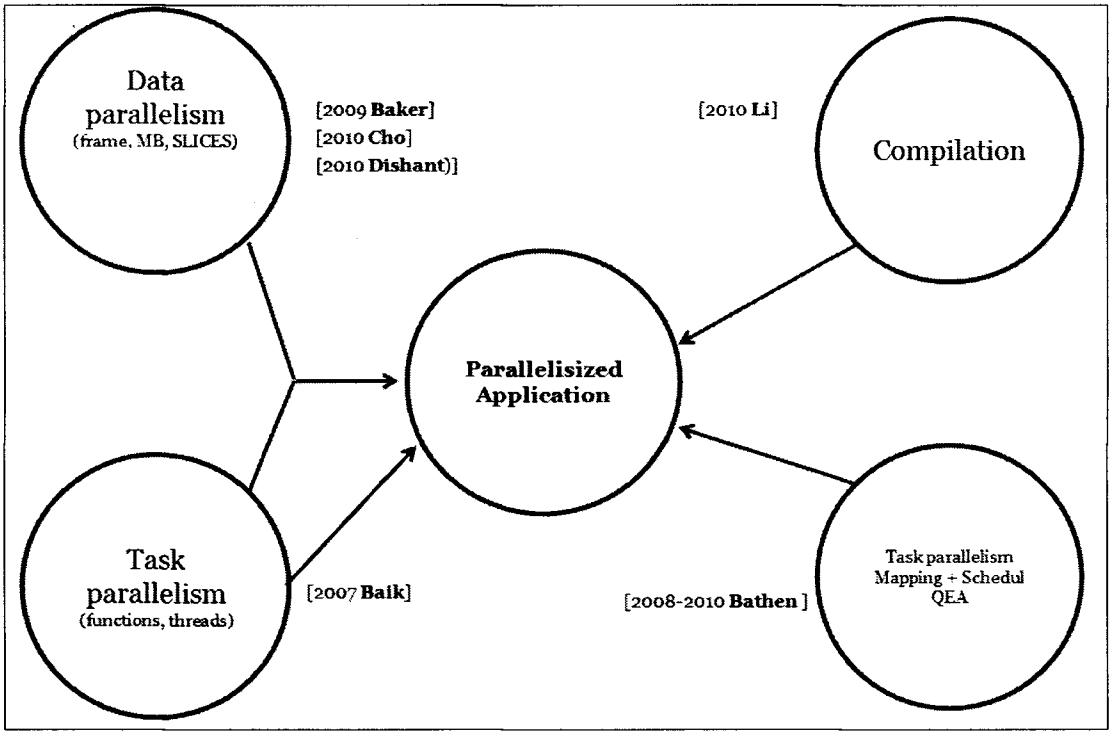


Figure 1-1: Parallelization methodologies

Task parallelism

Baik *et. al* implement a parallelized H.264 decoder on the CELL Broadband Engine (BE) processor based on profiling results [4]. The Cell BE processor is a multicore system developed jointly by Sony, Toshiba and IBM, it includes a main processor called Power Processing Element (PPE) and eight co-processors named Synergistic Processing Elements (SPE). This work presents a hybrid partitioning technique that combines both functional and data partitioning. This study does a profiling of code to identify code sections that contribute with the mayor fraction of execution time. It focuses on the problem of determining the number of processors needed to compute a given task on the application. This study does motion compensation at sub-block levels, such as 16x16, 8x8, 4x4 and 16x8. Using SIMD (Single Instruction, Multiple Data) instructions, the decoder is optimized lowering the dependencies.

Furthermore this reference presents the two principal parallelization techniques: functional and data parallelism. Functional partitioning decomposes functions into tasks that later can be executed in parallel on a multiprocessor system. Data partitioning is a division of a picture data into partitions and process each partition data on a single processor. Experiments were made on four 1920 x 1080 progressive HD stream using 1 PPE and 4 SPEs of a CELL BE processor at 2.5Mbps. Results show a parallelized decoder running about 3.5 times faster than single core with frame rate average of 20 fps (frames per second).

Data parallelism

With the same architecture as describe above, [5] explodes a H.264 parallel decoder. These studies explode data parallelism at the macroblock level. Macroblocks are the smallest organization units within an image. Inter coded macroblocks are always independent from others. For this reason they expose a tremendous amount of data parallelism. Dependencies between Intra-coded are addressed by partitioning a video frame into rows of macroblocks and assigning one full row of them to each decoding core. At first, experiments use four synergistic processing elements with a frame rate of 25.23 fps. A substantial achievement was reached using six processing elements, with results from ranging from 15.43 at 16Mbps to 34.94 fps with 2.5Mbps.

On 2010, [11] exploded data decomposition at frame and macroblock level. This study takes into account the bottleneck on the entropy decoding stage, CABAC (Context Adaptive Binary Arithmetic Coding). Frame-level parallelism techniques are used at entropy stage using multiple threads. The remaining decoding stages exploit macroblock-level parallelism. A macroblock software cache and a prefetching technique for the cache are used to facilitate macroblock pipelining.

In addition, an asynchronous macroblock buffering technique is used to eliminate the effect of load imbalance between pipeline stages. They classify the dependences into two classes: intra-frame and inter-frame dependence. The intra-frame dependence means that decoding a macroblock in a frame requires the result of

decoding some other macroblocks in the same frame. The inter-frame dependence refers to the case that decoding a macroblock in the current frame requires the result of decoding macroblocks in the previous frames.

This investigation evaluated its effectiveness by implementing a parallel H.264 decoder on an IBM Cell blade server. The evaluation results indicate that parallel H.264 decoder on a single Cell BE processor meets the real-time requirement of the full HD standard at level 4.0. When CABAC is used, this decoder meets the real-time requirement up to the bitrate of 12Mbps on a single Cell BE processor.

The x264, a free code library to encode video streams into the H.264 format, frequently uses the direct prediction mode to encode images, reason why some sequences present an increase number of dependencies than others. In the case of context-adaptive various lengths coding, CAVLC for short, the decoder achieves the real-time performance for all bit rates (i.e., 8Mbps, 12Mbps, 16Mbps, and 20Mbps). Moreover, it also meets the real-time requirement for the bitrate of 8Mbps when is run with CABAC on a PlayStation 3. Consequently, it satisfies the level 4.0 requirement and completely supports the full HD standard on a single Cell BE processor.

[2] does a parallelization at macroblock level and search range level. Search range consists in identify those macroblocks necessary to perform SAD operations. At macroblock level additionally partitioning subdivide macroblocks to exploit the large number of registers and texture referencing provided by NVIDIA CUDA architecture. This algorithm supports real-time processing and streaming for key applications such as e-learning, telemedicine and video-surveillance systems.

Mapping and scheduling algorithms

N. Dutt *et. al* started a series of studies on mapping and scheduling back in 2008 [8]. This work propose a framework for memory-aware task mapping on CMPs (Chip Multiprocessor) that tightly couples task scheduling with data mapping onto SPMs. SPMs unlike caches, do not have the hardware to support data transfers with the different levels of memory, compilers are then used to manage memory transfers using DMA (Direct Memory Access) engine. This feature is exploit for the current research. This framework permits designers to explore different memory-aware task maps and schedules procedures, for a given CMP platform, or a series of platforms.

This framework consists of three main components, the first is the simulated annealing (SA) exploration engine, which allows exploring different schedules, data placements for a given platform or a series of platforms, as well as to how many tasks to attempt to execute in parallel as this might impact the amount of thrashing/idle cycles processors will have. The second component is the memory-aware scheduler which tries to find task splitting opportunities by examining the tasks' loop nest, and schedules tasks with the goal of minimizing DMA transfers and improving the application's throughput. The last component is the modeling and evaluation of the final N schedules and data placements.

Experiments are based on the JPEG2000 image encoder. One of the most important features of this framework is that most of the data is fetched and stored in local SPMs. The base case approach executes N task sets (tiles) in parallel, so at any given moment there are a total of N tiles being processed by the N-CPU CMP. Performance is measured in terms of end-to-end execution cycles it takes to process an image on the bus level cycle accurate SystemC models. Since the objective is trying to minimize off-chip and DMA memory transfers, the main concern is dynamic power. CACTI v5.2 toolset is used to obtained power estimates. Experiments on JPEG2000 show an achievement up to 35% performance improvement and up to 66% power reduction over traditional scheduling/data allocation approaches.

Later on 2009, [7] propose a methodology to discover and facilitate parallelism opportunities via code transformations, distribution of computational loads and a decrease number of memory data transfers. Task partitioning, scheduling and data placement are the three principal processes on the application distribution for parallel systems. This research takes into account the mapping and scheduling as one single approach rather than two separate works.

Data placement and scheduling have a tightly relation between themselves because each one affect the other. Therefore, when mapping an application on to a multiprocessor platform, designers must look at both scheduling and data mapping as one tightly coupled step. This methodology is capable of discovering and enabling parallelism opportunities by partitioning tasks via code transformations, efficiently distributing the computational load across resources, and minimizing unnecessary data transfers. Code transformation refers to inter-kernel data reuse. Inter-data dependencies refer to dependency outside tasks. Typical task scheduling approaches, tasks are mapped without considering inter-kernel reuse.

Experiment setup consists of two base cases using JPEG2000 as the application target. The first is the classic CMP with only SPMs of the same size. The second uses classical CMP with local caches of the same size. Results show an average of 31% performance improvements and 35% in power reduction when compared to the base case. In some cases 80% performance improvements and 60% in power reduction where saw.

Keeping track of the same case study [6] add to the same methodology an increase code transformation taking both inter and intra data reuse as well as computational parallelism and exploiting the notion of priority edges in order to minimize unnecessary data transfer between on-chip and off-chip memories without sacrificing performance.

Because no other existing piece of literature (at current time) exists that has tried to exploit both inter and intra reuse opportunities as well as kernel level pipelining, they compare results with the closest work piece of work which is [7] that is using the JPEG2000 on the same scenario.

Results show that this technique can achieve up to 33% memory access reductions when compared to the state of the art, as well as up to 15% performance

improvement, effectively, reducing dynamic power due to off-chip memory accesses without sacrificing performance.

Not all configurations present a linear improvement of performance, the best case was for the 16 CPU configurations with 8KB memory size, for this case an achieve of 15% was obtained, justifying the reason to perform intra and inter data reuse whenever possible. Another important contribution of this work is the number of off-chip memory accesses present when using only inter-kernel vs intra-inter-kernel data reuse. For the configuration of 32 CPU with a memory size of 8K an achievement of 33% of the intra-inter over only inter was obtained.

Compilation

[21] presents a new compiler technique called multi-threaded memory methodology to discover data access patterns on multicore applications and determines the most effective partitioning of data. This partitioning is done at compiler time and it is enforced in runtime system to determine data allocation. The aim of this work is to choose an optimum data placement by discovering and representing multi-threaded memory access patterns (MMAP) at compilation time.

The author of this study highlights the predictable patterns of the stack and text sections and focuses on memory allocations and the usage of dynamically allocated memory, principally the memory allocator *malloc*. This allocator is one of the most representatives' data allocators, it is based on dynamic data management. This study does its experiments using cache memories. In addition, this methodology closely combines the analyzed results with cache performance through interaction between the compiler and run-time system.

1.5 Thesis outline

The organization of this work is described next. First, chapter 1 introduces the problem statement, the objective and the related work highlighting data parallelism. Then, chapter 2 does an introduction to the multimedia standards and in more detail to the H.263 encoder standard. It also presents a fast review of the standards that precede this design. Chapter 3 presents an explanation of the methodology proposed. It presents the principal parts on the methodology applied to a multimedia standard, the level parallelism and the metrics use to evaluate the efficiency of the framework. Next, chapter 4 addresses the methodology applied to the H.263. It explains the details of the SAD function. In addition, it describes the three different approaches to perform the distribution of tasks. To summarize, chapter 5 first presents the results of the profiling stage, then it explores the three parallelization approaches and the relation between task size and memory transfers. Finally, Chapter 6 presents the conclusions of the thesis and discusses future work.

Chapter 2

Background

2.1 Multimedia processing

Multimedia is the interaction of text, audio, images, animation and video. One of the most important features of multimedia is to process its data on a codified mode. The request for quality demands a huge volume of data and an important computing problem concerning the creation, processing and management of multimedia contents. Algorithms to process multimedia base its functionality on different tools. Motion estimation and compensation are two important phases on video compression for the MPEGs codec. These two different techniques have the common characteristic to identify similarities between sections over images on video sequences. The SAD is one of the preferred metrics used to evaluate the motion estimation and compensation.

The sequence of MB (Macroblock) encoding is from left to right in a row, and when all the MBs in the row are finished, the row bellow will be set as the next encoding target until no more rows exist in the frame, which indicate the end of the frame encoding. Obviously, the encoder must meet the requirement that every macro-block could not be encoded until its top, top-left, top-right and left neighbors (if exists) have all been encoded.

Image and video processing standards principal objective is to lower the amount of space required for storing and transferring on the internet. Without processing, both image and video occupy a large amount of memory. A colored image multiplies this memory space in order to store the three color channels (red, green and blue) that represent images. Coding and compressing these images are so important for storage and transmission purposes. For these reasons it is important to develop, use and improve image and video coding standards enabling image sharing and enhancing.

2.1.1 Image processing

One of the most important image compression algorithms is the JPEG, developed by the Joint Photographic Experts Group. The main feature of this algorithm it to get rid of useless information. JPEG takes advantage of the human eye range of visibility, in this form; data that is loss don't contribute with a substantial visual degradation for the human eye. JPEG2000, a newer version, has recently become an official standard. It bases its functionality on wavelet transforms.

2.1.2 Audio and Video processing

The most important associations for video compression standard are the Moving Pictures Expert Group (MPEG) and International Telecommunication Union (ITU) they have developed:

1. H.261 was designed by the ITU. It is the first member of the H.26x family of video coding standards in the domain of the ITU Video Coding Experts Group (VCEG), and was the first video codec that was useful in practical terms. The standard supported two video frame sizes: CIF and QCIF using a 4:2:0 sampling scheme. The next algorithm has been based closely on the H.261 design.
2. MPEG-1 is a standard for storage and retrieval of moving pictures and audio. The specific target of MPEG-1 is CD-ROM and DAT platforms for multimedia applications with a bandwidth of 1.5 Mbit/sec. The binary stream of encoding elements takes almost a 1.15 Mbit/s. The quality of MPEG-1 coded elements is similar to that of a VHS video.
3. H.262/MPEG-2 offers higher quality than MPEG-1, at a higher bandwidth it can reach to up to 10 Mbit/s. The scheme is very similar to MPEG-1.
4. H.263 is a standard originally designed for low-bitrate compression formats for videoconferencing. It was developed by the ITU Video Coding Experts Group. H.263 has many applications on the internet on sites such as YouTube, Google Video, MySpace, etc.
5. H.264/MPEG-4 bases its functionalities on the Fourier transforms or similar mathematical operations. Applications are video processing for the web, telephone and videophone calls and broadcast television.

2.2 Standard H.263

This thesis work has selected the H.263 coder as the optimization target. The H.263 coder inherits plenty of functionality characteristics that the modern (video) encoders use. This algorithm is the base for the widely used MPEG-4/H.264 codec, release by the Moving Picture Experts Group (MPEG) group formed by the International Organization for Standardization and the International Electro Technical

Commission. This standard uses Motion estimation and compensation as its engine to execute codification. These techniques require tremendous computer work, but it also represents a great opportunity to investigate the application of multiprocessing techniques to these algorithms.

Video encoders are considered to benefit greatly when more computational power can be used. More frames per second can be processed and better quality can be achieved when larger bit rates become available. The main reason to choose the H.263 codec is the great parallelization opportunities inherent to its functions. The H.263 is a member of the H.26x family of video coding standards designed by ITU (International Telecom Union). This codec had 8 extensions (or annexes): Inverse Transform Accuracy Specification, Hypothetical Reference Decoder, Syntax-based Arithmetic Coding, Considerations for Multipoint, Unrestricted Motion Vectors, Advance Prediction Mode, Frame Predictions, and Error Corrections.

Over the years, the algorithm has been improved by adding more extensions (or annexes) to it. In the year 2000, extensions known as Enhanced Reference Picture Selection Mode and Data-partitioned Slice Mode were added to improve frame selection. And from 2001 to 2005 some more modifications were made and the final version was published.

The H.263 standard is used on a variety of application mainly on internet video applications such as desktop video conferencing, video telephony, surveillance, and monitoring. It is also used as the video codec engine for sites like YouTube, MySpace, and Google Video. It also supports 3GPP (3rd Generation Partnership Project) file creation for mobile phones. An H.263 video can be played on LGPL-licensed (Lesser General Public License) software as VLC Media and MPlayer multimedia players, Real Video, Windows Media Player, and QuickTime.

All versions of H.263 supports five resolutions: CIF (Common Interchange Format), QCIF (Quarter Common Interchange Format), SQCIF, 4CIF, and 16CIF. 4CIF is 4 times the resolution of CIF, and 16CIF is 16 times the resolution. SQCIF is about half the resolution of QCIF.

Image type	Resolution	Luminance component	Chroma component	Total *
SQCIF	128x96	12288	3072	18432
QCIF	176x144	25344	6336	38016
CIF	352x288	101376	25344	152064
4CIF	704x576	405504	101376	608256
16CIF	1408x1152	1622016	405504	2433024

*Total = Luminance component + 2*Chroma component. Cr and Cb

Table 2-1: Image sizes and total pixels

The luminance component of the picture is sampled at these frame resolutions, while the chrominance components, Cb and Cr, are down sampled by two in both the

horizontal and vertical directions. That means that for each pixel is one luma component. Each block represents 8 x 8 components and each macroblock has four luma blocks, one blue chrominance block and one red chrominance block. Every macroblock represent 16 x 16 pixels, and because there is one luma component for each pixels each macroblock has 16 x 16 luma components 8 x 8 blue chrominance components and 8 x 8 red chrominance components as shown in Figure 2-1. Parallelization is done around luma components

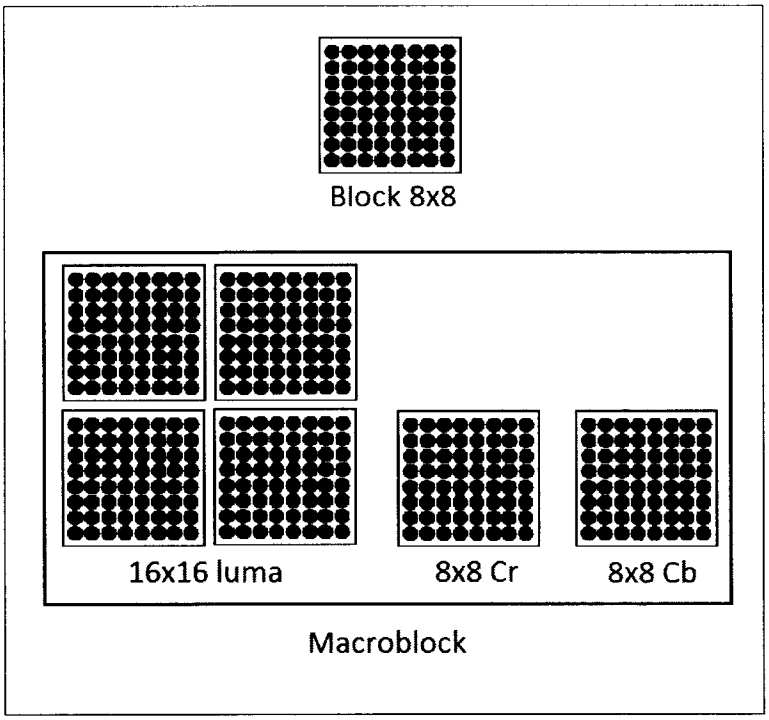


Figure 2-1: YCbCr Color space representation for 4:2:2 Sampling

2.2.1 Encoder

The objective of video coding is to provide a compact representation of the information in the video frames by removing spatial redundancies that exist within the frames, and also temporal redundancies that exist between successive frames. The H.263 standard uses the DCT (Discrete Cosine Transform) to eliminate spatial redundancies, and motion estimation and compensation to eliminate temporal redundancies. This algorithm uses two different encoding types: intra and inter coding modes. The intra coding mode uses the DCT to encode images. The result image of a DCT codification receives the name of I-picture. The inter-coding mode bases its functionality on the motion estimation and compensation. The result image of an inter-coding mode is called P-picture. A block diagram for a typical encoder is given in Figure 2-2.

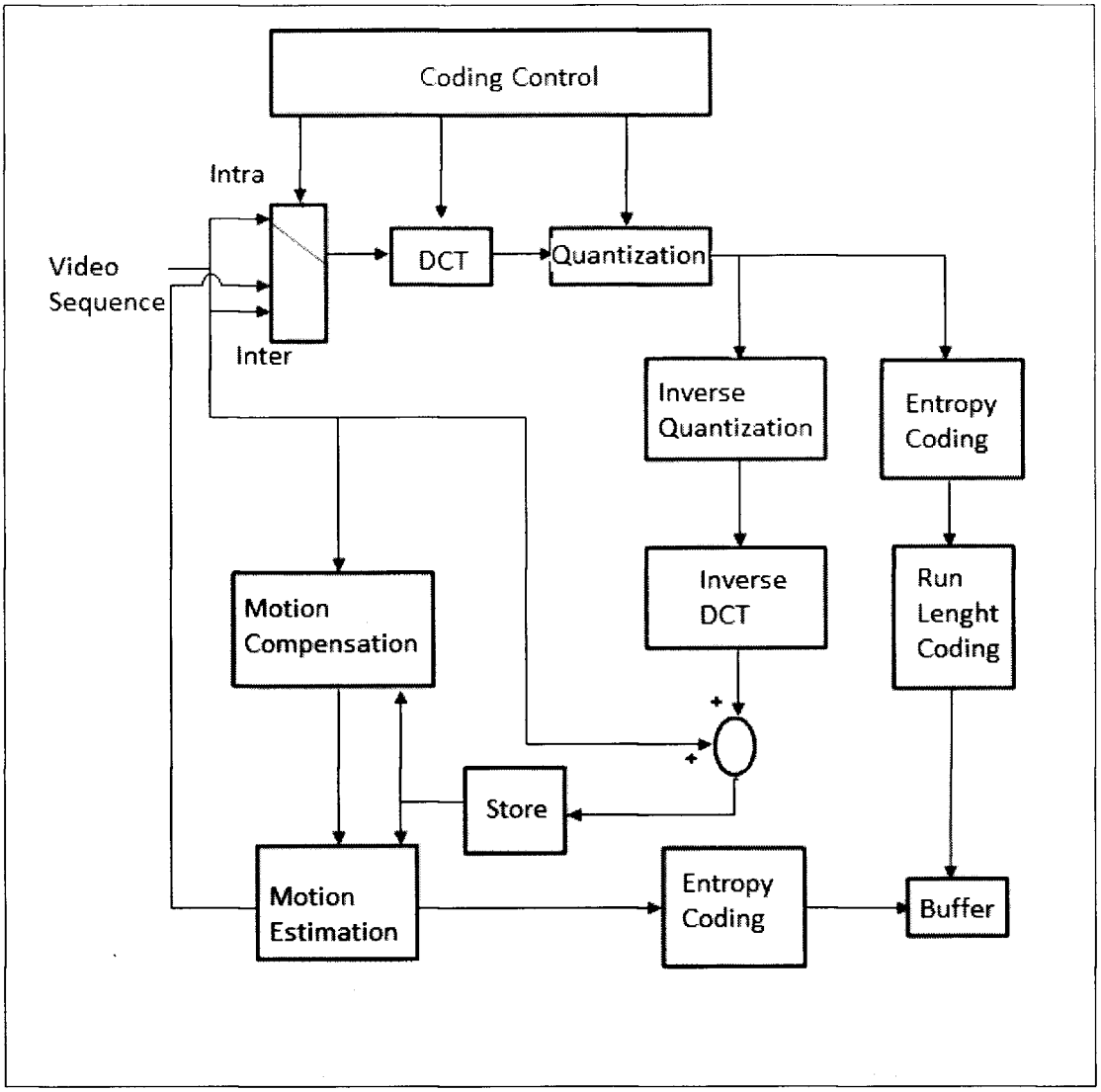


Figure 2-2: Block diagram of a H.263 video encoder

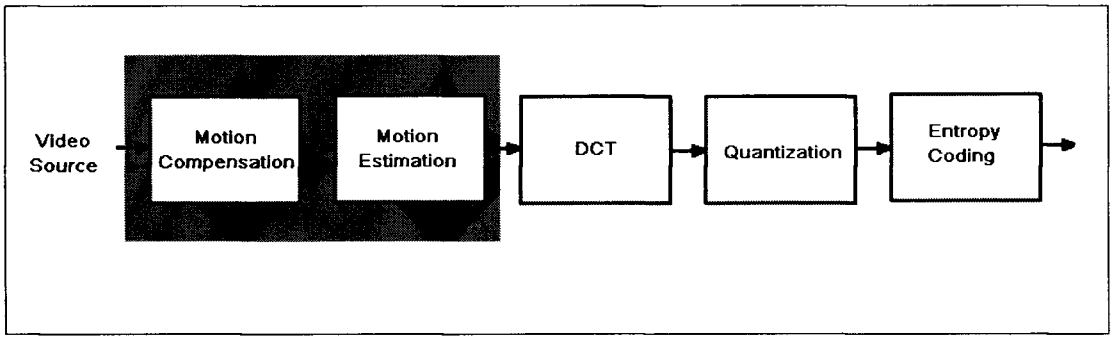


Figure 2-3: Parallelization focus on the H.263

2.2.2 Motion Estimation and Compensation

Similarities between frames are a characteristic that encoder algorithms use to enhance video processing. One simple approach to accomplish this objective is to consider the difference between the current frame and the previous reference and then encode the difference residual. A more efficient manner to process images is at macroblock level. A macroblock is an arrangement of 16 x 16 elements. Motion compensation is then a technique to predict the motion a macroblock will have over a video sequence. The aim of this technique is to find the best match between current and previous frame. Images are represented by the YCbCr color space, and motion compensation only use the luminance component. The bound of this searching is commonly called search factor or search window, and is the area where the encoder should look for a match.

A motion vector is used to represent the motion for every macroblock. Motion vectors have horizontal and vertical components to indicate the displacement between the previous and current blocks. The most popular metric to evaluate difference between macroclock differences is the SAD (Sum of Absolute Differences). The SAD is used to find the best matching between macroblocks. This metric calculate the sum of all differenced between every macroblock points. The SAD is an excellent tool to get good matching results but on the other hand it contributes with a great computational load.

2.2.3 Discrete Cosine Transform (DCT)

The DCT transforms a block of pixel values (or residual values) into a set of frequency coefficients. This is analogous to transforming a time domain signal into a frequency domain signal using a Fast Fourier Transform. The DCT operates on a two dimensional block of pixels. This transform efficiently represents block energy into a small number of coefficients. This means that only a few DCT coefficients are required to recreate a recognizable copy of the original block of pixels.

The DCT transform concentrates the energy of input samples into a small number of transform coefficients, which are easier to encode than the original samples. In addition to its relatively high decorrelation and energy compaction capabilities, the DCT is simple, efficient, and amenable to software and hardware implementations. A popular algorithm for implementing the DCT is that which consists of 8-point DCT transformation of the rows and columns, respectively.

2.2.4 Quantization

For a typical block of pixels, most of the coefficients produced by the DCT are close to zero. The quantizer module reduces the precision of each coefficient so that the near-zero coefficients are set to zero and only a few significant non-zero

coefficients are left. This is done in practice by dividing each coefficient by an integer scale factor and truncating the result. It is important to realize that the quantizer "throws away" information

The quantization is an important process on the encoding stage because it lower the data sent on the bit stream. Quantization is a compressing technique achieved to discard data. When the number of discrete symbols in a given stream is reduced, the stream becomes more compressible. The low sensitivity of human eye is taken in advanced by the quantizator to eliminate those components that humans can't perceive. Quick high frequency changes can often not be seen, and may be discarded. Slow linear changes in intensity or color are important to the eye. Therefore, the basic idea of the quantization is to eliminate as many of the nonzero DCT coefficients corresponding to high frequency components.

Quantization is applied to all DCT output elements in order to reduce useless information. The quantizers consist of equally spaced reconstruction levels with a dead zone centered at zero. In baseline H.263, quantization is performed using the same step size within a macroblock by working with a uniform quantization matrix, except the first coefficient of an intra-block that is coded using a step size of eight.

2.2.5 Entropy Coding

The entropy encoding in H.263 is used to compress the quantized DCT coefficients. An entropy encoder replaces frequently-occurring values with short binary codes and replaces infrequently-occurring values with longer binary codes. The result is a sequence of variable-length binary codes. These codes are combined with synchronization and control information (such as the motion "vectors" required to reconstruct the motion-compensated reference frame) to form the encoded H.263 bit stream.

Entropy is a useful tool to represent the estimated motion vectors and the quantized DCT coefficients. The entropy is performed by means of variable-length codes (VLCs). Motion vectors are first predicted by setting their components' values to median values of those of neighboring motion vectors already transmitted: the motion vectors of the macro blocks to the left, above, and above, right of the current macro block.

On the other hand, DCT coefficients are converted to a one dimension array by an ordered zigzag scanning operation. The result is then an array with a great number of nonzero components and some zero entries. To efficiently encode the whole array, each segment is assigned a code word, with the most frequent segments getting the code word with the least number of bits, and the least frequent segments getting the code word with the highest number of bits. The code word is generated based on three parameters (LAST, RUN, LEVEL). The symbol run is defined as the distance between two nonzero coefficients in the array (i.e., the number of zeros in a segment).

The symbol LEVEL is the nonzero value immediately following a sequence of zeros. The symbol LAST, when set to 1, is used to indicate the last segment in the array.

2.2.6 Coding Control

As mentioned earlier, coding phase can chose between intra or inter dependency encoding forms. The H.263 standard does not specify how to perform a coding control; it is leaved to the designer to choose a metric to evaluate estimation and compensation. The SAD is the criteria most commonly used. If a macro block has been detected to be closely similar to a previous block, then the encoder may skip encoding it. The encoder will send a message to the decoder to simply repeat the previous macroblock encoded.

2.2.7 Frame storage

The current frame must be stored so that it can be used as a reference when the next frame is encoded. Instead of simply copying the current frame into a store, the quantized coefficients are re-scaled, inverse transformed using an Inverse Discrete Cosine Transform and added to the motion-compensated reference block to create a reconstructed frame that is placed in a store (the frame store). This ensures that the contents of the frame store in the encoder are identical to the contents of the frame store in the decoder. When the next frame is encoded, the motion estimator uses the contents of this frame store to determine the best matching area for motion compensation.

2.3 MultiMake

MultiMake, work developed by [6], is a mapping design flow for minimizing unnecessary data transfers on multiprocessing systems. This application is capable of discovering and enabling parallelism opportunities via code transformations and distributing computational load across processors. Multimedia applications are suitable for MultiMake execution. Multimedia applications perform a series of operations over and over demanding great amount of repeatedly processing job and memory transfers. MultiMake decomposes the target application into smaller units called kernels. These kernels are then distributed and pipelined across different processing resources incrementing efficiency on memory usage. Experimental results on JPEG and JPEG2000 show up to 97% off-chip memory access reduction, and up to

80% execution time reduction over standard mapping and task-level pipelining approaches.

MultiMake is divided into three parts: Kernel creation, kernel analysis and pipelining. First step analyzes and decompose the target code (JPEG, JPEG2000) into a set of kernels. Second step analyzes kernels and determines inter-kernel data reuse, early execution analysis and computational node opportunities. At the final step the augmented task graph is pipelined across processor resources, if a solution is not possible then go back to step two and reduce the level of loop transformations.

2.4 Parallel architectures

Multicore design has gained popularity over applications requiring higher performance not reachable with single microprocessors. The objective of using multiple processors is both to increase performance and to improve availability. Running parallel processing has important challenges. Applications running on multiprocessor must have the property of running on independent tasks. For this reason not all applications are suitable from multiprocessing. Another major challenge is the large latency of remote access memory; this can cause an important delay effect.

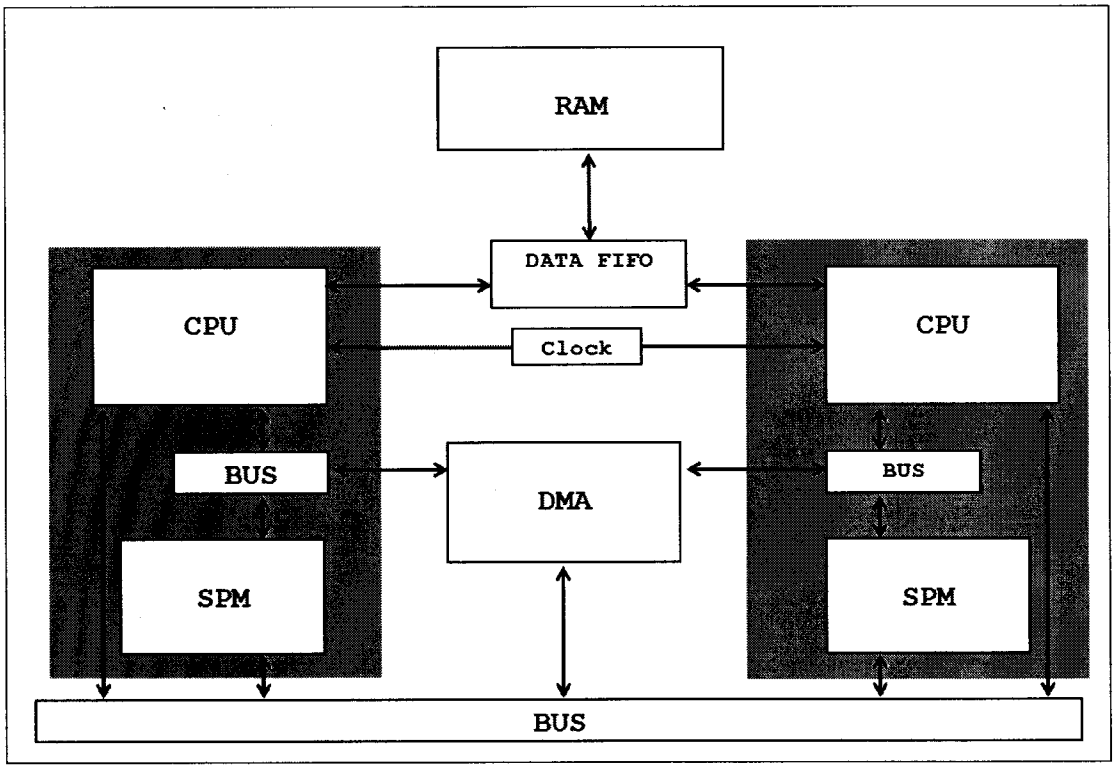


Figure 2-4: Multicore diagram for two processors

Multilevel caches can substantially reduce the memory bandwidth demands. [24] presents a work in which designers started to implement small scale multiprocessors using single physical memory shared by a bus. This could be accomplished principally for the large caches and the symmetric organization of memory. IBM introduced the first on-chip multiprocessor for general purpose computing on 2000. It is important to notice that multiprocessing has its major impact on servers and data-intensive applications rather than on a desktop pc.

2.5 Memory organization

According to the distribution of main memory, multiprocessors can have two different types of memory organization. Depending on the number of processors these types can be centralized and distributed shared-memory architectures [24]. The term shared memory means the address space is shared.

The principal characteristic of these two types of memory arrangement is its main memory. Centralized shared-memory has a single physical memory while the distributed configuration has the physical memory shared among processors. The criteria to select between these memory arrangements is based in the number of processors in the system, for a low number of processors a single memory still satisfies memory demands but not the case for an increase number of processors.

Chapter 3

Proposed strategy

3.1 Task representation

Our parallelized system must executed the H.263 encoder represented by several sections called tasks. These sections are first loaded to the scratch pad memories. A task is the fragment of job that a CPU must perform at a given time. The aim of creating kernels or task is to divide code into more simple blocks with a fix and known memory addresses and a minimum of operations to compute. There are three important aspects found during code analysis: type of memory accesses, total instructions and identification of *for* and *while* loops.

The amount of job and the waiting time of each task are the result of application segmentation and dependencies between tasks. Data information on the entire source code is then use to develop a parallelized model of the target H.263 encoder.

3.2 Parallelization steps

This study has five stages as shown on Figure 3-1: profiling, code extraction, task definition, memory transfers definition, CPU assignment and parallelization (three phases).

3.2.1 Profiling

This first step on our analysis aims to identify those sections of code that contribute with the greatest loads for the system. Our study takes into account that the multimedia applications are mainly written on C or C++ languages suitable for use with the analysis tools selected.

At the first approach, the code is analyzed in order to identify the sections that contribute with the greatest computing load. The input to this stage is the application code, and the outputs are the section codes requiring the greatest amount of work. The code is analyzed using the assembler instruction RDTSC (Read Time Stamp Counter) which gives the actual number of executed cycles from the start of the program. With this instruction the execution time for each function can be determined. The instruction gives the actual time at the start and end of every function. When the execution time of every function is known, all sub functions are analyzed in the same manner. During this analysis, all functions that contribute with considerable low computation are left aside. The final outcome of these steps is the code sections with higher computation demand.

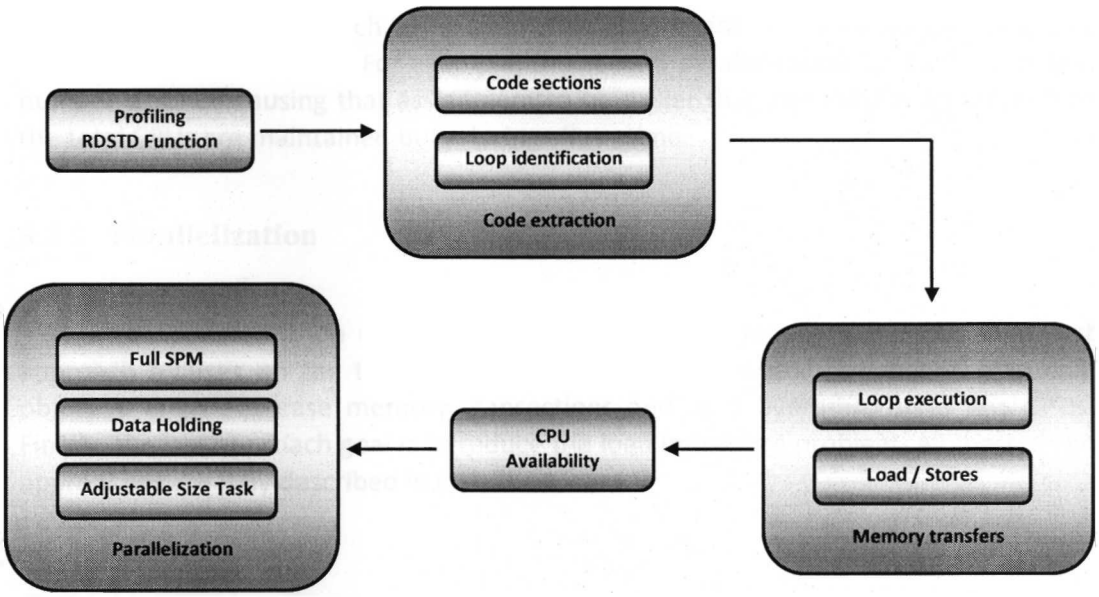


Figure 3-1: Methodology flow of experimental setup

3.2.2 Code extraction

The next step is the analysis to identify the code sections. At this stage the objective is to know the memory behavior and total instructions per function. A count

of load and stores instructions is made along with the static analysis of all instructions. All these data will be used later to do the task definitions. Task must reflect the behavior of the represented code and the exact number of cycles it required to compute.

3.2.3 Memory transfers definition

Each task needs the data coming from main memory or as a result from another task. These transfers must be defined at this step. The parameters of each memory transfer are the task requirement, the SPM size, the origin and the memory destination. The range and type of memory transfers are: RAM to SPM, SPM to SPM or SPM to RAM. Each SPMs of every CPU must have the necessary data to compute the task assigned.

3.2.4 CPU assignment

CPU assignment is in the order they become available. The number of CPUs used will depend on the approach being used. This assignment also depends on the size of each scratch pad memory. For example, the second parallelization approach uses less number of CPUs causing that assignment to be easier than the third approach where the total CPUs are maintained busy during all runtime.

3.2.5 Parallelization

Three parallelization approaches are used to decompose the H.263. The first approach focuses on the total use of scratch pad memories. The second approach objective is to decrease memory transactions and as a byproduct uses less CPUs. Finally, the last approach goal is to reduce the load imbalance problem. All these three approaches are fully described in Chapter 4.

3.3 Level parallelism

Research efforts have proofed two important techniques to distribute applications on parallel systems using two criteria: data and task decomposition. Data parallelism separates and performs operations over data. In contrast, task parallelism separates applications into different tasks that are computed on separated processors.

3.3.1 Data partitioning

Data partitioning is an accurate technique for those programs characterized by an extensive use of loops. These loops contain a small number of lines compared with the entire program; but these loops are executed several times until a certain condition is met. For this reason, data is distributed across the different resources to be processed in parallel, and each processor must perform all the necessary functions over these specific groups of data.

Multimedia applications are examples of programs carrying large amounts of loop workloads. Other important characteristic of multimedia application is that data is characterized into different levels depending on their size. A GOP (Group Of Picture) is the biggest division of video streams; it is composed by a series of frames and each GOP is independent from the rest. The rules to limit the bounds of a GOP mainly depend on the frame estimation and compensation. The other groups of data are the macro blocks; these are the smallest sections of an image. A H.263 macroblock is build from 16 x 16 pixels. Each macro block is independent from the others and a great number of operations base their functionality on them.

3.3.2 Functional partitioning

The principal characteristic is that each processor performs the same function but it exchanges data between main memory and the rest of processors. This contrasts with data parallelism, where each processor computes all necessary functions over a group of data. In a task parallel system, this technique consists on giving a certain thread to the processors in order to compute the same function for all data. An important characteristic of task parallelism is that threads constantly communicate with each other in order to share data.

3.4 Metrics

The metrics to measure the performance of a parallel system are runtime, speedup, efficiency, cost and overhead. Below are given the five equations for these metrics and Table 3-1 gives the definition for this metrics.

$$\boldsymbol{Runtime = executed\ cycles} \qquad \text{Eq. 3-1}$$

$$\boldsymbol{Speedup} = \frac{\boldsymbol{Sequential\ Runtime}}{\boldsymbol{Parallel\ Runtime}} \qquad \text{Eq. 3-2}$$

$$Efficiency = \frac{Speedup}{Processors} \quad \text{Eq. 3-3}$$

$$Cost = (Runtime)(Processors) \quad \text{Eq. 3-4}$$

$$Overhead = Cost - Sequential\ Runtime \quad \text{Eq. 3-5}$$

Metric	Description
Serial runtime	Total cycles to execute program.
Speedup	It is used to measure how faster a parallel algorithm is compared to its corresponding sequential algorithm. The maximum speedup value in a multicore system is equal to the number of processors involve.
Efficiency	Determines the level in which the resources are better used. The range to measure efficiency is from cero to one being one the maximum and an ideal target. The ideal efficiency is reached when the entire application code is exactly divided in *p* (number of processors) task, and each task takes exactly *t* time, where *t* is sequential runtime/*p*.
Cost	The product of runtime and the number of processors involve.
Overhead	Part of the cost that is not incurred by the serial algorithm on a sequential computer. It is the total time collectively spends by all the processors in the system minus the sequential algorithm time.

Table 3-1: Metric definitions

3.4.1 Metric example

A program is run on three different systems with different number of processors. The first has one CPU, the second and the third have two and four processors each. The program needs 1000 cycles to execute on the single CPU system and 500 and 450 on the two and four CPU systems respectively. The next table and figure show a description of the metrics used to evaluate the three systems.

	Two CPU	Four CPU
Runtime	500	450
Speedup	1000/500=2	1000/450=2.222
Efficiency	2/2=1	2.222/4=0.555
Cost	(500)(2) = 1000	(450)(4) = 1800
Overhead	1000 – 1000 = 0	1800 – 1000 = 800

Table 3-2: Example of metric use

Ideal parallel systems should reduce runtime up to the same number of processors involved. Figure 3-2 shows the runtime of the metric example. The system with two processors has the ideal runtime of 1000/2, and the four processors system must have a runtime of 1000/4.

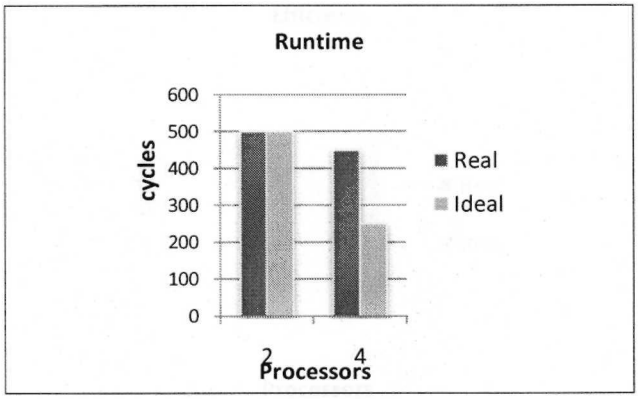


Figure 3-2: Runtime of metric example

Ideal parallel systems should reduce runtime up to the same number of processors involved. Figure 3-2 shows the runtime of the metric example. The system with two processors has the ideal runtime of 1000/2, and the four processors system must have a runtime of 1000/4.

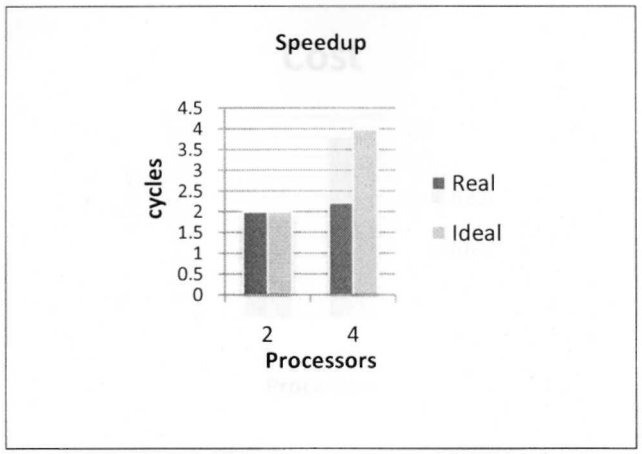


Figure 3-3: Speed up of metric example

Figure 3-3 shows the speedup of the metric example. The ideal system with two CPUs has a speedup of two. A multicore system reaches a parallelization limit when the speedup remains the same even when doubling the number of processor.

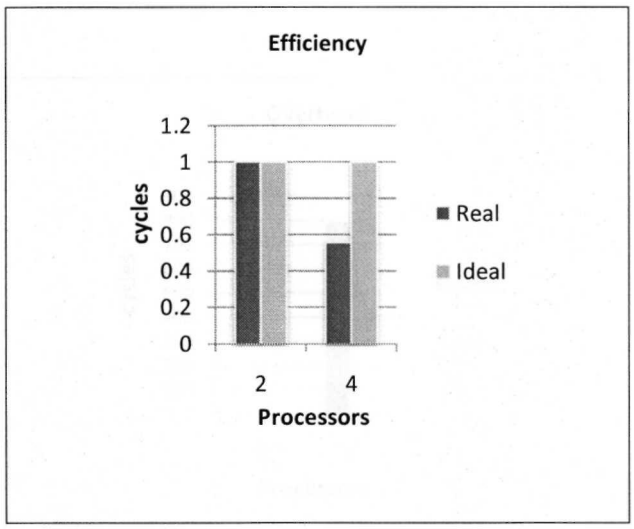


Figure 3-4: Efficiency of metric example

Efficiency is a metric to measure the effectiveness of the use of every resource on the system. An ideal efficiency should be one. Figure 3-4 presents the efficiency of the metric example. In the example, the four CPU system does not reduce runtime by 4 (1000/4). This indicates that an increment in the number of processors does not improve efficiency. As a result the effectiveness of the system is less than 1.

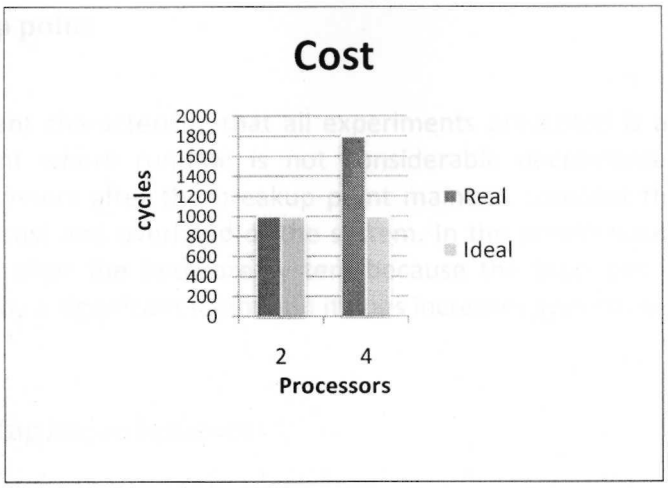


Figure 3-5: Cost of metric example

The cost of the systems is proportional to the time and the number of processors involve. Ideally the runtime should be lower as the number of CPUs increases. A base cost of a system is taken with the sequential algorithm. Figure 3-5 presents the result of the cost for the metric example.

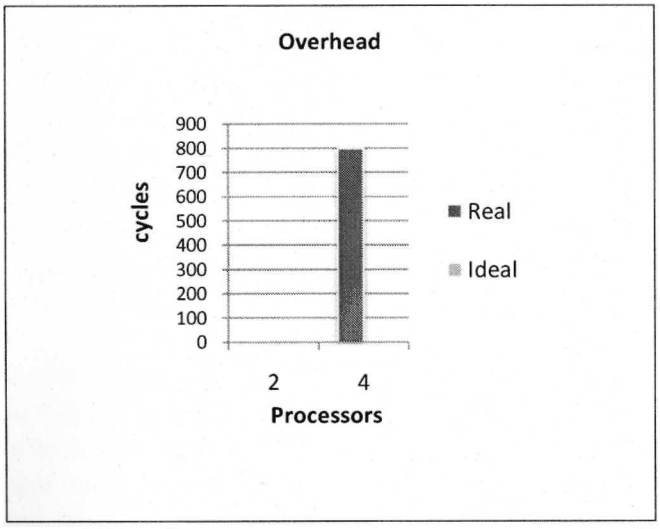


Figure 3-6: Overhead of metric example

Overhead enclose information to measure resource's time used efficiently. Overhead is caused by three factors: Inter processor communication, load imbalance and extra computation. In the example results shown in Figure 3-6, the four CPU do not make a good use of resources causing a big overhead.

3.4.2 Breakup point

An important characteristic that all experiments presented is a break up point. This is the point where runtime is not considerable decremented. Doubling the number of processors after the breakup point maintain constant the runtime but it increments the cost and overhead of the system. In the previous example a breakup point is located after the two-core system because the four-core system does not reduce runtime by a significant factor and it does increases system cost and overhead.

3.4.3 Measuring improvements

The method to measure the improvements between different systems is to compare the results between the N-core system with its corresponding one-core result. The value to be reported is the percentage represented by the difference between the sequential execution and the best case. For example, in previous example the best runtime is 450 and the runtime of the serial execution is 1000; the four-core system has a runtime improvement of 55% (550/1000).

Chapter 4

Parallelizing the H.263 encoder

4.1 Parallelization approaches

This parallelization research has two principal goals. The first objective is to analyze a multimedia application behavior based on motion estimation, and the second objective is to categorize parallelization work based on different parallelization lines. Our study targets three different parallelization approaches that rely on the SPM size, data reuse and CPU availability.

4.1.1 Full Memory - First parallelization approach

Greater SPM size means larger space to allocate data and avoid memory misses. Motion estimation depends on the operation of two sections from different images. In order to execute a task each SPM - CPU must have the necessary information, that is, a slice of two different images. A greater SPMs size means that its CPU should have more chances to increase its memory hits. But in addition, allowing a CPU to have greater slice of images means that it must perform a greater amount of work. This situation works well for some scenario but for other circumstances do not contribute with a substantial performance. Large groups of data can even increase the costs on the system. The cost on a system rises due to two principal factors: poor utilization of CPUs, and incorrect division of tasks

The first approach in our parallelization framework is to use all available SPM memories. One CPU performs one task at the time. The number of CPUs and tasks to process a video sequence is depended on the image and SPM sizes. As the image size increases, there is also a growth in the number of task performing computer work. As

the SPM size increases, it is possible to allocate a larger amount of data resulting in a decrement in the total number of tasks. This is presented because a CPU will have slices with greater size.

Figure 4-1 shows this distribution among three images. On the figure, the image and SPM size limit the number of processors to four for performing the SAD on every two images. The first four CPUs (0 – 3) perform the SAD between the *image 0* and *image 1*, and the second group of processors (4 – 7) do the SAD between *images 1* and *2*.

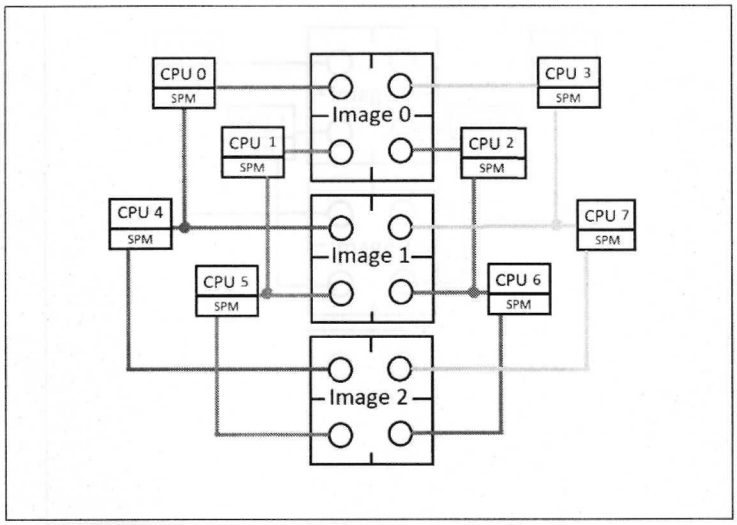


Figure 4-1: First parallelization approach for three images

4.1.2 Holding Data – Second parallelization approach

Previous sections provide an explanation about how to split tasks in order to fulfill all SPM space available; thus, CPU should process tasks of biggest size. In the second parallelization approach, a task must keep the same task size. Once a SPM of a CPU is loaded with a given image section, this CPU has to keep performing all the operations of this image section. No other CPU must perform operation over this image section. Therefore, a smaller number of CPUs will be required under the second parallelization approach. So in this new approach, a lower quantity of CPU is going to be required.

Figure 4-2 and 4-3 show the distribution of tasks for three images on two instants of time using only four processors. As on previous section, eight CPU are available, but only four are needed. On the first figure, the first four processors perform the SAD operation on the *images 0* and *1*. This operation is done similarly to the first parallelization approach. Instead of assigning the next two images to processors 4, 5, 6 and 7, this new approach waits until processors 0,1, 2 and 3 finish their tasks. Then, these same processors do the SAD for the next image as shown of Figure 4-3.

The principal difference of this second approach is that only four processors are required compared to the 8 CPUs required by the first approach. The objective of this is not to waste time on reloading again that same data on other processors. This approach is orientated for those SAD tasks that have a low search window. Tasks that perform the SAD operation with a low search window consume an amount of time comparable with the time required by the DMA transfers to load SPMs, for that reason, this new approach provides a solution for a certain size of task.

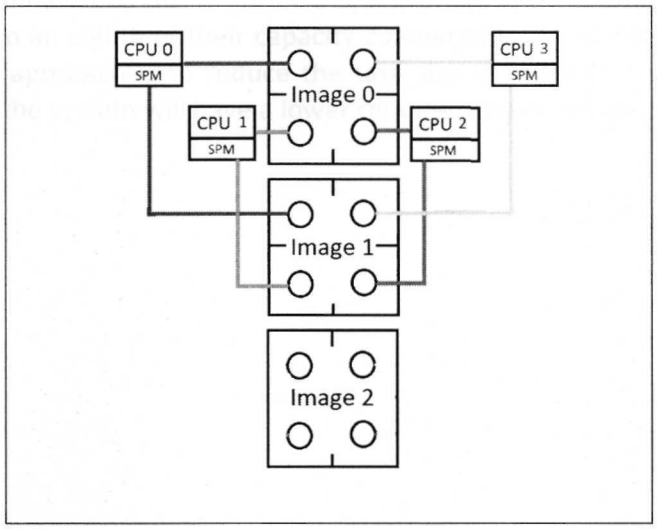


Figure 4-2: Second parallelization approach for three CPU, part 1

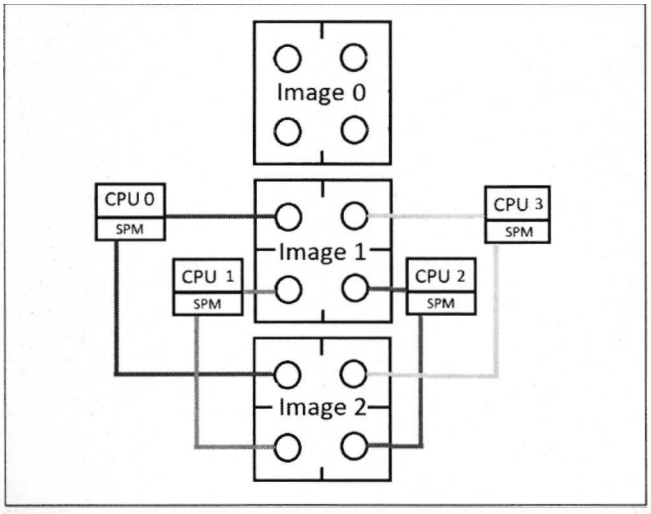


Figure 4-3: Second parallelization approach for three CPU, part 2

4.1.3 Adjustable Size Task – Third parallelization approach

This last approach presents a solution for the load imbalance task problem. Figure 4-4 presents the one-task per CPU behavior of parallel systems with one, two, four and eight processors. The system with a single CPU must perform a serialized computation of all tasks. The system with two CPUs must compute two tasks at the time with a 50% of reduction in the task size. The four-processor system must perform four tasks at the time using four different CPUs with their corresponding memories filled to a quarter of their capacity compared to the single CPU system. The system with 8 CPUs have the memories filled to an eighth of their capacity compared to the single CPU system. The objective of this approach is to reduce the task size in order to keep all processors busy. Therefore, the system will have a lower requirement in memory space.

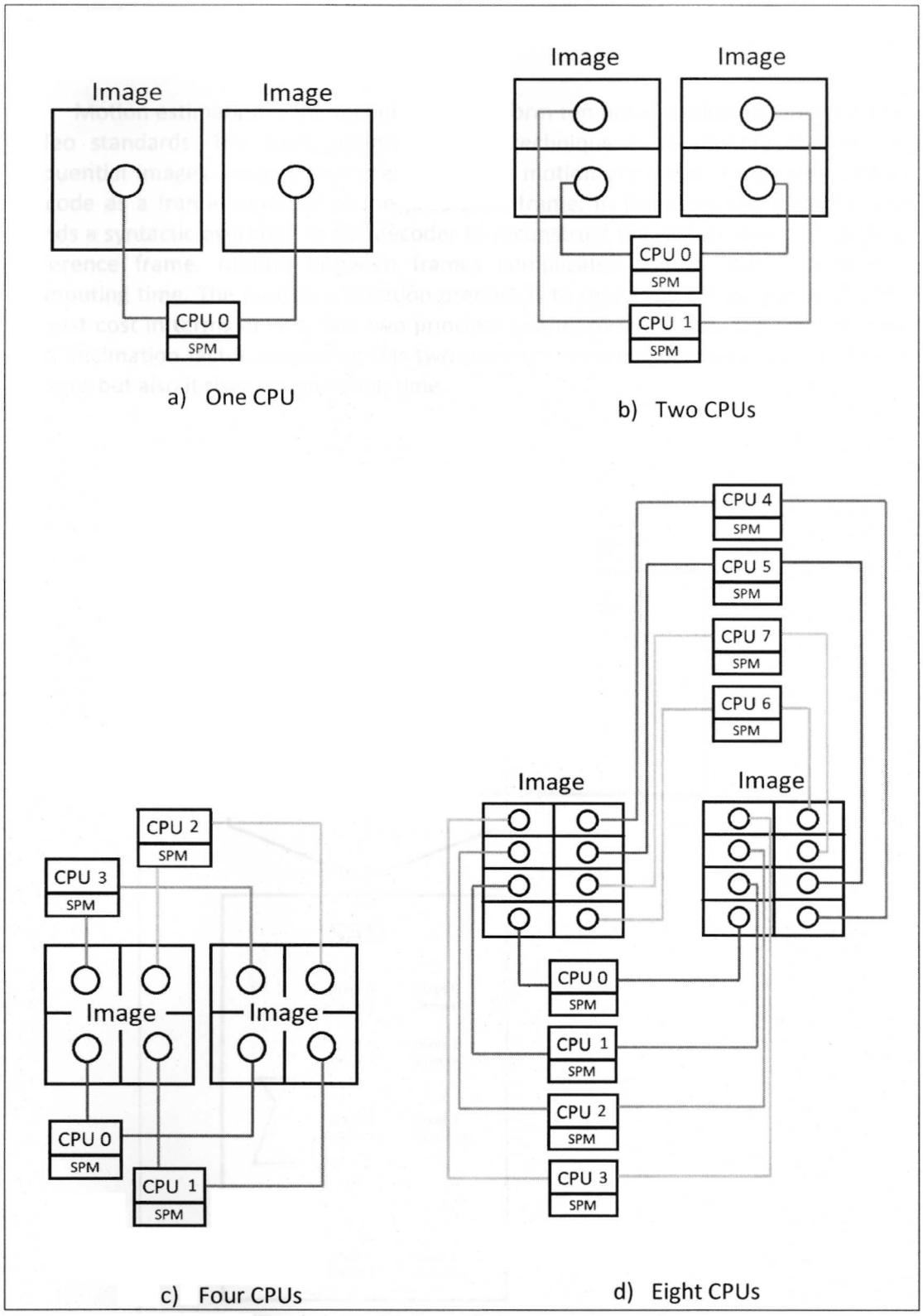


Figure 4-4: Third parallelization approach

4.2 Sum of Absolute Differences (SAD)

Motion estimation is the technique to perform temporal prediction among MPEG video standards. The basic premise of this technique is to identify changes on sequential images. Images that present a zero motion are easier to identify and to encode as a frame duplicate of the prediction frame. In this case, the encoder only sends a syntactic overhead to the decoder to reconstruct the picture from the original reference frame. Motion between frames complicates coding task incrementing computing time. The motion estimation premise is to represent the new image at the lowest cost in terms of size. The two principal factors to define size are search range and decimation factor, improving this two parameters increases directly the quality of images but also it rises computation time.

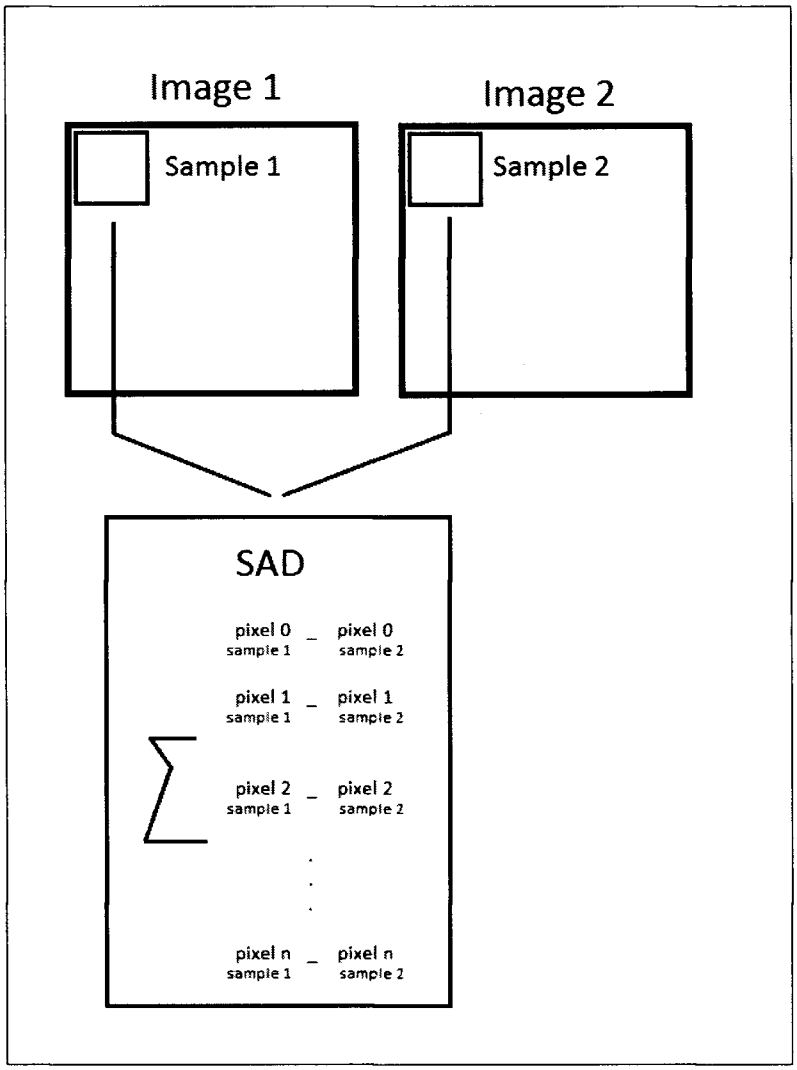
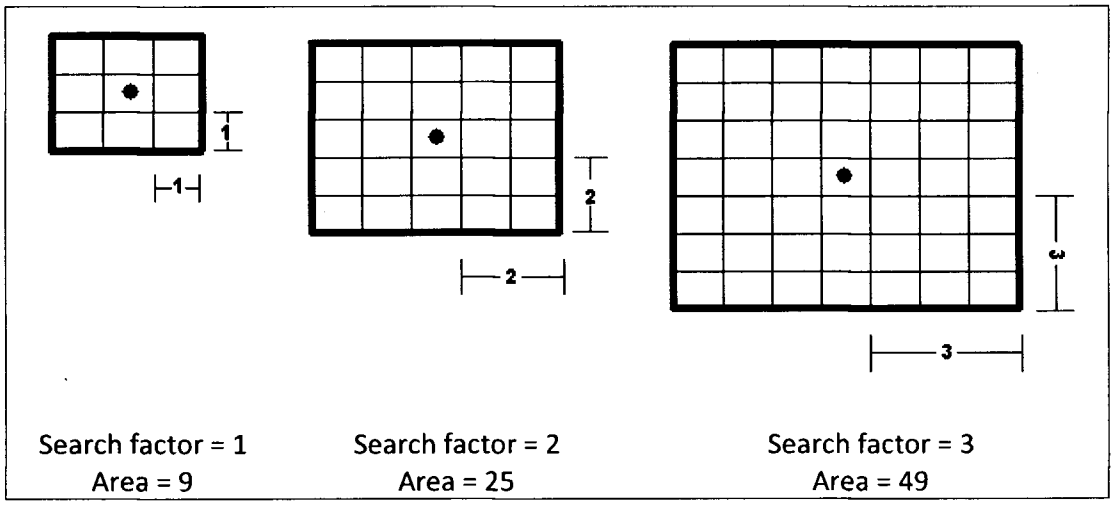


Figure 4-5: SAD Functionality description

The sum of absolute differences (SAD) is the most common metric to evaluate motion estimation. The SAD accumulates absolute differences between the corresponding pixels involve from the two images. The sum is then used as a metric to measure block similarity. If the result is lower than a given limit, the section is considered similar and the section is not encoded instead it only sends a message to the decoder as explain above. An illustration on the SAD function is show in Figure 4-5.

The search factor parameter defines the number of pixels needed in each SAD function. Larger samples improve image quality and also increase computing demand. This value is the number of pixels between the center and one of the edges on the sample target as shown on Figure 4-6. This figure presents three different search factors values. The workload depends directly on this value; each level of searching adds more pixels than previous levels. The most common values for these search factors are up to 15, as shown on Figure 4-7.

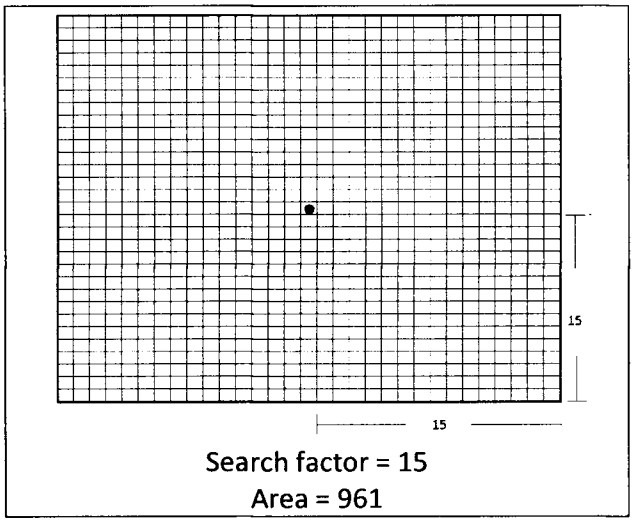


Search factor = 1
Area = 9

Search factor = 2
Area = 25

Search factor = 3
Area = 49

Figure 4-6: Search area on SAD



Search factor = 15
Area = 961

Figure 4-7: SAD with search area of 15

Pseudo code on Figure 4-8 shows the number of iterations of SAD code on a video sequence. The number of iteration depends directly from the number of images, the image size and the search factor. The search factor affects directly the SAD count. As the amount of search factor increases; the SAD count has a proportional increment. The search factor parameter specifies the number of pixels to compare against a central pixel. Figure 4-8 shows an illustration of the search area on SAD execution and table 4-1 specifies the counter for different search factors on the five different sizes of images.

```
FOR 0:N
FOR 0:lines/MB_SIZE
        FOR 0:columns/MB_SIZE
                FOR 1: factor
                        FOR a=0:a*area_search
                                DO SAD

                    N   Number of images
              MB_SIZE   Macroblock size (16)
               factor   Search Factor
                  SAD   SAD function
```

Figure 4-8: SAD Iterations

		Search factor							
		1	3	5	7	9	11	13	15
Format	sub-QCIF	3456	20736	51840	96768	155520	228096	314496	414720
	QCIF	7128	42768	106920	199584	320760	470448	648648	855360
	CIF	28512	171072	427680	798336	1283040	1881792	2594592	3421440
	4CIF	114048	684288	1710720	3193344	5132160	7527168	10378368	13685760
	16CIF	456192	2737152	6842880	12773376	20528640	30108672	41513472	54743040

Table 4-1: SAD iterations for a video sequence of 9 images

$$SAD\ iterations = N \frac{lines}{MB\_Size} \frac{columns}{MB\_Size} \sum_{i=1}^{sxy} \sum_{j=0}^{8*i} 1$$

Where:

N	Number of images in sequence
MB_Size	Macro block size (16)
sxy	Motion vector search window
lines	Format (columns x lines)
columns	SQCIF 128x96
	QCIF 176x144
	CIF 352x288
	4CIF 704x576
	6CIF 1408x1152

Figure 4-9: SAD iteration counter

Chapter 5

Results

5.1 Experimental Setup

The experiments are run in the Advanced Microcontroller Bus Architecture (AMBA) [1], an open source tool. This tool facilitates the design of multicore systems with great number of resources (processors, cache memory, SPM, data buses). SimpleScalar Tool Set [27], an open source tool used to obtain code information (memory type accesses, total instructions and assembly code from C progams). Another aid used to profile the H.263 coder is the RDTSC (Read Time Stamp Counter), instruction present on x86 Intel processors to return the 64-bit time clock counter. The operating system used is Fedora Core 13 Linux Kernel 2.6.34.8-68.fc13.i686.PAE [12]. Experiments use a sample base of 9 images sequence in 4-CIF format of size 5,474,304 bytes [SEE APPENDIX] and a *H.263 encoder version 1.7* [23]. The image sequence and the encoder code were obtained from Mediabench Consortium, an association from The University of California at Los Angeles.

5.2 Profiling

It is important to address that not all code needs to be parallelized. An important goal in multimedia applications is to increase quality and speed for use under internet. Coding standards for multimedia consume great amount of processing work to compute data. Source codes for multimedia standards are built from some thousand of

lines but not all the code provides the same computing load. There are specific sections that contribute with a great portion of computing time. Our experiments start with this demonstration. H.263 source code is analyzed to identify the heavy load sections that contribute with the highest computing job demand.

Using RDTSC over sample input shows that it needs 11,845,939,668 cycles to execute and there are also three principal parts on the source code: Coding, Prediction and First Image Codification. Figure 5-1 shows the performance of these three sections. From this figure, it is easily seen that Coding and Prediction sections contribute with up to 97% of computing time. These two functions are analyzed and then showed they are sharing one single sub-function consuming most of the time for each of them.

Table 5-1 shows the H.263 behavior. Each field in Table 5-1 has the number of executed instruction cycles and its corresponding percentage over the total program execution. The first column enumerates the four sections on code. Section one, first image, encodes the very first image, even though this job is done without estimation or prediction, it only contributes with a small fraction of time. The prediction section refers to the part in which it is compared the actual image with the very next and the previous images. This section contributes with the 43.3% of computing time. Coding section encodes data already in process representing the 54.4% of computing time. Finally, control instructions, parameters definition and so on are group in a section called Other.

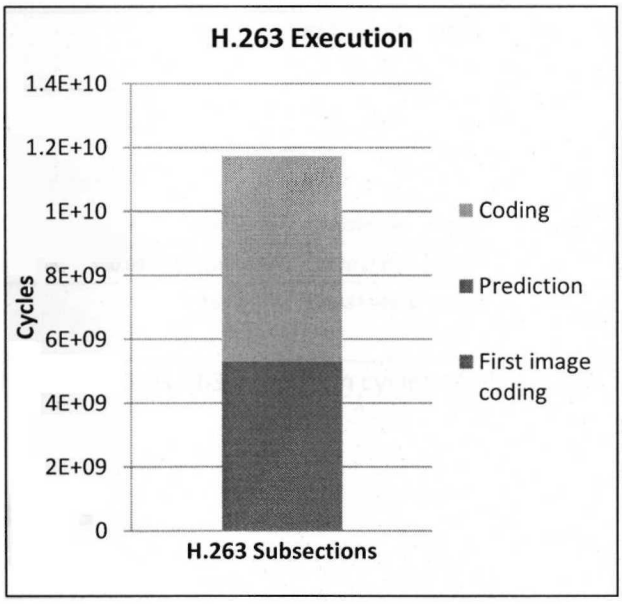


Figure 5-1: Iteration between all H.263 code with every part of its sections

The second and third columns on table 5-1 show the function hierarchy. Both Prediction and Coding sections share the lower level function SAD. The SAD function

consumes 55.6% off all program execution. Table 5-1 concludes that coding and prediction sections consume the greater amount of computing time from all programs with up to 86% of time. These high percentages are enough reasons to work on these two functions.

It is important to address that this two functions are written with 600 lines of code from a total of 7000 from the total code and that the SAD function is less than fifty lines of code. At this point, a parallelization job of seven thousand lines is reduced to only few lines. The reason that SAD consumes the greater amount of computation time is simple; this function invokes a great number of memory transfers. Appendix A.2 provides 'C' SAD code description. Appendix A.3 presents SAD assembly representation.

Section	Cycles executed	%	Part	Cycles executed	%	Sub-part	Cycles executed	%
First Image	160947816	1,36						
			Other	11130984	0,09	Other	1791813454	15,13
Prediction	5130597996	43,31	→ Estimation	5119467012	43,22	→ SAD	3327653558	28,09
			Total	5130597996	**43,31**	**Total**	**5119467012**	**43,22**
			Other	1407086016	11,88	Other	1757177646	14,83
Coding	6427593576	54,26	→ Estimation	5020507560	42,38	→ SAD	3263329914	27,55
			Total	6427593576	**54,26**	**Total**	**5020507560**	**42,38**
Other	126800280	1,07						
			Other	1705965096	14,40	Other	3548991100	29,96
Total	**11845939668**	**100,00**	→ Estimation	10139974572	85,60	→ SAD	6590983472	55,64
			Total	**11845939668**	**100,00**	**Total**	**10139974572**	**85,60**

Table 5-1: H.263 execution cycles of all its parts

5.3 Experiments

Experiments were done over a sub-QCIF format on 9 images. Simulations used two different computation loads, each varying the search factor on the SAD function. These two computation loads are: light and heavy with search factor of 1 and 15 respectively. The experimental section is broken in three approaches: maximizing use of SPM, data holding and reducing task size. On the first approach, there are three scenarios associated to different SPM sizes: 4KB, 8KB and 16KB. The second approach

differs only on reusing data still allocated on SPMs. And, the third approach is a reduction of task size in order to increment total task number for reducing processor idle time.

5.3.1 Full Memory - First parallelization approach

For this first part, the total number of tasks depends directly on the SPM and image size. Figure 5-2 presents the distribution of tasks for this stage, the number of tasks is the quotient of image size and SPM as expressed on eq. 5-1. This distribution uses all available SPM.

$$\textbf{\textit{Number of task}} = \frac{\textit{Image size}}{\textit{SPM size (KB)}} \qquad \text{Eq. 5-1}$$

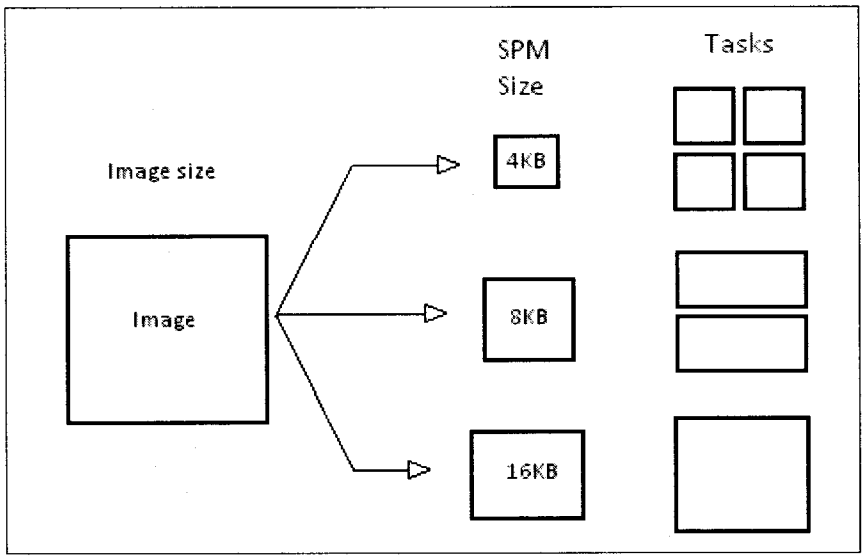


Figure 5-2: First parallelization approach

Figure 5-3 shows runtime of the three SPM sizes for the light and heavy level loads. Both graphics show a breakup point after two and four core systems. The best case for the light level load is the two-core system, it presents a 15% reduction of runtime with a SPM size of 8KB. For the heavy load, a good solution is to use four processors with SPM size of 16KB, it presents a 33% runtime reduction.

Figure 5-4 shows speedup for the two different loads. Ideal speedup is equal to the number of processors involved. For the light level load, the biggest speedup is achieved for the system with two processors, with a speedup of 1.46. It is important to address that systems with higher number of CPUs and/or SPM sizes still have the same

speedup but at the cost of not using all memory. This is a good indicator when estimating memory size for a system. Next graphs will show that doubling processor number or SPM size does not represent a substantial improvement on performance but it does increase cost and overhead.

The next two figures showed that systems with many processors and/or SPM sizes do not contribute with a substantial improvement. Figure 5-5 shows the cost for both loads, it exhibits the effects of doubling resources without using them correctly. The biggest cost is presented for the thirty two processors for the three SPM sizes (4KB, 8KB and 16KB).

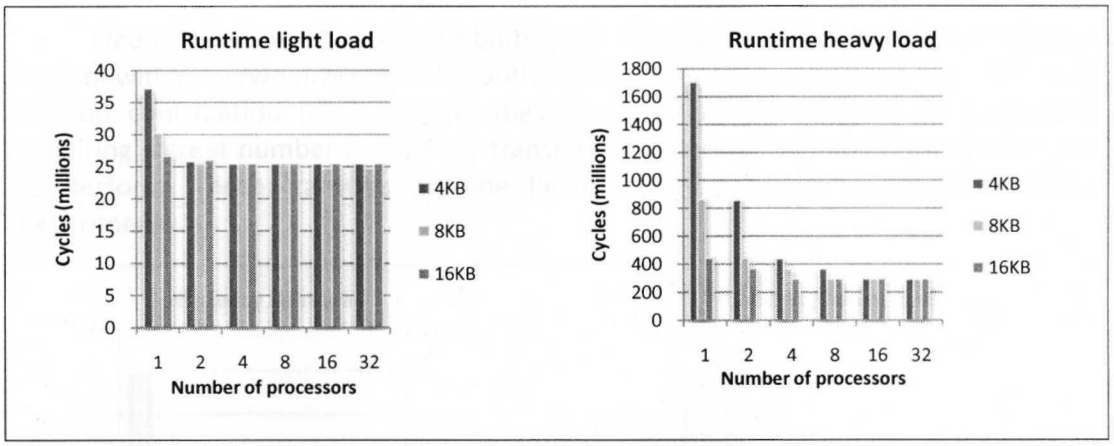


Figure 5-3: Runtime for light and high level load

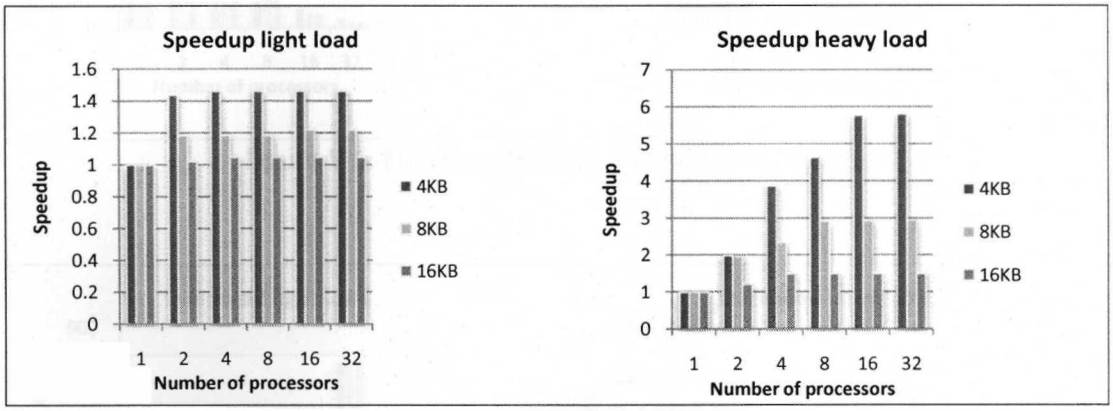


Figure 5-4: Speedup for light and heavy loads

From Figures 5-3, 5-4 and 5-5, it is learned that, at some point, there is no benefit in doubling neither CPU number nor SPMs size. The optimum number of CPUs and the optimum SPM size depend directly from the behavior of the computational load. The two principal parameters that define an application behavior are the memory transfers, the instructions needed to complete tasks and the amount of data loaded to the SPM memories.

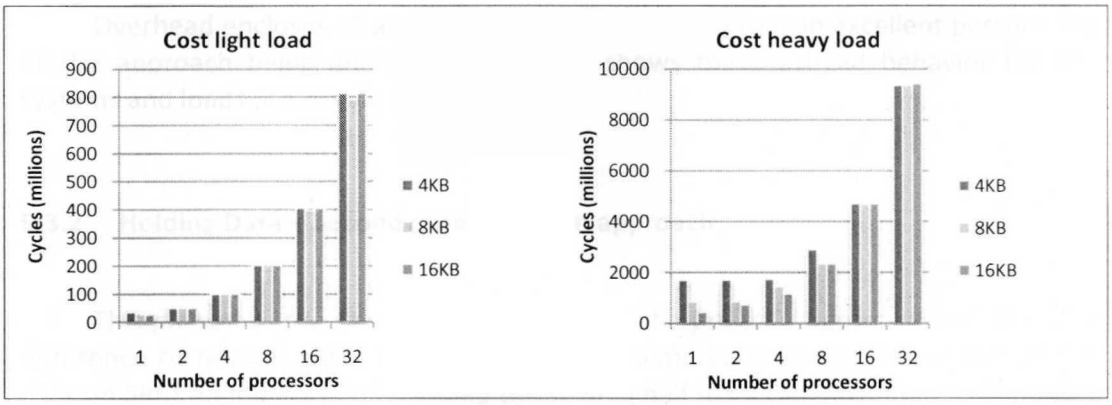


Figure 5-5: Cost for light and heavy level loads

Figure 5-6 show efficiency for both loads. The biggest efficiency is reached for a system with only two processors for both loads. For the two graphs, a bigger SPM size has no contribution for increasing the efficiency, this is due to SAD operations requiring a great number of memory transfers compare to the number of instructions to perform. These transfers are the factor which increases overhead on many experiments.

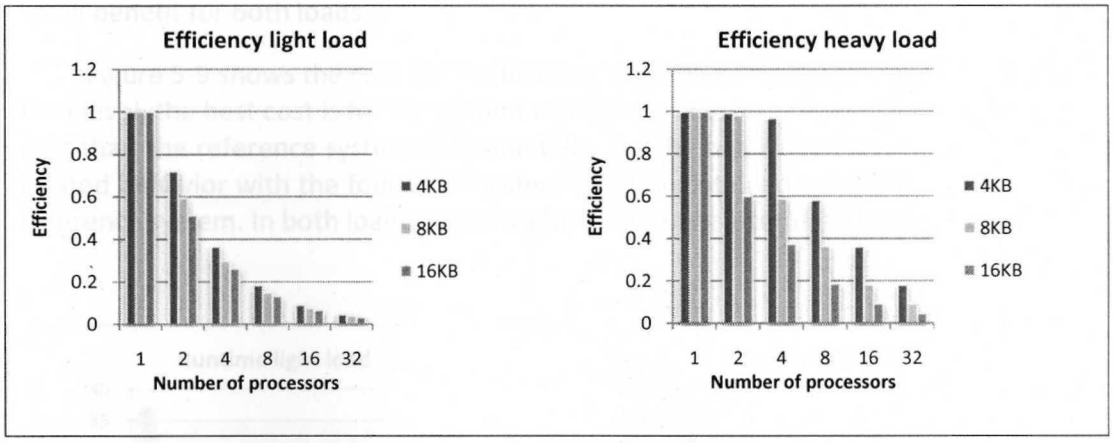


Figure 5-6: Efficiency for light and heavy level loads

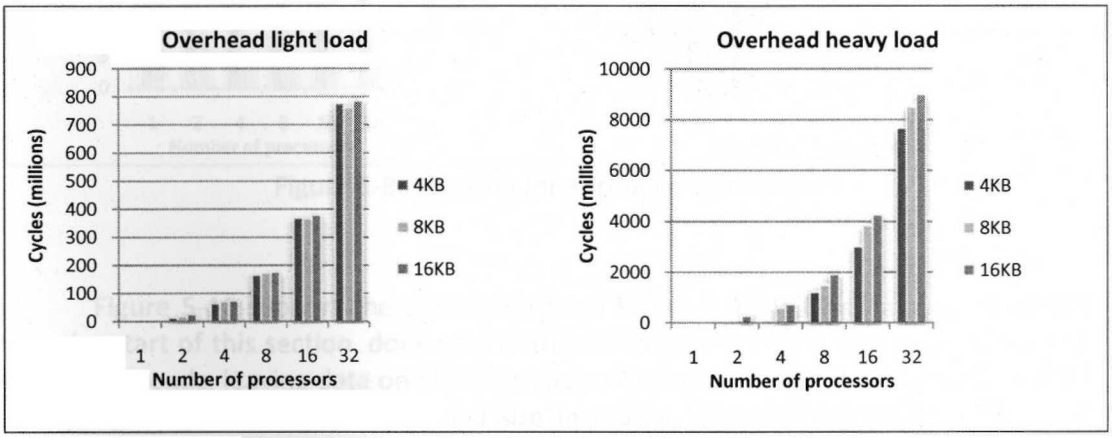


Figure 5-7: Overhead for light and heavy level loads

Overhead encloses all applications behavior and it gives an excellent perspective of the approach being analyzed. Figure 5-7 shows the overhead behavior for the systems and loads previously presented.

5.3.2 Holding Data – Second parallelization approach

This phase has the same number of task as the previous approach with the only difference of reusing data. This data reusing scheme consists on leaving half of the data on SPM memories and only load the second half from the next image. This cause a lower need of the number of processors required, reason that represent a disadvantage for some scenarios but an improvement for others. The next results present these ideas when the size of the load varies.

Figure 5-8 exhibit the good and bad of reusing data on these criteria. While reused data is still allocated on each SPM, the runtime for light level load decrease 73% of runtime with the four CPUs system. The best case for the heavy load is the four CPUs configuration with 71% runtime reduction. Doubling the number of CPUs gives a small benefit for both loads.

Figure 5-9 shows the cost for the first two parallelization approaches. For the light load level, the best cost is for the system with two processors. This system has a similar cost than the reference system with one CPU. For the heavy load level, the cost has a related behavior with the four CPUs system which maintained constant the cost of the reference system. In both loads appears a breakup point after the four CPUs system.

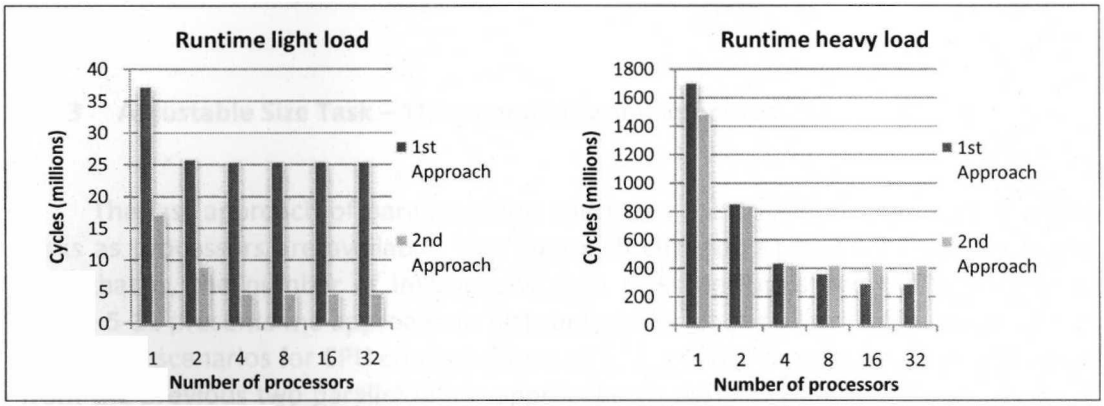


Figure 5-8: Runtime for approach one and two

Figure 5-10 support the conclusion from Figure 5-9; Holding data, as described on the start of this section, does not contribute with performance for all size of loads. For small loads, leaving data on SPM does actually decrease considerably the overhead on parallel systems; but as the load size increases, this does not represent a great reduction of time. This is only a good solution among light level loads.

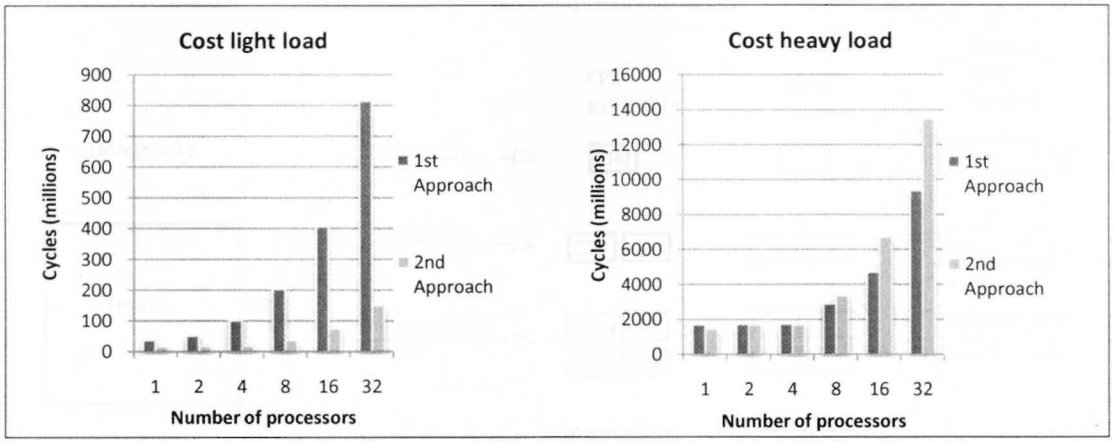


Figure 5-9: Cost for the second parallelization approach

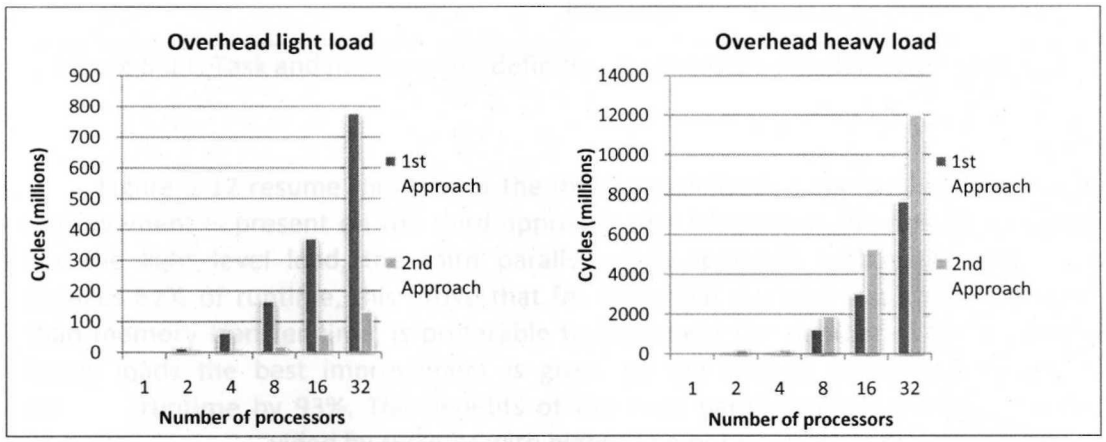


Figure 5-10: Overhead for first two approaches

5.3.3 Adjustable Size Task – Third parallelization approach

The last approach of parallelization addresses the criteria to compute as many tasks as processors are available. This approach changes all task on their respective size, that is, the number of instructions each tasks performs and the size of data. Figure 5-11 presents the approach to distribute tasks among all available processors. It shows the scenarios for CPU configurations of 1, 2, 4 and 8 processors. The difference from the previous two parallelization approaches is that the number of task currently performing is the same as available processors with the effect of using less memory space per processors as shows.

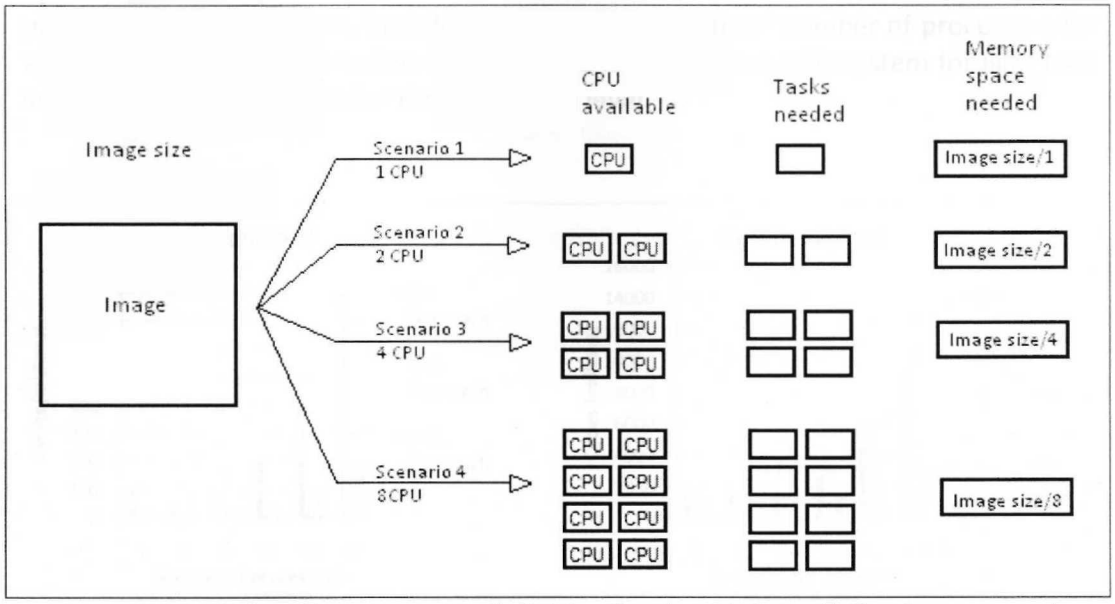


Figure 5-11: Task and memory size definition for the third parallelization approach

Figure 5-12 resume runtime for the three parallelization approaches. The mayor improvement is present on the third approach that increments the number of tasks. For the light level load, the third parallelization approach with eight processors reduces 82% of runtime, this prove that for tasks that demand less computing time than memory transfer time, is preferable to increment the number of tasks. For the heavy loads the best improvement is given by the sixteen processors system, it reduced runtime by 93%. The benefits of the third parallelization approach for the heavy loads are provided by systems with higher number of CPUs.

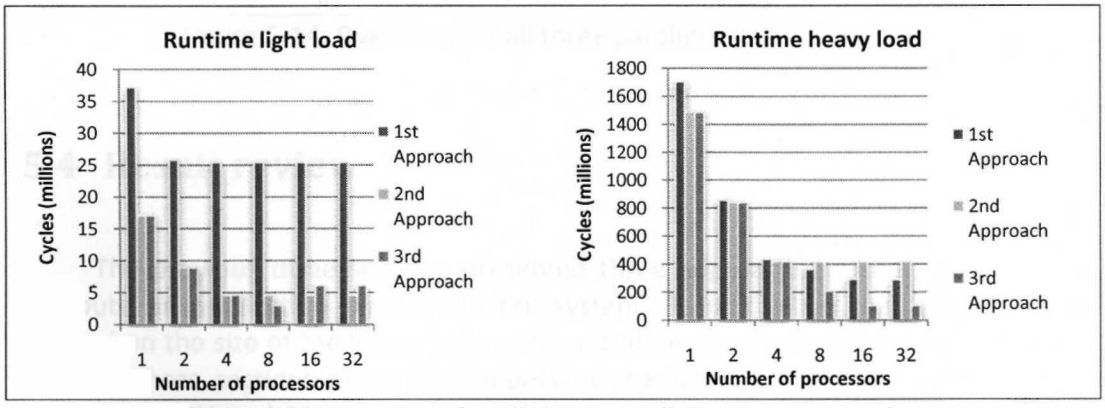


Figure 5-12: Runtime for all three parallelization approaches

Figure 5-13 shows the cost for the last parallelization approach. For the light load level, the cost starts rising from the four-CPU system, and for the heavy load size it starts with the eight-CPU system. Figure 5-14 shows the overhead for the three

parallelization approaches. Both load levels have an optimal number of processors for which the performance reaches a limit. These limits are two-CPU system for light load level and four-CPU system for the heavy load level.

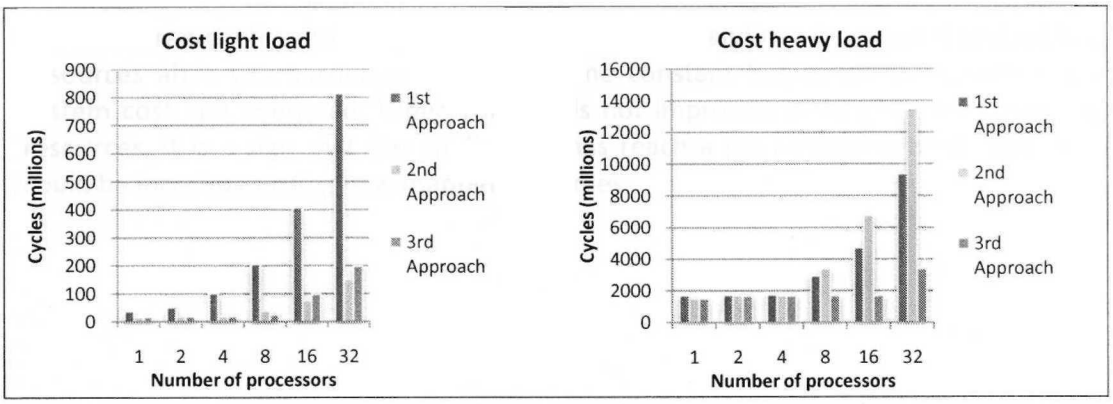


Figure 5-13: Cost for all three parallelization approaches

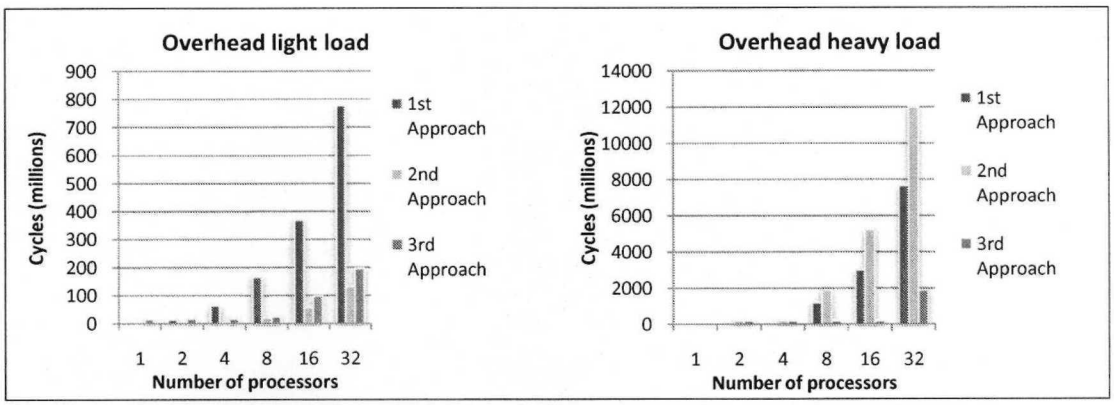


Figure 5-14: Overhead for all three parallelization approaches

5.4 Result review

The previous three sections presented three approaches to distribute data to compute an application on a multicore system. Results prove that improvements depend on the size of the load and the amount of memory transfers required for each task. All three sections present an important characteristic: at some point doubling resources (CPU and SPM size) do not contribute with a reduction on runtime, as a result cost and overhead for those system increases.

The best configuration for both load sizes is as follow. For the first approach the best option for the light level load are the two-CPU system with 8KB SPM and four-CPU system with 16KB SPM for the heavy load level. The second approach also presents a

breakup point after the four-CPU system for both level loads, with 4KB SPM. Finally, the third parallelization approach presents a best case for light level load with the eight-CPU system and sixteen-CPU system for the heavy load. The results showed that the third parallelization approach gives the best runtime over the two other parallelization approaches.

All three configurations present a breakup point, and results prove that doubling resources after this barrier maintains runtime constant but cause an increment on system cost and overhead. When runtime is not improved even when incrementing resources, it is a sign that the application has reach a parallelization limit. This limit could be increment using new mapping schemes.

Chapter 6

Conclusions

6.1 Conclusions

This study presented a parallelization of the H.263. Profiling work exhibits that most of the computing time is spent at the Estimation stage on the H.263, and this stage based its functionality on the SAD technique, this function contributes with 55.64% of total computing time. Data parallelism is the criteria selected to decompose the application. Three parallelization approaches show that the relation between memory transfer time and task execution time must take into account to apply a certain approach in order to reduce runtime and raise at maximum the system efficiency.

The parallelization focused on the Sum of Absolute Difference functions. Computational loads have two sizes depending on the search factor used, these are named light and heavy load levels. These two levels differ on the size of the search factor of the SAD. Bigger search windows exhibit greater video quality but increase computational work, on the contrary, smaller search window demand less computer work.

The first parallelization approach consists on exploiting at maximum SPM memory space. This is the first step on parallelization, it decompose SAD function into different tasks. For the light load the best configuration is the two-core system, it presents a 15% reduction of runtime with 8KB SPM. The heavy load best solution is the four processors system with 16KB SPM, it presents a 33% runtime reduction. Result shows a point where increasing resources such as processors or memory space do not contribute with substantial improvement, runtime remain stable but system's cost increases.

The second approach optimized the tasks with small-factor search windows. It was observed that the light level of load takes little less time than it takes the DMA to transfer data to its corresponding SPM. For the light load the best configuration is the four-core system, it presents a 73% reduction of runtime. The heavy load best solution is the four processors system, it presents a 71% runtime reduction. The difference with the first approach is that once a group of data is loaded to a certain CPU, this CPU will perform all operation involving that given data, as a result, a low number of CPU were required and system cost is lowered.

On the third parallelization approach, the tasks are divided into an equal number of processors available. Results showed a benefit for both load levels. For the light load the best configuration is the eight-core system, it presents a 82% reduction of runtime. The heavy load best solution is the sixteen processors system, it presents a 93% runtime reduction.

At current time, there is not a golden rule to follow in order to obtain a perfect distribution of task. Parallelization goals are: (1) find the optimum way to distribute data into available processors in order to reduce memory transfers and computing time; (2) assemble those characteristics to design specific compilers to exploit at maximum decompositions; (3) make aware programmer designers of the important features of multicore systems and how to exploit those capabilities. The golden age of multiprocessing will come when all this ideas converge.

6.2 Future Work

Apply these approaches to actual multiprocessors on the market embedded on mobile devices such as the SGXMPx multicore (2, 4 and 16 processors) systems from the POWERVR SGX GRAPHICS family of Imagination Technologies. These processing elements are present on variety applications such as Apple devices, with two cores, as the iPad2 and next generation of iPhone, PlayStations Vita, with four cores, and the Texas Instruments OMAP5430.

Modeling tools are as important as the parallelization itself. An important work to do is to enhance those tools to easily evaluate new concepts regarding multiprocessing and the necessary documentation for future researchers to continue the studies. Apply MultiMake to more intensive computer applications such as video processing. And translate those models to physical systems as the IBM Cell B.E. Processors. Conclusions of these results will be of great help to design chips of more than 32 processors.

Appendix

A.1 RDTSC code for use on 'C' language programs

```
unsigned long long rdtsc_()
{
#ifdef SS
return 0;
#else
#define rdtsc(low, high) \
__asm__ __volatile__("rdtsc" : "=a" (low), "=d" (high))
unsigned long low, high;
rdtsc(low, high);
return ((unsigned long long)high << 32) | low;
#endif
```

A.2 SAD 'C' code

```
SAD(unsigned char *ii, unsigned char *act_block, int h_length, int Min_FRAME)
{
 int i;
 int sad = 0;
 unsigned char *kk;

 kk = act_block;
 i = 16;
 while (i--) {
  sad += (abs(*ii    - *kk    ) +abs(*(ii+1 ) - *(kk+1) )
       +abs(*(ii+2) - *(kk+2) ) +abs(*(ii+3 ) - *(kk+3) )
       +abs(*(ii+4) - *(kk+4) ) +abs(*(ii+5 ) - *(kk+5) )
       +abs(*(ii+6) - *(kk+6) ) +abs(*(ii+7 ) - *(kk+7) )
       +abs(*(ii+8) - *(kk+8) ) +abs(*(ii+9 ) - *(kk+9) )
       +abs(*(ii+10)- *(kk+10)) +abs(*(ii+11) - *(kk+11))
       +abs(*(ii+12)- *(kk+12)) +abs(*(ii+13) - *(kk+13))
       +abs(*(ii+14)- *(kk+14)) +abs(*(ii+15) - *(kk+15)) );

  ii += h_length;
  kk += 16;
  if (sad > Min_FRAME)
   return INT_MAX;
```

Appendix

A.1 RDTSC code for use on 'C' language programs

```
unsigned long long rdtsc_()
{
#ifdef SS
return 0;
#else
#define rdtsc(low, high) \
__asm__ __volatile__("rdtsc" : "=a" (low), "=d" (high))
unsigned long low, high;
rdtsc(low, high);
return ((unsigned long long)high << 32) | low;
#endif
```

A.2 SAD 'C' code

```
SAD(unsigned char *ii, unsigned char *act_block, int h_length, int Min_FRAME)
{
 int i;
 int sad = 0;
 unsigned char *kk;

 kk = act_block;
 i = 16;
 while (i--) {
  sad += (abs(*ii    - *kk    ) +abs(*(ii+1 ) - *(kk+1) )
       +abs(*(ii+2) - *(kk+2) ) +abs(*(ii+3 ) - *(kk+3) )
       +abs(*(ii+4) - *(kk+4) ) +abs(*(ii+5 ) - *(kk+5) )
       +abs(*(ii+6) - *(kk+6) ) +abs(*(ii+7 ) - *(kk+7) )
       +abs(*(ii+8) - *(kk+8) ) +abs(*(ii+9 ) - *(kk+9) )
       +abs(*(ii+10)- *(kk+10)) +abs(*(ii+11) - *(kk+11))
       +abs(*(ii+12)- *(kk+12)) +abs(*(ii+13) - *(kk+13))
       +abs(*(ii+14)- *(kk+14)) +abs(*(ii+15) - *(kk+15)) );

  ii += h_length;
  kk += 16;
  if (sad > Min_FRAME)
   return INT_MAX;
```

```
 }
 return sad;
}
```

A.3 SAD assembly code

```
SAD_Macroblock:
        .frame  $sp,0,$31               # vars= 0, regs= 0/0, args= 0, extra= 0
        .mask   0x00000000,0
        .fmask  0x00000000,0
        .def    i;      .val    10;     .scl    4;      .type   0x4;     .endef
        .def    sad;    .val    9;      .scl    4;      .type   0x4;     .endef
        .def    kk;     .val    5;      .scl    4;      .type   0x1c;    .endef
        move    $8,$4

        .loc    1 4006
        move    $9,$0

        .loc    1 4011
        li      $10,0x0000000f                  # 15
        li      $11,-1                  # 0xffffffff
$L1119:

        .loc    1 4012
        .set    noreorder
        lbu     $4,0($8)
        .set    reorder
        .set    noreorder
        lbu     $2,0($5)
        .set    reorder
        .set    noreorder
        lbu     $3,1($5)
        .set    reorder
        subu    $4,$4,$2
        .set    noreorder
        lbu     $2,1($8)
        .set    reorder
        bgez    $4,1f
        #nop
        subu    $4,$0,$4
1:
        subu    $2,$2,$3
        bgez    $2,1f
        #nop
        subu    $2,$0,$2
1:
        addu    $4,$4,$2
        .set    noreorder
        lbu     $2,2($8)
        .set    reorder
        .set    noreorder
        lbu     $3,2($5)
        #nop
        .set    reorder
        subu    $2,$2,$3
        bgez    $2,1f
        #nop
        subu    $2,$0,$2
1:
        addu    $4,$4,$2
        .set    noreorder
        lbu     $2,3($8)
        .set    reorder
        .set    noreorder
        lbu     $3,3($5)
        #nop
```

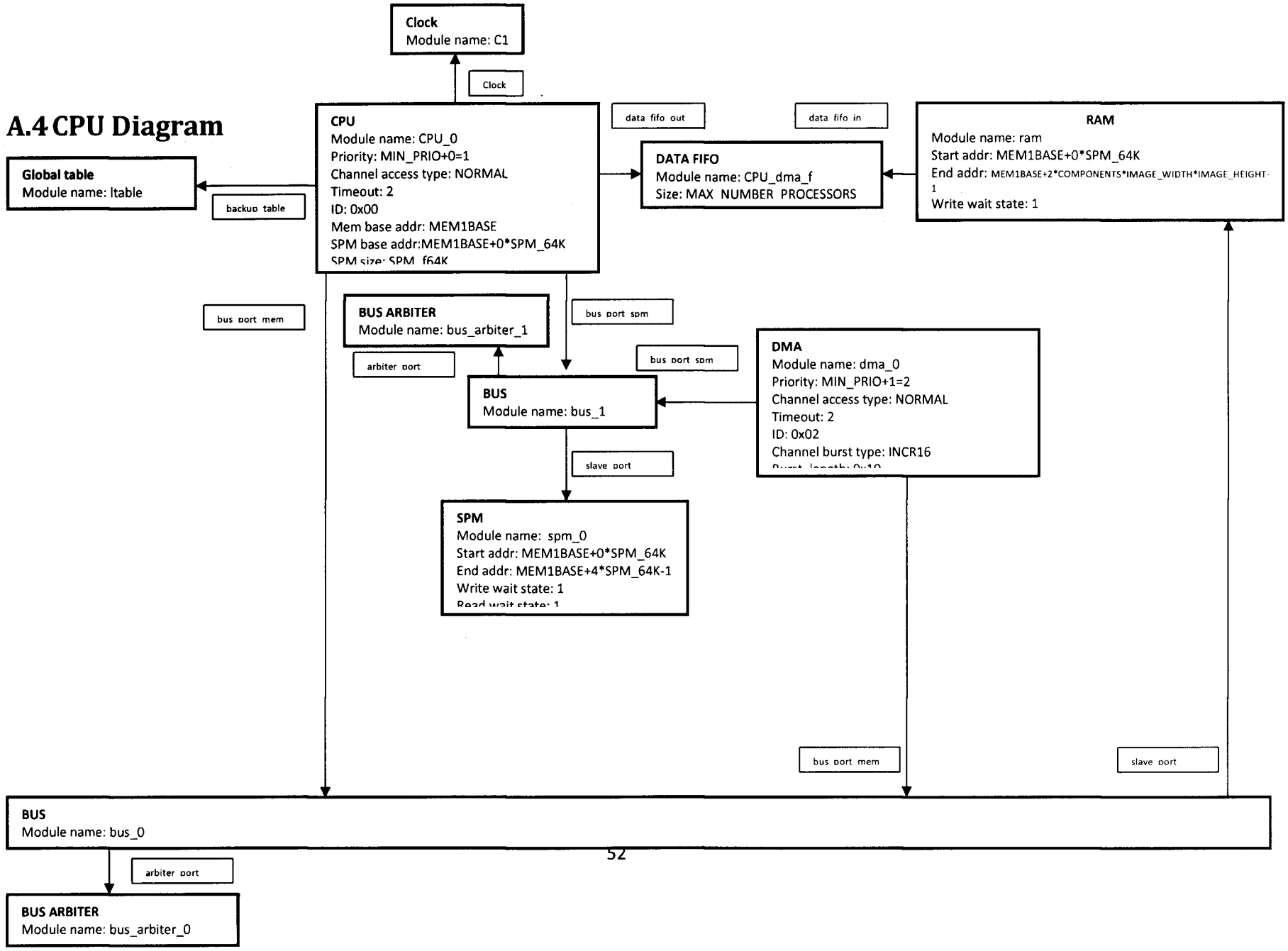
```
 }
 return sad;
}
```

A.3 SAD assembly code

```
SAD_Macroblock:
        .frame  $sp,0,$31               # vars= 0, regs= 0/0, args= 0, extra= 0
        .mask   0x00000000,0
        .fmask  0x00000000,0
        .def    i;      .val    10;     .scl    4;      .type   0x4;     .endef
        .def    sad;    .val    9;      .scl    4;      .type   0x4;     .endef
        .def    kk;     .val    5;      .scl    4;      .type   0x1c;    .endef
        move    $8,$4

        .loc    1 4006
        move    $9,$0

        .loc    1 4011
        li      $10,0x0000000f                  # 15
        li      $11,-1                  # 0xffffffff
$L1119:

        .loc    1 4012
        .set    noreorder
        lbu     $4,0($8)
        .set    reorder
        .set    noreorder
        lbu     $2,0($5)
        .set    reorder
        .set    noreorder
        lbu     $3,1($5)
        .set    reorder
        subu    $4,$4,$2
        .set    noreorder
        lbu     $2,1($8)
        .set    reorder
        bgez    $4,1f
        #nop
        subu    $4,$0,$4
1:
        subu    $2,$2,$3
        bgez    $2,1f
        #nop
        subu    $2,$0,$2
1:
        addu    $4,$4,$2
        .set    noreorder
        lbu     $2,2($8)
        .set    reorder
        .set    noreorder
        lbu     $3,2($5)
        #nop
        .set    reorder
        subu    $2,$2,$3
        bgez    $2,1f
        #nop
        subu    $2,$0,$2
1:
        addu    $4,$4,$2
        .set    noreorder
        lbu     $2,3($8)
        .set    reorder
        .set    noreorder
        lbu     $3,3($5)
        #nop
```

```
        .set    reorder
        subu    $2,$2,$3
        bgez    $2,1f
        #nop
        subu    $2,$0,$2
1:
        addu    $4,$4,$2
        .set    noreorder
        lbu     $2,4($8)
        .set    reorder
        .set    noreorder
        lbu     $3,4($5)
        #nop
        .set    reorder
        subu    $2,$2,$3
        bgez    $2,1f
        #nop
        subu    $2,$0,$2
1:
        addu    $4,$4,$2
        .set    noreorder
        lbu     $2,5($8)
        .set    reorder
        .set    noreorder
        lbu     $3,5($5)
        #nop
        .set    reorder
        subu    $2,$2,$3
        bgez    $2,1f
        #nop
        subu    $2,$0,$2
1:
        addu    $4,$4,$2
        .set    noreorder
        lbu     $2,6($8)
        .set    reorder
        .set    noreorder
        lbu     $3,6($5)
        #nop
        .set    reorder
        subu    $2,$2,$3
        bgez    $2,1f
        #nop
        subu    $2,$0,$2
1:
        addu    $4,$4,$2
        .set    noreorder
        lbu     $2,7($8)
        .set    reorder
        .set    noreorder
        lbu     $3,7($5)
        #nop
        .set    reorder
        subu    $2,$2,$3
        bgez    $2,1f
        #nop
        subu    $2,$0,$2
1:
        addu    $4,$4,$2
        .set    noreorder
```

A.4 CPU Diagram



A.5 Task code on CPU

```
void cpu::Coding_Phase() //
{

//loop: 10
//Description:
//For every macroblock on each image do Estimation()

  printf("Time: %g Name: %s : Start Coding Phase %d\n", sc_simulation_time(), name(),
m_ID);
  int count = 0;

       cycle_inc_n(H_FOR_HEADER);
       for (int i = 0; i < 8; i++) //just for ten block 0 - 9
       {
          cycle_inc_n(H_FOR);
          cycle_inc_n(H_JUMP);
          printf("Time: %g %s  ", sc_simulation_time(), name());
          printf("Block Coded: %d  ", count++);
          printf("current_start_addr_range: %d \n", current_start_addr_range);
          Estimation(i,-1); //calls to estimation with the number of block and the
image offset, for this case 1

       }
       printf("Time: %g Name: %s : End Coding Phase: %d\n", sc_simulation_time(),
name(), m_ID);
}//end Coding_Phase




void cpu::Prediction_Phase_1Back()
{

//loop: 10
//Description:
//For every macroblock on each image do Estimation()

  printf("Time: %g Name: %s : Start Coding Phase %d\n", sc_simulation_time(), name(),
m_ID);
  int count = 0;

       cycle_inc_n(H_FOR_HEADER);
       for (int i = 0; i < 8; i++) //just for ten block 0 - 9
       {
          cycle_inc_n(H_FOR);
          cycle_inc_n(H_JUMP);
          Estimation(i,-1);
          printf("Time: %g  %s  ", sc_simulation_time(), name());
          printf("Block Coded: %d\n", count++);

       }
       printf("Time: %g Name: %s : End Coding Phase: %d\n", sc_simulation_time(),
name(), m_ID);

}


void cpu::Prediction_Phase_1Forward()
{

//loop: 10
//Description:
//For every macroblock on each image do Estimation()

  printf("Time: %g Name: %s : Start Coding Phase %d\n", sc_simulation_time(), name(),
m_ID);
  int count = 0;

       cycle_inc_n(H_FOR_HEADER);
```

```
        for (int i = 0; i < 8; i++) //just for ten block 0 - 9
        {
            cycle_inc_n(H_FOR);
            cycle_inc_n(H_JUMP);
            Estimation(i,1);
            printf("Time: %g Name: %s  CPU: %d\n", sc_simulation_time(), name(), m_ID);
            printf("Block Coded: %d\n", count++);

        }
        printf("Time: %g Name: %s : End Coding Phase: %d\n", sc_simulation_time(),
name(), m_ID);

}

void cpu::Prediction_Phase_2Forward()
{

//loop: 10
//Description:
//For every macroblock on each image do Estimation()

  printf("Time: %g Name: %s : Start Coding Phase %d\n", sc_simulation_time(), name(),
m_ID);
  int count = 0;

        cycle_inc_n(H_FOR_HEADER);
        for (int i = 0; i < 8; i++) //just for ten block 0 - 9
        {
            cycle_inc_n(H_FOR);
            cycle_inc_n(H_JUMP);
            Estimation(i,2);
            printf("Time: %g Name: %s  CPU: %d\n", sc_simulation_time(), name(), m_ID);
            printf("Block Coded: %d\n", count++);

        }
        printf("Time: %g Name: %s : End Coding Phase: %d\n", sc_simulation_time(),
name(), m_ID);

}


void cpu::Estimation(int block, int image_offset) //method for CodeOneOrTwo and NextToPB
{

//loop: 960
//Description:
//For each analysis (one analysis per pixel), find the SAD

//For each call to Estimation() (including SAD_Macroblock)
     //reads: 491 520
     //writes: 960

   int sxy = 1;  //should be 15

 // printf("Time: %g Name: %s : Start Estimation(): %d\n", sc_simulation_time(), name(),
m_ID);


  /* Spiral search */
  cycle_inc_n(H_FOR_HEADER);
  for (int l = 1; l <= sxy; l++){
  cycle_inc_n(H_FOR);
     cycle_inc_n(H_FOR_HEADER);
     for (int k = 0; k < 8*l; k++){
     cycle_inc_n(H_FOR);
        /* 16x16 integer pel MV */
     cycle_inc_n(H_JUMP);
        SAD_Macroblock(block, image_offset); //find the sad of a vector
     look_up_table->update_task_iteration(m_ID, current_task_id, l*sxy+k);

     }
     //printf("%d \n", l);
  }
```

```
 // printf("Time: %g Name: %s : End Estimation(): %d\n", sc_simulation_time(), name(),
m_ID);

}//end Estimation()


void cpu::SAD_Macroblock(int block, int image_offset) //method for Estimation()
{

//reads: 256 bytes
//reads: Image + 256 bytes
//writes: 1 byte
//cicles: 833

//Description:
//For each Estimation do 960 SAD analysis

//inside Estimation()
      //reads: 491 520
      //writes: 960

//inside Estimation_Picture:
      //reads: 778 567 680
      //writes: 1 520 640     printf("Time: %g Name: %s : End NextTwoPB cpu_id: %d\n",
sc_simulation_time(), name(), m_ID);


  int sad = 0;
  int ii=0;
  int jj=0;
  int counter=0;
MEMORY_DATA_TYPE data_1;
MEMORY_DATA_TYPE data_2;
MEMORY_DATA_TYPE tmp;


      for(int i = 0; i<32; i++){
//    printf("%d \n",counter + block*BLOCK_SIZE, counter + block*BLOCK_SIZE + 2048);
            cycle_inc_n(H_READ);
            data_1 = read_next_element(counter + block*BLOCK_SIZE);
            cycle_inc_n(H_READ);
            data_2 = read_next_element(counter + block*BLOCK_SIZE + 2048);
            cycle_inc_n(H_NOP);
            cycle_inc_n(H_SUB);
            tmp = data_1 - data_2;
            cycle_inc_n(H_BRANCH);
            if(tmp >= 0){
            cycle_inc_n(H_NOP);
            cycle_inc_n(H_SUB);
            tmp = 0 - tmp;
            }
            cycle_inc_n(H_ADD);
            sad = sad + tmp;

      look_up_table->update_task_iteration(m_ID, current_task_id, counter);
      counter = counter + 1;
//    printf("%d\n", counter++);
      cycle_inc_n(H_ADD);
      ii = ii + 31;
      cycle_inc_n(H_ADD);
      jj = jj + 16;
      }
      write_next_element(block + current_start_addr_range/256, sad); //write on image-
>next_1
      //printf("Wrote on %d \n", block + current_start_addr_range/256);

}//end SAD_Macroblock()



void cpu::Predict_P() //method for CodeOneOrTwo
{
//reads: 256 bytes
//reads: Image + 256 bytes
//writes: 256 bytes
```

```
MEMORY_DATA_TYPE data_1;
MEMORY_DATA_TYPE data_2;

  cycle_inc_n(H_FOR_HEADER);
    for (int n = 0; n < MB_SIZE; n++){
    cycle_inc_n(H_FOR);
      cycle_inc_n(H_FOR_HEADER);
      for (int m = 0; m < MB_SIZE; m++){
      cycle_inc_n(H_FOR);
     data_1 = read_next_element(16*n + m); //read data from image->next_1
//    data_2 = read_next_element(IMAGE_SIZE + 16*n + m); //read data from image->next_2
     data_2 = read_next_element(16*n + m); //read data from image->next_2
     cycle_inc_n(H_INSTRUCTION);
     data_1 = data_1 - data_2;
     write_next_element(16*n + m, data_1); //write on image->next_1
     look_up_table->update_task_iteration(m_ID, current_task_id, n*MB_SIZE+m);
     }
       }
}
```

A.6 Example of coding stage

```
    task_info t0;
    t0.task_name = "CodingPhase";
    t0.task_type = CodingPhase;
    t0.task_id = 0;
    t0.cpu_id = 0;
    t0.start_address = 0;
    t0.end_address = 4096;
    t0.offset = 0 + 1*IMAGE_SIZE;
    t0.pipe_stage = 1;
    t0.iteration = -1;
    t0.cpu_id_dep = -1;
    //t0.read_lut.add_lut_entry(0 + 0*IMAGE_SIZE, 4096 + 0*IMAGE_SIZE, MEM1BASE +
0*SPM_4K, true);
    t0.read_lut.add_lut_entry(0 + 1*IMAGE_SIZE, 4096 + 1*IMAGE_SIZE, MEM1BASE +
0*SPM_4K, true);
    t0.write_lut.add_lut_entry(0 + 1*IMAGE_SIZE, 4096 + 1*IMAGE_SIZE, MEM1BASE +
0*SPM_4K, true);

    task_info t1;
    t1.task_name = "CodingPhase";
    t1.task_type = CodingPhase;
    t1.task_id = 1;
    t1.cpu_id = 1;
    t1.start_address = 0;
    t1.end_address = 4096;
    t1.offset = 4096 + 1*IMAGE_SIZE;
    t1.pipe_stage = 1;
    t1.iteration = -1;
    t1.cpu_id_dep = -1;
    //t1.read_lut.add_lut_entry(4096 + 0*IMAGE_SIZE, 8192 + 0*IMAGE_SIZE, MEM1BASE +
1*SPM_4K, true);
    t1.read_lut.add_lut_entry(4096 + 1*IMAGE_SIZE, 8192 + 1*IMAGE_SIZE, MEM1BASE +
1*SPM_4K, true);
    t1.write_lut.add_lut_entry(4096 + 1*IMAGE_SIZE, 8192 + 1*IMAGE_SIZE, MEM1BASE +
1*SPM_4K, true);
    task_info t2;
    t2.task_name = "CodingPhase";
    t2.task_type = CodingPhase;
    t2.task_id = 2;
    t2.cpu_id = 2;
    t2.start_address = 0;
    t2.end_address = 4096;
    t2.offset = 8192 + 1*IMAGE_SIZE;
    t2.pipe_stage = 1;
    t2.iteration = -1;
    t2.cpu_id_dep = -1;
    //t2.read_lut.add_lut_entry(8192 + 0*IMAGE_SIZE, 12288 + 0*IMAGE_SIZE, MEM1BASE +
2*SPM_4K, true);
    t2.read_lut.add_lut_entry(8192 + 1*IMAGE_SIZE, 12288 + 1*IMAGE_SIZE, MEM1BASE +
2*SPM_4K, true);
    t2.write_lut.add_lut_entry(8192 + 1*IMAGE_SIZE, 12288 + 1*IMAGE_SIZE, MEM1BASE +
2*SPM_4K, true);
    task_info t3;
    t3.task_name = "CodingPhase";
    t3.task_type = CodingPhase;
    t3.task_id = 3;
    t3.cpu_id = 3;
    t3.start_address = 0;
    t3.end_address = 4096;
    t3.offset = 12288 + 1*IMAGE_SIZE;
    t3.pipe_stage = 1;
    t3.iteration = -1;
    t3.cpu_id_dep = -1;
    //t3.read_lut.add_lut_entry(12288 + 0*IMAGE_SIZE, 16384 + 0*IMAGE_SIZE, MEM1BASE +
3*SPM_4K, true);
    t3.read_lut.add_lut_entry(12288 + 1*IMAGE_SIZE, 16384 + 1*IMAGE_SIZE, MEM1BASE +
3*SPM_4K, true);
    t3.write_lut.add_lut_entry(12288 + 1*IMAGE_SIZE, 16384 + 1*IMAGE_SIZE, MEM1BASE +
3*SPM_4K, true);
```

A.7 Example of prediction stage

```
//PredictionPhase
      task_info t10;
      t10.task_name = "PredictionPhase1Back";
      t10.task_type = PredictionPhase1Back;
      t10.task_id = 10;
      t10.cpu_id = 8;
      t10.start_address = 0;
      t10.end_address = 4096;
      t10.offset = 0 + 2*IMAGE_SIZE;
      t10.pipe_stage = 1;
      t10.iteration = -1;
      t10.cpu_id_dep = 4;
      t10.add_dependence(5);
      t10.add_dependence(6);
      t10.add_dependence(7);
      t10.add_dependence(8);
      //t10.read_lut.add_lut_entry(0  +  1*IMAGE_SIZE,  4096  +  1*IMAGE_SIZE,  MEM1BASE  +
8*SPM_4K, true);
      t10.read_lut.add_lut_entry(0   +   2*IMAGE_SIZE,4096   +   2*IMAGE_SIZE,MEM1BASE   +
8*SPM_4K, true);
      t10.write_lut.add_lut_entry(0  +  2*IMAGE_SIZE,  4096  +  2*IMAGE_SIZE,  MEM1BASE  +
8*SPM_4K, true);
      task_info t11;
      t11.task_name = "PredictionPhase1Back";
      t11.task_type = PredictionPhase1Back;
      t11.task_id = 11;
      t11.cpu_id = 9;
      t11.start_address = 0;
      t11.end_address = 4096;
      t11.offset = 4096 + 2*IMAGE_SIZE;
      t11.pipe_stage = 1;
      t11.iteration = -1;
      t11.cpu_id_dep = 4;
      t11.add_dependence(5);
      t11.add_dependence(6);
      t11.add_dependence(7);
      t11.add_dependence(8);
      //t11.read_lut.add_lut_entry(4096 + 1*IMAGE_SIZE, 8192 + 1*IMAGE_SIZE, MEM1BASE +
9*SPM_4K, true);
      t11.read_lut.add_lut_entry(4096   +   2*IMAGE_SIZE,8192   +   2*IMAGE_SIZE,MEM1BASE   +
9*SPM_4K, true);
      t11.write_lut.add_lut_entry(4096 + 2*IMAGE_SIZE, 8192 + 2*IMAGE_SIZE, MEM1BASE +
9*SPM_4K, true);
      task_info t12;
      t12.task_name = "PredictionPhase1Back";
      t12.task_type = PredictionPhase1Back;
      t12.task_id = 12;
      t12.cpu_id = 10;
      t12.start_address = 0;
      t12.end_address = 4096;
      t12.offset = 8192 + 2*IMAGE_SIZE;
      t12.pipe_stage = 1;
      t12.iteration = -1;
      t12.cpu_id_dep = 4;
      t12.add_dependence(5);
      t12.add_dependence(6);
      t12.add_dependence(7);
      t12.add_dependence(8);
      //t12.read_lut.add_lut_entry(8192 + 1*IMAGE_SIZE, 12288 + 1*IMAGE_SIZE, MEM1BASE +
10*SPM_4K, true);
      t12.read_lut.add_lut_entry(8192   +   2*IMAGE_SIZE,12288   +   2*IMAGE_SIZE,MEM1BASE   +
10*SPM_4K, true);
      t12.write_lut.add_lut_entry(8192 + 2*IMAGE_SIZE, 12288 + 2*IMAGE_SIZE, MEM1BASE +
10*SPM_4K, true);
      task_info t90;
      t90.task_name = "PredictionPhase1Back";
      t90.task_type = PredictionPhase1Back;
      t90.task_id = 90;
      t90.cpu_id = 11;
```

```
	t90.start_address = 0;
	t90.end_address = 4096;
	t90.offset = 12288 + 2*IMAGE_SIZE;
	t90.pipe_stage = 1;
	t90.iteration = -1;
	t90.cpu_id_dep = 4;
	t90.add_dependence(5);
	t90.add_dependence(6);
	t90.add_dependence(7);
	t90.add_dependence(8);
	//t90.read_lut.add_lut_entry(12288 + 1*IMAGE_SIZE, 16384 + 1*IMAGE_SIZE, MEM1BASE +
11*SPM_4K, true);
	t90.read_lut.add_lut_entry(12288 + 2*IMAGE_SIZE,16384 + 2*IMAGE_SIZE,MEM1BASE +
11*SPM_4K, true);
	t90.write_lut.add_lut_entry(12288 + 2*IMAGE_SIZE, 16384 + 2*IMAGE_SIZE, MEM1BASE +
11*SPM_4K, true);

	task_info t13;
	t13.task_name = "PredictionPhase1Forward";
	t13.task_type = PredictionPhase1Forward;
	t13.task_id = 13;
	t13.cpu_id = 12;
	t13.start_address = 0;
	t13.end_address = 4096;
	t13.offset = 0 + 2*IMAGE_SIZE;
	t13.pipe_stage = 1;
	t13.iteration = -1;
	t13.cpu_id_dep = 4;
	t13.add_dependence(5);
	t13.add_dependence(6);
	t13.add_dependence(7);
	t13.add_dependence(8);
	//t13.read_lut.add_lut_entry(0 + 3*IMAGE_SIZE, 4096 + 3*IMAGE_SIZE, MEM1BASE +
12*SPM_4K, true);
	t13.read_lut.add_lut_entry(0 + 2*IMAGE_SIZE,4096 + 2*IMAGE_SIZE,MEM1BASE +
12*SPM_4K, true);
	t13.write_lut.add_lut_entry(0 + 2*IMAGE_SIZE, 4096 + 2*IMAGE_SIZE, MEM1BASE +
12*SPM_4K, true);
	task_info t14;
	t14.task_name = "PredictionPhase1Forward";
	t14.task_type = PredictionPhase1Forward;
	t14.task_id = 14;
	t14.cpu_id = 13;
	t14.start_address = 0;
	t14.end_address = 4096;
	t14.offset = 4096 + 2*IMAGE_SIZE;
	t14.pipe_stage = 1;
	t14.iteration = -1;
	t14.cpu_id_dep = 4;
	t14.add_dependence(5);
	t14.add_dependence(6);
	t14.add_dependence(7);
	t14.add_dependence(8);
	//t14.read_lut.add_lut_entry(4096 + 3*IMAGE_SIZE, 8192 + 3*IMAGE_SIZE, MEM1BASE +
13*SPM_4K, true);
	t14.read_lut.add_lut_entry(4096 + 2*IMAGE_SIZE,8192 + 2*IMAGE_SIZE,MEM1BASE +
13*SPM_4K, true);
	t14.write_lut.add_lut_entry(4096 + 2*IMAGE_SIZE, 8192 + 2*IMAGE_SIZE, MEM1BASE +
13*SPM_4K, true);
	task_info t15;
	t15.task_name = "PredictionPhase1Forward";
	t15.task_type = PredictionPhase1Forward;
	t15.task_id = 15;
	t15.cpu_id = 14;
	t15.start_address = 0;
	t15.end_address = 4096;
	t15.offset = 8192 + 2*IMAGE_SIZE;
	t15.pipe_stage = 1;
	t15.iteration = -1;
	t15.cpu_id_dep = 4;
	t15.add_dependence(5);
	t15.add_dependence(6);
	t15.add_dependence(7);
	t15.add_dependence(8);
```

```
    //t15.read_lut.add_lut_entry(8192 + 3*IMAGE_SIZE, 12288 + 3*IMAGE_SIZE, MEM1BASE +
14*SPM_4K, true);
    t15.read_lut.add_lut_entry(8192 + 2*IMAGE_SIZE,12288 + 2*IMAGE_SIZE,MEM1BASE +
14*SPM_4K, true);
    t15.write_lut.add_lut_entry(8192 + 2*IMAGE_SIZE, 12288 + 2*IMAGE_SIZE, MEM1BASE +
14*SPM_4K, true);
    task_info t91;
    t91.task_name = "PredictionPhase1Forward";
    t91.task_type = PredictionPhase1Forward;
    t91.task_id = 91;
    t91.cpu_id = 15;
    t91.start_address = 0;
    t91.end_address = 4096;
    t91.offset = 12288 + 2*IMAGE_SIZE;
    t91.pipe_stage = 1;
    t91.iteration = -1;
    t91.cpu_id_dep = 4;
    t91.add_dependence(5);
    t91.add_dependence(6);
    t91.add_dependence(7);
    t91.add_dependence(8);
    //t91.read_lut.add_lut_entry(12288 + 3*IMAGE_SIZE, 16384 + 3*IMAGE_SIZE, MEM1BASE +
15*SPM_4K, true);
    t91.read_lut.add_lut_entry(12288 + 2*IMAGE_SIZE,16384 + 2*IMAGE_SIZE,MEM1BASE +
15*SPM_4K, true);
    t91.write_lut.add_lut_entry(12288 + 2*IMAGE_SIZE, 16384 + 2*IMAGE_SIZE, MEM1BASE +
15*SPM_4K, true);

    task_info t16;
    t16.task_name = "PredictionPhase2Forward";
    t16.task_type = PredictionPhase2Forward;
    t16.task_id = 16;
    t16.cpu_id = 16;
    t16.start_address = 0;
    t16.end_address = 4096;
    t16.offset = 0 + 2*IMAGE_SIZE;
    t16.pipe_stage = 1;
    t16.iteration = -1;
    t16.cpu_id_dep = 4;
    t16.add_dependence(5);
    t16.add_dependence(6);
    t16.add_dependence(7);
    t16.add_dependence(8);
    //t16.read_lut.add_lut_entry(0 + 4*IMAGE_SIZE, 4096 + 4*IMAGE_SIZE, MEM1BASE +
16*SPM_4K, true);
    t16.read_lut.add_lut_entry(0 + 2*IMAGE_SIZE,4096 + 2*IMAGE_SIZE,MEM1BASE +
16*SPM_4K, true);
    t16.write_lut.add_lut_entry(0 + 2*IMAGE_SIZE, 4096 + 2*IMAGE_SIZE, MEM1BASE +
16*SPM_4K, true);


    task_info t17;
    t17.task_name = "PredictionPhase2Forward";
    t17.task_type = PredictionPhase2Forward;
    t17.task_id = 17;
    t17.cpu_id = 17;
    t17.start_address = 0;
    t17.end_address = 4096;
    t17.offset = 4096 + 2*IMAGE_SIZE;
    t17.pipe_stage = 1;
    t17.iteration = -1;
    t17.cpu_id_dep = -1;
    //t17.read_lut.add_lut_entry(4096 + 2*IMAGE_SIZE, 8192 + 2*IMAGE_SIZE, MEM1BASE +
17*SPM_4K, true);
    t17.read_lut.add_lut_entry(4096 + 2*IMAGE_SIZE, 8192 + 2*IMAGE_SIZE, MEM1BASE +
17*SPM_4K, true);
    t17.write_lut.add_lut_entry(4096 + 2*IMAGE_SIZE, 8192 + 2*IMAGE_SIZE, MEM1BASE +
17*SPM_4K, true);



    task_info t18;
    t18.task_name = "PredictionPhase2Forward";
    t18.task_type = PredictionPhase2Forward;
```

```
	t18.task_id = 18;
	t18.cpu_id = 18;
	t18.start_address = 0;
	t18.end_address = 4096;
	t18.offset = 8192 + 2*IMAGE_SIZE;
	t18.pipe_stage = 1;
	t18.iteration = -1;
	t18.cpu_id_dep = 4;
	t18.add_dependence(5);
	t18.add_dependence(6);
	t18.add_dependence(7);
	t18.add_dependence(8);
	//t18.read_lut.add_lut_entry(8192 + 4*IMAGE_SIZE, 12288 + 4*IMAGE_SIZE, MEM1BASE +
18*SPM_4K, true);
	t18.read_lut.add_lut_entry(8192  +  2*IMAGE_SIZE,12288  +  2*IMAGE_SIZE,MEM1BASE  +
18*SPM_4K, true);
	t18.write_lut.add_lut_entry(8192 + 2*IMAGE_SIZE, 12288 + 2*IMAGE_SIZE, MEM1BASE +
18*SPM_4K, true);
	task_info t92;
	t92.task_name = "PredictionPhase2Forward";
	t92.task_type = PredictionPhase2Forward;
	t92.task_id = 92;
	t92.cpu_id = 19;
	t92.start_address = 0;
	t92.end_address = 4096;
	t92.offset = 12288 + 2*IMAGE_SIZE;
	t92.pipe_stage = 1;
	t92.iteration = -1;
	t92.cpu_id_dep = 4;
	t92.add_dependence(5);
	t92.add_dependence(6);
	t92.add_dependence(7);
	t92.add_dependence(8);
	//t92.read_lut.add_lut_entry(12288 + 4*IMAGE_SIZE, 16384 + 4*IMAGE_SIZE, MEM1BASE +
19*SPM_4K, true);
	t92.read_lut.add_lut_entry(12288  +  2*IMAGE_SIZE,16384  +  2*IMAGE_SIZE,MEM1BASE  +
19*SPM_4K, true);
	t92.write_lut.add_lut_entry(12288 + 2*IMAGE_SIZE, 16384 + 2*IMAGE_SIZE, MEM1BASE +
19*SPM_4K, true);
```

A.8 Complete results for First Parallelization Approach (graphs)

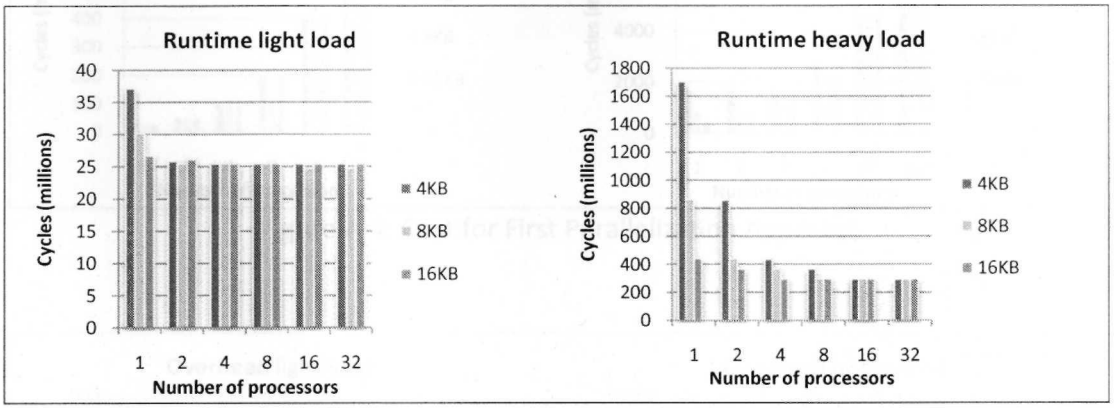


Figure A-1: Runtime for First Parallelization Approach

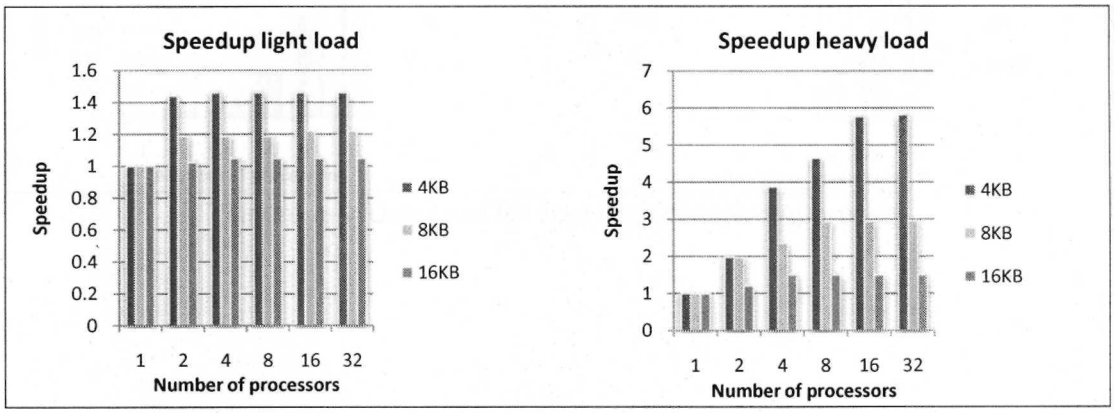


Figure A-2: Speedup for First Parallelization Approach

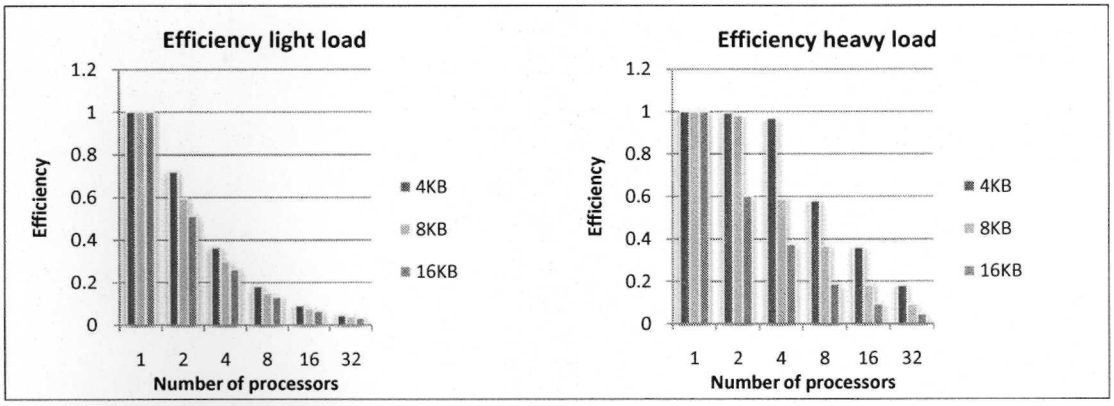


Figure A-3: Efficiency for First Parallelization Approach

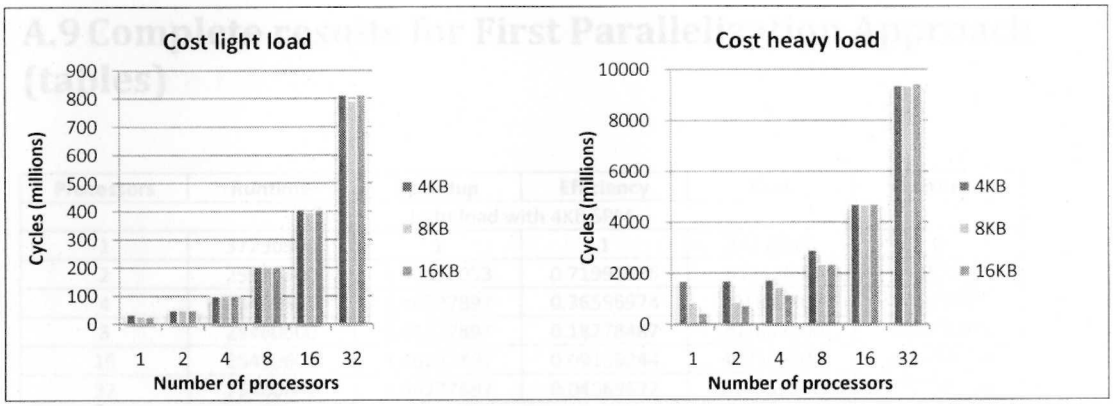


Figure A-4: Cost for First Parallelization Approach

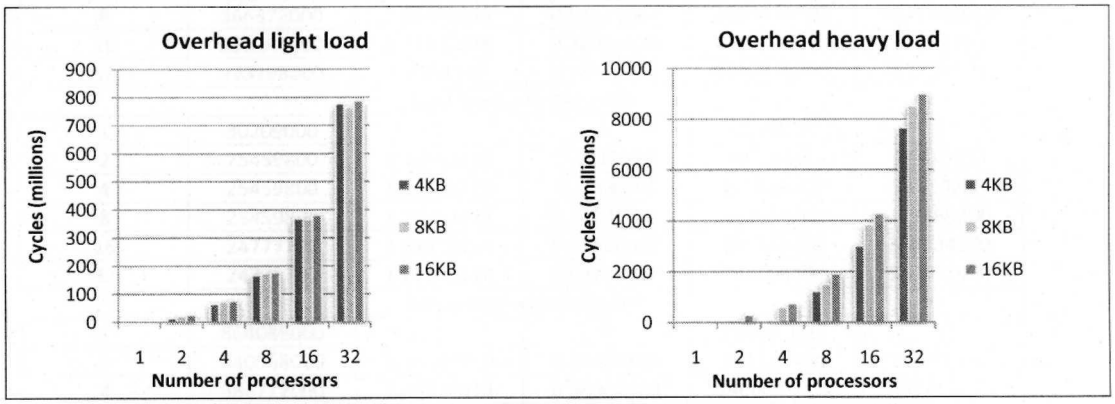


Figure A-5: Overhead for First Parallelization Approach

A.9 Complete results for First Parallelization Approach (tables)

Processors	Runtime	Speedup	Efficiency	Cost	Overhead
Light load with 4KB SPM					
1	37230500	1	1	37230500	0
2	25856300	1.43990053	0.71995026	51712600	14482100
4	25460600	1.46227897	0.36556974	101842400	64611900
8	25460600	1.46227897	0.18278487	203684800	166454300
16	25460600	1.46227897	0.09139244	407369600	370139100
32	25460600	1.46227897	0.04569622	814739200	777508700
Heavy load with 4KB SPM					
1	1704980000	1	1	1704980000	0
2	858118000	1.98688292	0.99344146	1716236000	11256000
4	439545000	3.87896575	0.96974144	1758180000	53200000
8	366372000	4.65368533	0.58171067	2930976000	1225996000
16	295244000	5.77481676	0.36092605	4723904000	3018924000
32	293198000	5.8151147	0.18172233	9382336000	7677356000
Light load with 8KB SPM					
1	30209000	1	1	30209000	0
2	25459800	1.18653721	0.5932686	50919600	20710600
4	25459800	1.18653721	0.2966343	101839200	71630200
8	25459800	1.18653721	0.14831715	203678400	173469400
16	24777700	1.21920114	0.07620007	396443200	366234200
32	24777700	1.21920114	0.03810004	792886400	762677400
Heavy load with 8KB SPM					
1	864085000	1	1	864085000	0
2	440568000	1.96129769	0.98064884	881136000	17051000
4	366712000	2.35630413	0.58907603	1466848000	602763000
8	295243000	2.9266909	0.36583636	2361944000	1497859000
16	294561000	2.93346709	0.18334169	4712976000	3848891000
32	293197000	2.94711406	0.09209731	9382304000	8518219000
Light load with 16KB SPM					
1	26698300	1	1	26698300	0
2	26078800	1.02375493	0.51187746	52157600	25459300
4	25459300	1.04866591	0.26216648	101837200	75138900
8	25459300	1.04866591	0.13108324	203674400	176976100
16	25459300	1.04866591	0.06554162	407348800	380650500
32	25459300	1.04866591	0.03277081	814697600	787999300
Heavy load with 16KB SPM					
1	443636000	1	1	443636000	0
2	368757000	1.20305784	0.60152892	737514000	293878000
4	295243000	1.5026131	0.37565328	1180972000	737336000
8	295243000	1.5026131	0.18782664	2361944000	1918308000
16	295243000	1.5026131	0.09391332	4723888000	4280252000
32	295243000	1.5026131	0.04695666	9447776000	9004140000

Table A-1: First Parallelization Approach

A.10 Complete results for Second Parallelization Approach (graphs)

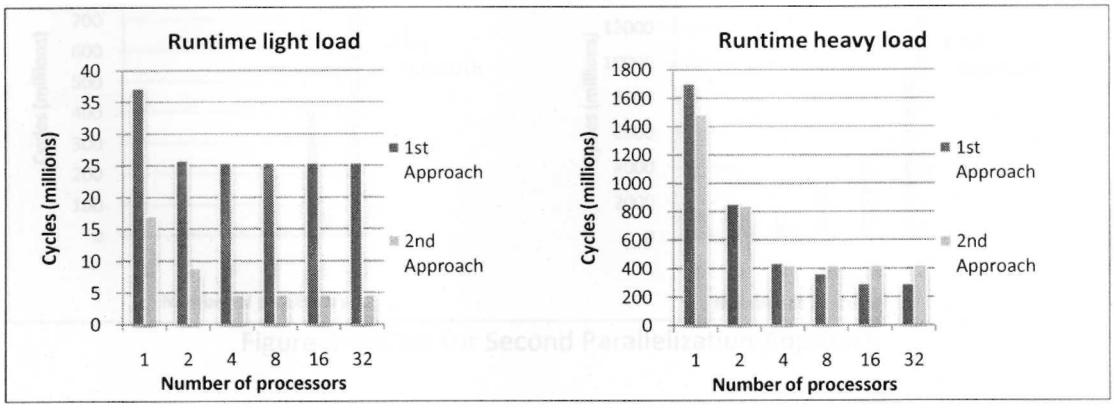


Figure A-6: Runtime for Second Parallelization Approach

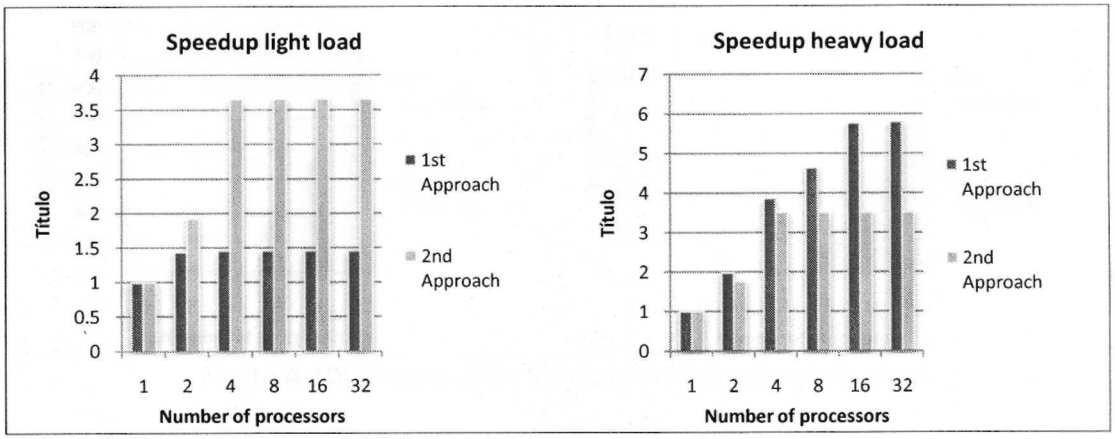


Figure A-7: Speed for Second Parallelization Approach

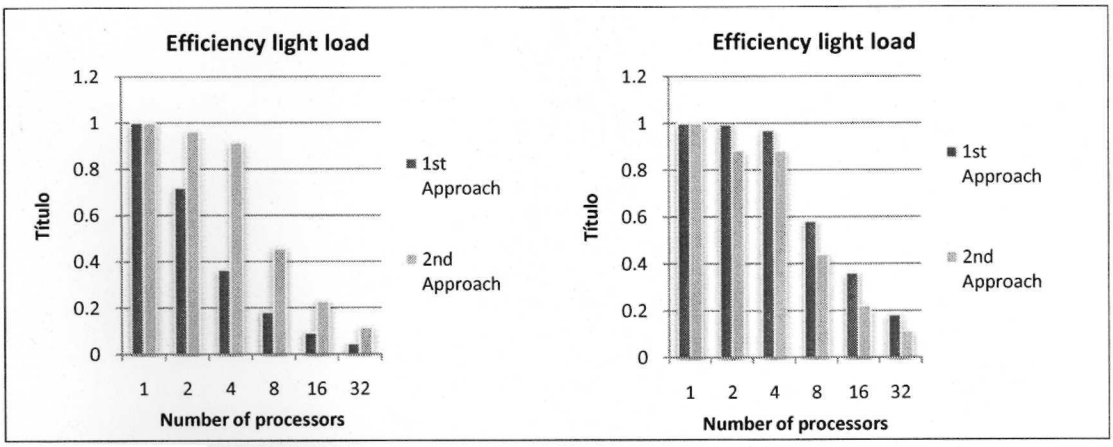


Figure A-8: Efficiency for Second Parallelization Approach

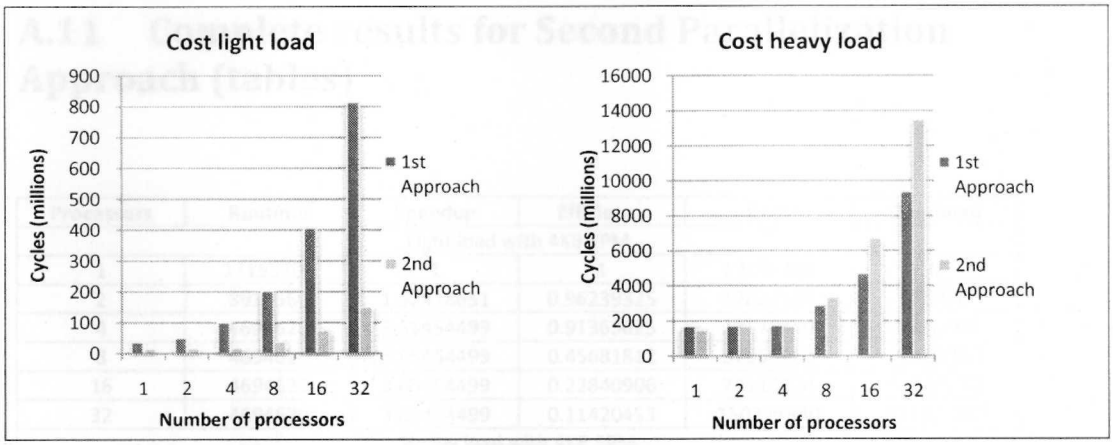


Figure A-9: Cost for Second Parallelization Approach

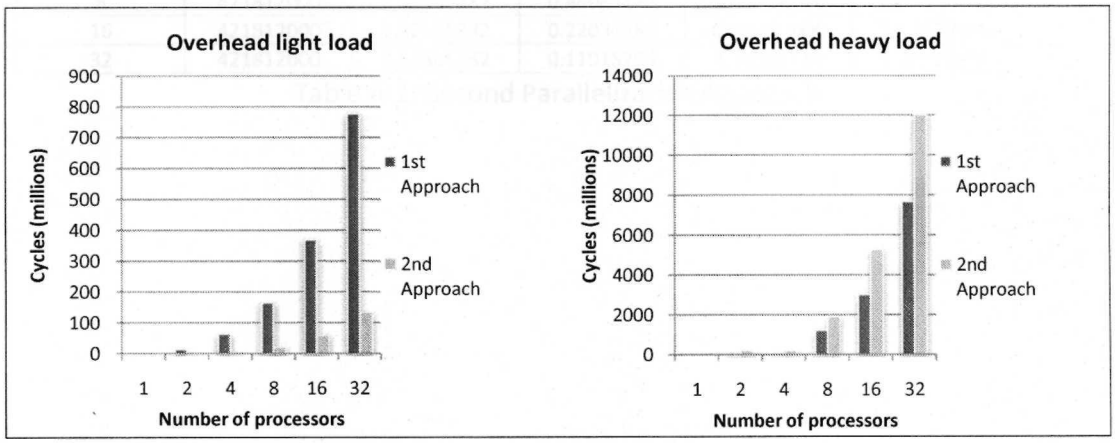


Figure A-10: Overhead for Second Parallelization Approach

A.11 Complete results for Second Parallelization Approach (tables)

Processors	Runtime	Speedup	Efficiency	Cost	Overhead
Light load with 4KB SPM					
1	17156700	1	1	17156700	0
2	8913560	1.92478651	0.96239325	17827120	670420
4	4694620	3.65454499	0.91363625	18778480	1621780
8	4694620	3.65454499	0.45681812	37556960	20400260
16	4694620	3.65454499	0.22840906	75113920	57957220
32	4694620	3.65454499	0.11420453	150227840	133071140
Heavy load with 4KB SPM					
1	1486910000	1	1	1486910000	0
2	842602000	1.76466469	0.88233235	1685204000	198294000
4	421812000	3.52505382	0.88126345	1687248000	200338000
8	421812000	3.52505382	0.44063173	3374496000	1887586000
16	421812000	3.52505382	0.22031586	6748992000	5262082000
32	421812000	3.52505382	0.11015793	1.3498E+10	1.2011E+10

Table A-2: Second Parallelization Approach

A.12 Complete results for Third Parallelization Approach (graphs)

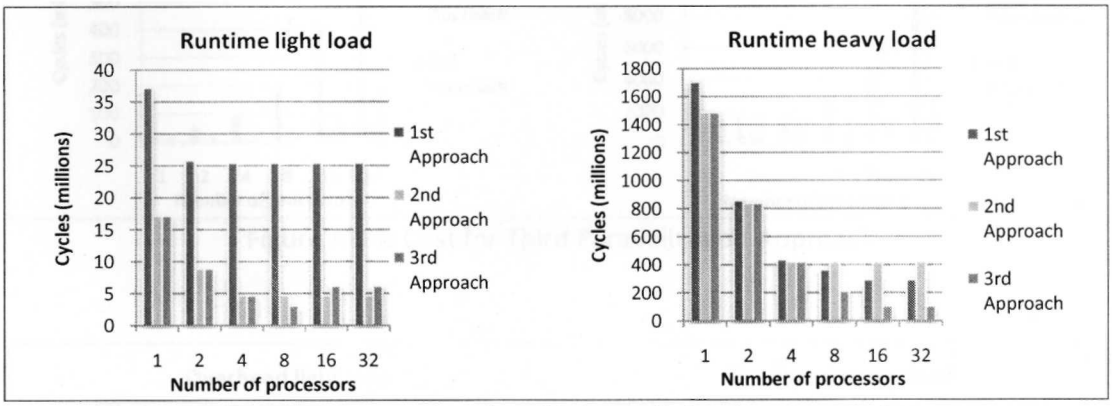


Figure A-11: Runtime for Third Parallelization Approach

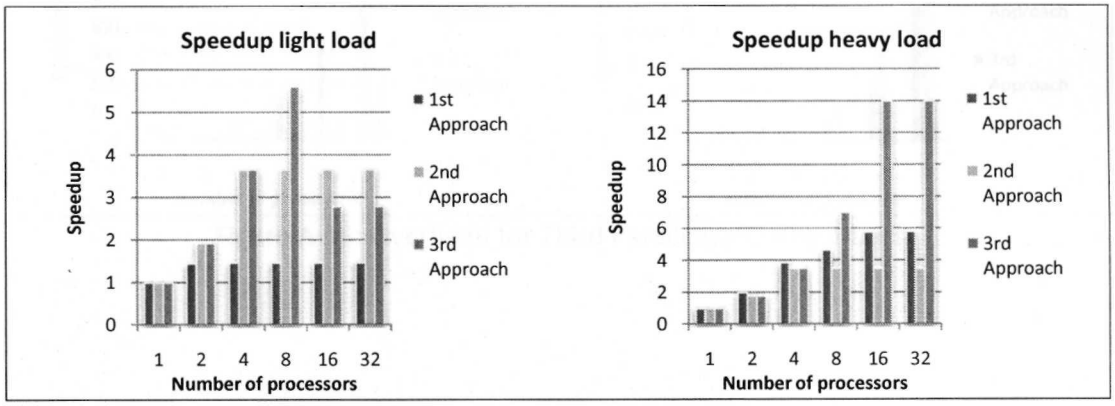


Figure A-12: Speedup for Third Parallelization Approach

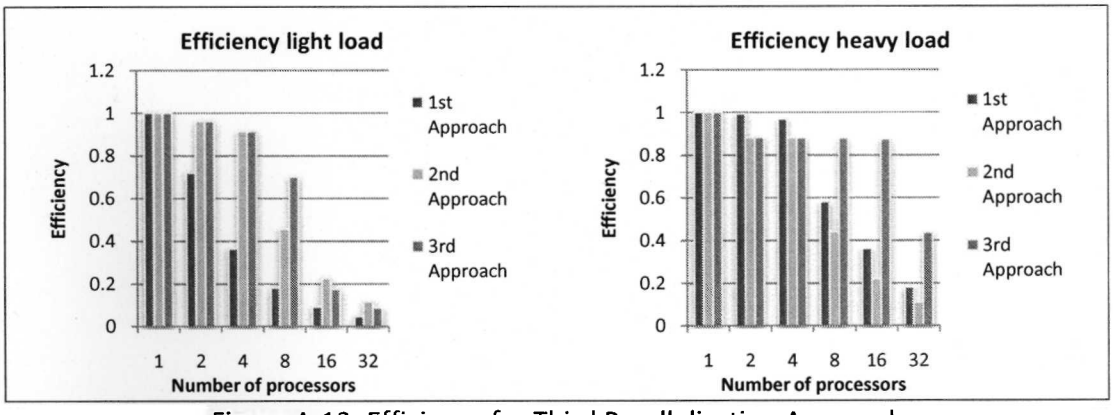


Figure A-13: Efficiency for Third Parallelization Approach

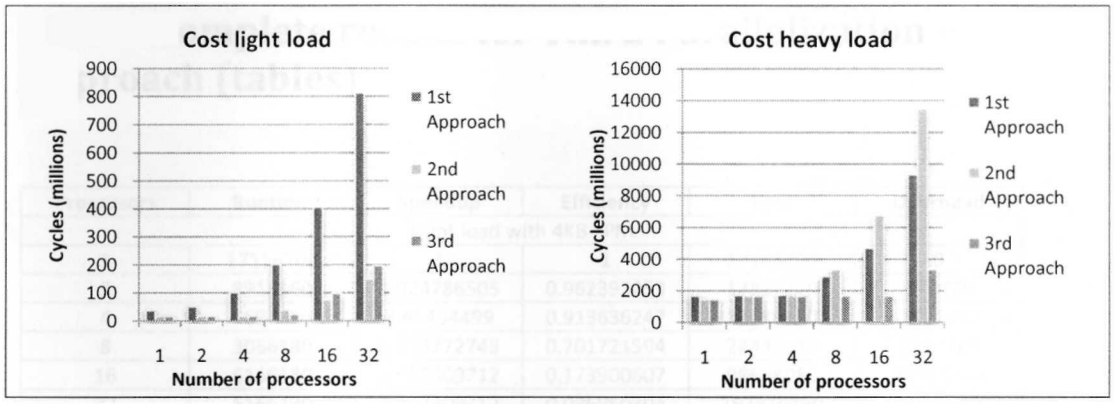


Figure A-14: Cost for Third Parallelization Approach

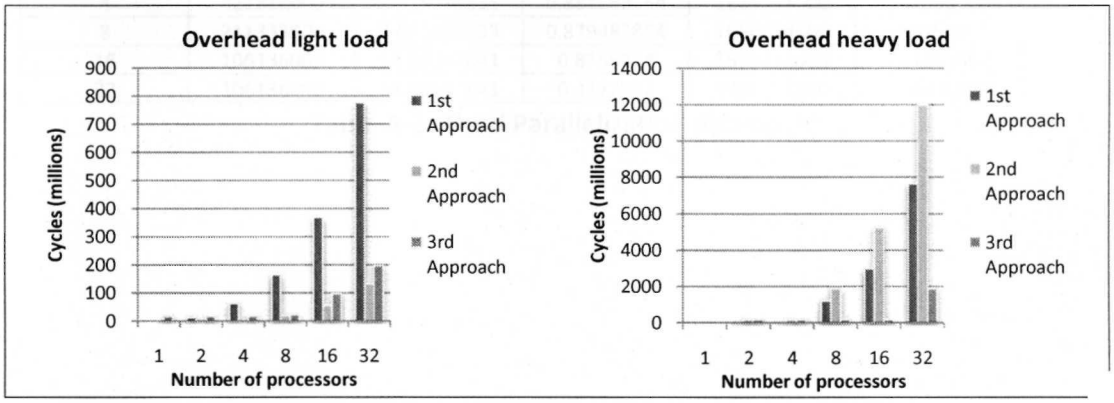


Figure A-15: Overhead for Third Parallelization Approach

A.13 Complete results for Third Parallelization Approach (tables)

Processors	Runtime	Speedup	Efficiency	Cost	Overhead
Light load with 4KB SPM					
1	17156700	1	1	17156700	0
2	8913560	1.924786505	0.962393253	17827120	670420
4	4694620	3.65454499	0.913636247	18778480	1621780
8	3056180	5.613772749	0.701721594	24449440	7292740
16	6166130	2.782409712	0.173900607	98658080	81501380
32	6166130	2.782409712	0.086950304	197316160	180159460
Heavy load with 4KB SPM					
1	1486910000	1	1	1486910000	0
2	842602000	1.764664693	0.882332347	1685204000	198294000
4	421812000	3.525053815	0.881263454	1687248000	200338000
8	211333000	7.035862833	0.879482854	1690664000	203754000
16	106136000	14.00947841	0.8755924	1698176000	211266000
32	106136000	14.00947841	0.4377962	3396352000	1909442000

Table A-3: Third Parallelization Approach

A.14 Image sequence

Figure A-16: First six images of sequence

Bibliography

[1] AMBA Open Specification, http://www.arm.com

[2] Ailawadi, D., Mohapatra, M. K., & Mittal, A. (2010). Frame-Based Parallelization of MPEG-4 on Compute Unified Device Architecture (CUDA). *IEEE* , 6.

[3] ARM. (1983). *Advanced RISC Machines*. Retrieved January 1, 2011, from ARM Web Site: http://www.arm.com/

[4] Baik, H., Sihn, K.-H., Kim, Y.-i., Bae, S., Han, N., & Song, H. J. (2007). Analysis and Parallelization of H.264 decoder on Cell Broadband Engine Architecture. *IEEE* , 5.

[5] Baker, M. A., Dalale, P., Chatha, K. S., & Vrudhula, S. B. (2009). A Scalable Parallel H.264 Decoder on the Cell Broadband Engine Architecture. *ACM* , 10.

[6] Bathen, L. A., Ahn, Y., & Dutt, N. D. (2010). Inter and Intra Kernel Reuse Analysis Driven Pipelining on Chip-Multiprocessors. *IEEE* , 4.

[7] Bathen, L. A., Ahn, Y., Dutt, N. D., & Pasricha, S. (2009). Inter-kernel Data Reuse and Pipelining on Chip-Multiprocessors fo Multimedia Apllications. *IEEE* , 10.

[8] Bathen, L. A., Dutt, N. D., & Pasricha, S. (2008). A Framework for Memory-aware Multimedia Application Mapping on Chip-Multiprocessors. *IEEE* , 6.

[9] Brey, B. B. (2001). *Los Microprocesadores Intel.* Mexico: Pearson Educacion de Mexico S.A. de C.V.

[10] Cho, D., Pasricha, S., Issenin, I., Dutt, N., Paek, Y., & Ko, S. (2008). Compiler Driven Data Layout Optimization for Regular/Irregular Array Access Patterns. *ACM* , 10.

[11] Cho, Y., Kim, S., Lee, J., & Shin, H. (2010). Parallelizing the H.264 Decoder on the Cell BE Architecture. *ACM* , 10.

[12] Fedora Project, http://fedoraproject.org/

[13] Flynn, M. J. (1995). *Computer Architecture.* London: Jones and Bartlett Publishers International.

[14] Gonzalez, J. A. (2009). *Profiling and analysis of irregular memory accesses of memory-intensive embedded programs.* Monterrey: ITESM.

[15] Han, K.-H., & Kim, J.-H. (2002). Quantum-Inspired Evolutionary Algorithm for a Class of Cominatorial Optimization. *IEEE, 10*.

[16] IBM. (1988). *International Business Machines*. Retrieved January 10, 2011, from IBM Web Site: http://www.ibm.com/us/en/

[17] Intel. (18 de July de 1968). *Intel Corporation*. Retrieved May 20, 2011, from Semiconductors: http://www.intel.com

[18] Kogge, P. (February de 2011). The tops in flops. *Spectrum* , 7.

[19] Kumar, V., Grama, A., Gupta, A., & Karypis, G. (1993). *Introduction to Parallel Computing.* Minneapolis: Addison Wesley.

[20] Liu, X.-p., Wang, E.-z., Zhen, L.-p., & Wei, X.-w. (2006). Study on Template for Parallel Computing in Visual Parallel Programming Platform. *IEEE* .

[21] Li, Y., Abousamra, A., Melhem, R., & Jones, A. K. (2010). Compiler-assisted Data Distribution for Chip Multiprocessors. *ACM* , 12.

[22] Madl, G., Pasricha, S., Zhu, Q., Bathen, L. A., & Dutt, N. (2006). Formal Performance Evaluation of AMBA-based System-on-Chip Desings. *ACM* , 10.

[23] MediaBench Consortium, http://euler.slu.edu/~fritts/mediabench/

[24] Patterson, D. A., & Hennessy, J. L. (2007). *Computer Organization and Design* . Burlington: Morgan Kaufmann.

[25] *Power Arquitecture Solution Portal.* (n.d.). Retrieved January 13, 2011, from Power.org: http://www.power.org/

[26] Richardson, I. E., & Richardson, I. E. (2003). *H.264 and MPEG-4 Video Compression: Video Coding for Next Generation Multimedia.* Scotland : Wiley.

[27] SimpleScalar Tool Set, http://www.simplescalar.com

[28] Wolf, W. (2005). *Computers as Components.* San Francisco: Elsevier Inc.

[29] Yang, L. T., & Guo, M. (2005). *High-Performance Computing: Paradigm and Infrastructure.* Wiley-Interscience.

[30] Yan, Z., Liu, L., & Ma, L. (2009). The Method of Parallel Optimizatin and Parallel Recognition Based on Data Dependence. *IEEE* , 5.

[31] Ye, X., & Li, P. (2010). On-the-Fly Runtime Adaptation for Efficient Excution of Parallel Multi-Algorithm Circuit Simulation. *IEEE* , 7.