VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

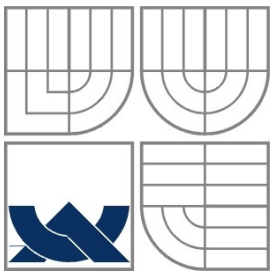
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PARALLELIZATION AND OPTIMIZATION OF IMAGE PROCESSING APPLICATIONS

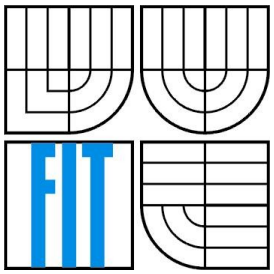
BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE **JAKUB ŠIŠKA**
AUTHOR

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PARALELIZACE A OPTIMALIZACE APLIKACÍ ZPRACUJÍCÍCH OBRAZ

PARALLELIZATION AND OPTIMIZATION OF IMAGE PROCESSING APPLICATIONS

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE — **JAKUB ŠIŠKA**
AUTHOR

VEDOUCÍ PRÁCE — **ING. EVA DOKLÁDALOVA PH.D.**
SUPERVISOR

BRNO 2009

Abstrakt

Tento dokument navrhuje řešení algoritmu pro urychlení zpracování obrazu a jeho úpravu pro použití s real-time video streamem z infračervené kamery. První část se zabývá vlastnostmi a základními principy IR technologie, následuje specifikace použité kamery. Následující text se zabývá problémy síťové komunikace s kamerou, charakteristiku formátu výstupního proudu a jeho vizualizace. Podstatná část této práce se zabývá paralelizací a optimalizací zpracování dat videa a obrazových souborů. Problém paralelizace je vysvětlen spolu s implementovanou metodou paralelizace pro tento případ. Celá teoretická část je podepřena reálnými výsledky a testy, které jsou uvedeny v poslední kapitole.

Klíčová slova

Infračervená kamera, infračervený, IR, kamera, pozorovaní, paralelizace, PThreads, pipe, roury, optimalizace, matematická morfologie, eroze, dilatace, OpenGL

Abstract

This paper proposes solution for speeding up image processing algorithm and its adoption for use with real-time video stream from the infra red camera. The first part discusses characteristics and basic principles of the IR technology, followed by specifications of used camera. Ongoing text also proposes solution of problems concerning network communication with the camera. In addition, it describes camera's output stream format characteristics and solution for output visualisation. Substantial part of this work covers issues concerning parallelization and optimization of video stream and image file data processing. Problem of the parallelisation for this case is explained together with implemented parallelization method. Entire theoretical part is supported with the real results, benchmarks, which are presented in the last chapter.

Keywords

Infra red camera, infra red, IR, camera, surveillance, parallelisation, PThreads, pipe, optimization, mathematical morphology, erosion, dilation, OpenGL

Citation

Šiška Jakub: Parallelization and Optimization of Image Processing applications. Brno, 2009, Bachelor's thesis, FIT BUT.

Parallelisation and Optimization of Image Processing Applications

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Evy Dokládalove Ph.D. a Doc. Dr. Ing. Jana Černockého.
Další informace mi poskytli Thierry Grandpierre, Petr Matas a Andrej Rypák
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jakub Šiška
20. 5. 2009

Poděkování

V Prvom rade by som sa chcel poďakovať vedúcim mojej bakalárskej práce. Na ESIEE to bola Ing. Eva Dokládalova Ph.D. a na VUT Doc. Dr. Ing. Jan Černocký. Moja vďaka patrí tiež Thierrymu Grandpierrovi, ktorý učí na ESIEE počítačové siete a bol mi nápomocný počas tvorby sieťového rozhrania s kamerou. Ďalej by som sa chcel poďakovať Petrovi Matasovi a Andrejovi Rypákovi, ktorý mi vždy radi zodpovedali moje otázky.

Thanks

In first place, I would like to thanks to my bachelor thesis supervisors. At ESIEE it was Ing Eva Dokládalova Ph.D. and at BUT Doc. Dr. Ing. Jan Černocký. I also appreciate Thierry Grandpierre's helpfulness in answering all of my questions about the networking. In addition I would like to thank to my colleagues Petr Matas and Andrej Rypák, who were always helpful to help me.

Table of Contents

1 Introduction

Computer vision and multimedia has become a part of everyone's lives. They are all around. Image processing is undoubtedly one of the most growing and evolving fields within a multimedia. Computer vision applications are usually demanding in a computational power, thus optimizing computer vision applications is essential for their use in real environment.

This thesis was evolved at ESIEE, Paris Computer Science Department. ESIEE is well known French engineering school with aim on electrical engineering. Computer Science Department's domain is image processing and dedicated architectures for image processing. Department mainly deals with a medical imaging and a video surveillance applications. For example, model applications done at this department are person detection, aggression detection at public places.

My task was to optimize given processing algorithm, implement it with a video stream from the infra red camera and display the output. Chain of these processes needed to be properly connected and optimized for general use, with the best results as possible. On figure [1.1] you can see simplifies scheme of the stream chain.

Further chapters present problems and solutions concerning the IR camera and its implementation with chosen graphic algorithm. In the first part, we will discuss characteristics and basic principles of an IR technology. Following chapters are dealing with problems of the network communication with camera in addition with the output stream format characteristic. The last and the most substantial part of this work covers the issues concerning parallelisation and optimization of the implementation. These chapters explain theoretical issues along with implementation burdens. End of this document contains benchmarks that quantify speedup of the proposed solution.

Figure [1.1], Depicted application's chain

2 Image Acquisition: IR Camera

This chapter is describing characteristics and basic principles of the infra red based imaging device. In our case we used IR camera FLIR ThermoVision™ A320, which is a general purpose IR camera [1].

Usage of infrared cameras allows us to see even in absolute dark (night vision) or survey surface's temperature (thermography) of scanned objects. Night vision and thermography works on different principles. We will strictly orient on thermography principles, because used camera is a thermograph.

Infra red cameras are mainly used in a military and in an industry. They are commonly used for area surveillance as they can easily detect an object emitting heat e.g. man, vehicle, fire. In industry, infra red cameras are mostly used for product quality checks or heat loss elimination.

2.1 Camera Specifications

We have used infra red camera, FLIR ThermoVision™ A320 made by Swedish manufacturer FLIR Systems, in our project. It is a general purpose camera with very a sophisticated firmware, which supports many transfer protocols, image formats and it can be fully used even from a regular web browser. Output example is shown on figure 2.1. Camera's big advantage is the passive cooling and the lightness, which makes it very portable with almost no restrictions in use.

The sensor resolution of the camera is 320×240 pixels. Camera has an autofocus and a motorized manual focus. It can measure temperature ranging from -20ºC up to +120ºC (+1200ºC in the special mode) with an accuracy ±2ºC. Camera is powered by a 12V power supply. For the output, camera has an RJ-45 connector, an analogue BNC connector and a connector for proprietary serial six vein cable.

Camera offers variety of output formats and many ways how to access those data. The most comfortable way to use camera is to stream output to the web browser. Camera provides GUI for the web browser, which is more than sufficient for a regular surface surveillance. For camera's implementation into our application we need to use a special protocols supported by camera. Data are transferred from the camera to the computer by trio of the network protocols: RTSP/RTP/UDP. By using these protocols, we could directly access camera's output. Camera provides several streams with two types of data formats. A common video format MPEG-4 and RAW format, providing uncompressed and raw data stream, best for video post processing.

2.2 Network Communication & Data Streaming

As it was mentioned in the section above, camera uses the RTSP/RTP/UDP protocols to communicate and transfer data to a computer or any other network device that supports such protocols. In our case, we needed to use camera's data stream for the image processing, so it was necessary to study and implement those protocol within our program.

2.2.1 RTSP

RTSP stands for Real Time Streaming Protocol, fully described in RFC 2326[2].

RTSP is the text based network protocol usually using port 554 and can be implemented over TCP or UDP. This protocol is very similar to HTTP protocol. It has a message header, which describes a request or a response action. This protocol uses response codes with the same number prefixes as HTTP, with the same meanings (1-info; 2-success; 3-redirection; 4, 5-error).

For our application, we had to establish a connection using this protocol. Establishing such connection takes several stages, during which client negotiates conditions of communication and data transfer specifications with server. First of all, a client demanding connection sends OPTIONS request asking a server for the list of all possible actions provided by server [Annex A]. Server sends back provided actions, generally including PLAY, PAUSE, TEARDOWN actions for stream regulation, and SETUP and DESCRIBE actions needed for a stream creation. Used infra red camera server supports all of those actions. In next step, client sends the DESCRIBE request, demanding server to send him, the list of all multimedia streams provided by the server with detailed specifications [Annex B]. For the response carrying specifications and listings of all streams, SDP (Session Description Protocol) [3] protocol is used. This protocol is briefly described in the end of this chapter. Camera provides many streams, regular MPEG-4 encoded stream in two resolutions and the RAW one, which is also provided in two resolutions. All information about those streams are included in the one server response preceded by the DESCRIBE request. Client, after receiving the response, chooses desired stream and according to received parameters sends the SETUP request to the server [Annex C]. The SETUP request is announcing to a server client's setup and requirements for a network transfer of the chosen stream. Client mainly negotiates listening ports, transfer protocol and type of cast address. In our case we used random ports, RTP/AVP transport protocols and unicast address type. Server's positive response mainly includes session number, which needs to be included in future clients' requests for identification purposes. In addition, response holds the client and server's ports, the transport protocol and the type of cast address. After this step, actual transfer setup is settled and data transfer may begin.

To start data a transfer, client sends the PLAY request to the server [Annex D]. Server responses and starts sending data stream on agreed ports. Data transfer uses UDP so there is no way for the server to determine if the client is still listening. Server stops transfer after certain time if there

is no hear from a client. To keep the connection alive, our client occasionally sends Receiver Report, using RTCP protocol, to the server. Such message does not interrupt transfer and has no other impact on server's behaviour. After streaming has begun, client receives data until it is required. During this time client may send the PAUSE or TEARDOWN request to a server. PAUSE request suspend a data transfer. After issuing the PLAY request again, stream will continue from place where it has stopped. In our application, there is no need for the stream pausing so we have not implemented this feature. Last but not least important request in sequence of stream handling is the TEARDOWN request [Annex E]. This request stops a stream, for good.

2.2.1.1 SDP

Session Description Protocol [3] describes the format for session parameters exchanged during a stream initialization. As we have mentioned above, server using RTSP sends, after the DESCRIBE request, list of all available streams with parameters using the SDP.

SDP defines many parameters but in our application we have used only very few of them. From the response it is essential to obtain a URI of the stream, a video resolution and a stream port issued by the server. URI and frame resolution are constant during the each program execution, so we have set constants in a header file, to bypass searching for these parameters in the SDP response during each run. Server's port parameter depends on each execution so we have to look for it each time in the DSP response.

2.2.2 RTP

RTP, a Transport Protocol for Real-Time Applications [4], is another protocol that we have come across while creating the client for streaming data from the infra red camera. This protocol provides informational and in some way control resources for communication between a sender and a receiver. Various extensions, of this protocol, specify data encapsulation within this protocol.

This protocol operates over UDP and consists of two parts; the header and the body. Protocol has big-endian bite ordering. Since protocol is using UDP, we could not guarantee quality of service. In our application we have used only one feature of this particular protocol. We used it, as was mentioned above, for keeping the connection alive by occasionally sending Receiver Report message.

This type of message is generally used for sending feedback to a sender, mainly describing quality of the communication (lost rate, jitter etc.). Protocol is fairly complex, but we have used only this one part from it, because we have not intended to build robust client for receiving multimedia streams. In this phase of building application our aim was to build bare client, strictly dedicated for networking with used infra red camera.

2.2.2.1 RTP Payload Format for Uncompressed Video

RTP Payload Format for Uncompressed Video [5] is a protocol that specifies data encapsulation in, before mentioned, the RTP protocol packets. In our application we have used only RAW stream from the camera, thus raw data were transferred according to this protocol. Each stream packet consists of the header and the body. Header has fixed size and carries important information about transferred data such as scan lines numbers, length, sequence number of packet ordering, timestamp and many others. In our case we have used only a line number field from the header, rest was dismissed. In the 15-bit line number field, protocol stores a number indicating order number for the scan line(s) transferred in packet. According to this number, we have stored line data into program data structure. This needed to be done because transfer was handled over UDP, so we could guarantee neither lossless data transfer nor right order of incoming packets.

In each packet, after the header, the body part is carrying a scan line data. In our stream from the camera, in each UDP packet, data of two scan lines are transferred. According to the specification of this protocol, there are compulsory header fields for the each transferred line within a packet. Three fields: length, line number and offset must be supplied in the packet header for the each line in the body. In our case we have two lines in the each packet so each of those fields is included twice. Despite of this fact, our program evaluates only the first line number field, because second is always consecutive. Offset fields are always constant with zero value. Length fields depend on the horizontal resolution and between packets are changeless.

2.3 Camera Output Format

As we mentioned before, camera encodes output into two different formats (MPEG-4, RAW). In our application we have used only the RAW format. In comparison to MPEG-4 it provides better quality, since used MPEG-4 is a loose format, In addition, RAW format provides data in 14-bit signal. Another MPEG-4's disadvantage is that it needs to be decoded before use. Decoding consumes system resources and this may slow down actual image processing.

Camera's sampling rate is 10 frames per second. In each UDP packet from the camera stream we receive two lines of uncompressed RAW 14-bit image data. One pixel is represented by 16-bit value despite of the fact that it carries only values in the 14-bit range. Infra red camera image is not represented in a standard RGB colour scheme but it uses the YUV colour space; the Y160 format. The Y160 format uses only Y component, others are omitted. We can say that Y160 format is simple high quality greyscale format, which is using 16-bit colour depth for monochrome colour. Our program uses the highest possible resolution of RAW stream, 320×240 pixels. During a second, up to 1200 UDP packets might be received, depending on the network bandwidth and the CPU load.

After receiving each UDP packet, program seeks for the line number in the header and then stores data of two scan lines into the structure on position according to obtained line number. Simple,

one dimensional array is used as a structure, and each array cell represents one pixel. As transmission is held over UDP, some packets may get lost. This solution is proof against such problems because new lines overwrite old ones, strictly according to the line number. They are not overwritten only in sequential order depending only on packet arrival time. If some packets get lost, algorithm uses scan lines from preceding scan, so on display no black fields appear. This approach is more user friendly, because if scene is static with very few dynamic objects, loss of few lines can by indistinguishable for a user.

Figure [2.1], Infra red camera's output to a web browser

3 Parallelized Algorithm

This chapter describes algorithm that we have parallelized in our application. Algorithm is proposed in [6] by Petr Dokládal and Eva Dokládlova. Eva Dokládalova has been my supervisor of bachelor thesis as well. In the time of closing deadline for this document, algorithm has not been published yet, so there may be slight difference in comparison to final published version.

3.1 Description

Used algorithm optimally implements dilation and an erosion with flat rectangles. The primary advantages of this algorithm are its low memory requirements, sequential data access and zero latency.

Algorithm is very similar to The Lemire algorithm [7] despite of the fact that it computes a dilation/erosion at the same time and its limitation is that the origin of the SE is always at the right-most end. This is the first algorithm that is capable of processing 2D data stream without latency and without an intermediate storage. These virtues are beneficial for efficient stream implementation of compound operators. As a consequence of these properties algorithm produces the output before entire input has been read.

3.2 Algorithm Principles

Algorithm works in a stream. It reads data from the input and pixel's index stores in the reading position counter. It does the same with writing pixel to output, stores its index in the writing position counter. This is done for the lines and for the columns as well. It uses four position counters. According to these counters algorithm compares and reads new values. Algorithm mainly compares values, it does not use addition or multiplication, thus uses much less system resources.

Dilation/erosion algorithm processes data in the three steps.

1. *Function encoding*: It drops unneeded data that do not take part in computations. Relevant data stores in the FIFO structure, in pairs (value, position)
2. *Look if there is a new value to propagate*: Compare currently propagated maximum with the fist sample from FIFO.
3. *Dilation/erosion by propagation of maxima*: Algorithm detects whether the current maximum has finished its propagation, by position comparison. It compares position of the current maximum plus span size of the structuring element on the left from the origin against the current writing position. If the propagation of the current maximum has ended, a new maximum is sought in the FIFO.

We can see basic illustration of this algorithm on figure [3.1]. Processed image has size of N×M pixels. Big rectangle represents structuring element (SE). Inside this box reading and writing position is marked. Reading position is always in very left down corner and writing position can be set anywhere inside the SE rectangle. SE rectangle must get around the whole image in order to process it. Box never moves backward as this algorithm is strictly sequential.

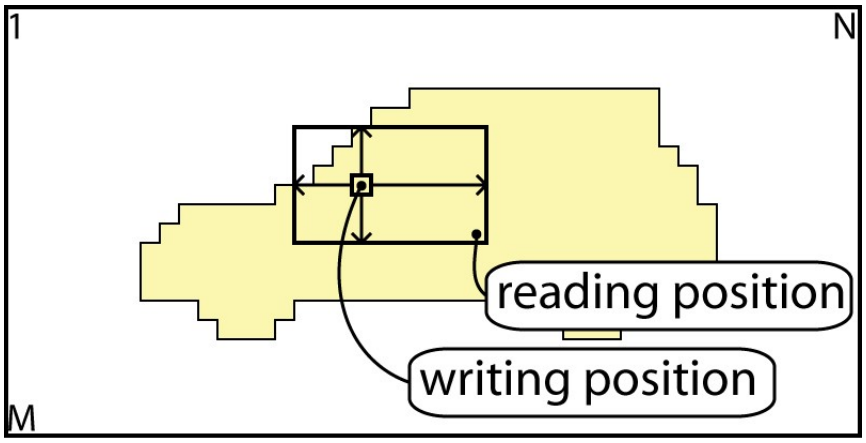


Figure [3.1], illustration of image processing algorithm

3.3 Dilatation and Erosion

Dilatation and erosion are very basic operators of mathematical morphology [8]. Figure the [3.2] below, show us the original 1D signal with comparison to the same signals processed by the dilation and the erosion. As we can see, original signal is "smoothed" by dilatation or erosion. Intensity of this "smoothness" effect depends on size of the structuring element. Difference between a dilatation and an erosion is that dilatation works with MAX values in given region and erosion with MIN values.

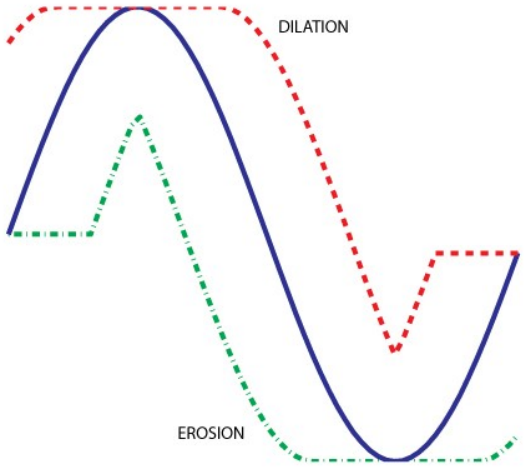


Figure [3.2], 1D signal of image data, normal (blue), dilation (red), erosion (green)

This phenomenon is illustratively showed on figures: [3.3], [3.4] and [3.5]. We can see the original Van Gogh's painting (The Church at Auvers) transferred into greyscale on the figure [3.3]. On the figure [3.4], the original picture is transformed using closing filter and on the figure [3.5] transformed using opening filter. Both pictures, on figures [3.4] and [3.5], are processed by our application; it is actual application's output. Figures [3.6] and [3.7] show unfiltered and filtered stream output from the infra red camera.

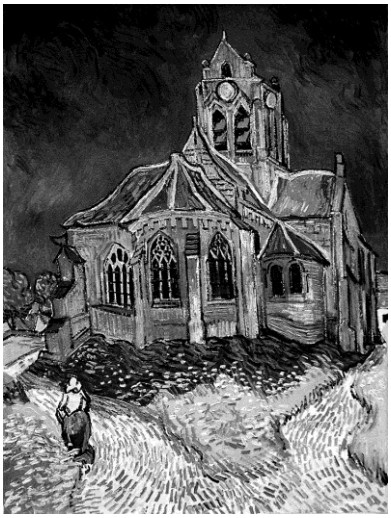


Figure [3.3], Original Van Gogh's painting

Figure [3.4], Closing filter applied on original [3.3]

Figure [3.5], Opening filter applied on original [3.3]

Figure [3.6], Unfiltered output from the infra red camera

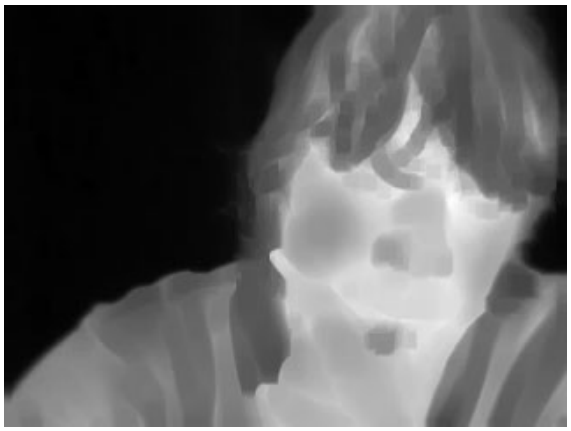


Figure [3.7], Filtered output from the infra red camera

3.4 Alternative Sequential Filters

Dilation and erosion functions are usually combined together. Basic combination of dilatation and erosion produces opening and closing filters. Each opening and closing filter consists of one dilation

and one erosion. In opening filter, firstly dilation is applied and then erosion. In closing filter, it is vice versa.

Alternative Sequential Filters (ASF) were originally proposed by [9] and studied in [10]. These filters are mostly created by combining opening and closing filters. ASF filters are commonly used in the mathematical morphology.

3.5 Algorithm Latency

There are two types of latency, the operator latency and the algorithm latency. Operator latency is a delay caused by non causal filters. Algorithm latency is time that algorithm needs for creating result after source data are available. Sum of these two latencies is final delay between the input and he output availability. Algorithm with a zero-latency produces an output only with latency of the operator.

This algorithm has zero-latency. It is the first of the kind that has such attribute. It is the first algorithm that implements dilation and erosion in such way. It starts writing output even before all source data have been completely read. As stated before, delay between output and input is strictly operator latency, which depends on size of the structuring element.

3.6 Algorithm Memory Requirements

In the real-time algorithms, memory usage is very important attribute. This algorithm tries to be as much memory efficient as possible. It does not need any intermediate storage between the stages and memory requirements are only within each processing stage.

In the worst case, algorithm's memory usage is N memory blocks of size 2L (vertical part) plus 1 memory block of size 2K (horizontal part). Where N is a width and M is a height of processing image. K, L represents width and height of applied structuring element. Algorithm that processes a image with dimensions 800×600 pixels and uses a square shaped structuring element with size of 20 pixels will not use more than 32,040 (800·40+1·40) bytes of memory. In the previous example, 1 pixel is encoded in 1 byte of memory. Note that no input or output data needs to be stored in memory, so this algorithm is considered to be very memory efficient and memory independent. Such memory efficiency is mainly achieved by the fact that algorithm processes data in sequential order with no repeated access to previous pixels. Only pixels between the reading and writing position need to be stored.

Previous example of the memory usage is computed only for one pass of dilatation/erosion. In the case with more passes, memory demand will be equal to sum of memory usage of each pass. For each pass applies formula presented in the previous paragraph. Total memory usage consist only of

this sum, because there is no need in the intermediate data storage between the stages and input or output does not need to be stored in a memory neither.

4 Parallelisation

Nowadays most of the new computers embrace multi-core processor that is capable of running more applications simultaneously or running one application on more cores at once. Parallelising applications has became more important than whenever before. Parallelisation is used especially for accelerating computing algorithms, which are used in real-time applications. Not all algorithms are parallelisable and almost all of them can not be parallelised fully. Before actual parallelisation of the algorithm, we need to determine independent parts that can run simultaneously and critical sections that can not be parallelized. Depending on this study, parallel programming model is evolved and application is designed.

In our case we needed to parallelize the image processing algorithm for a still image and for a video stream. Our goal was to speed up serial algorithm using parallelization and common patterns, to reach real-time processing.

Parallelization can be done in several ways, but the most common way is to use threads or processes. Processes are more complex structures, they own one's memory space, file handlers, sockets etc. If two processes need to share data, those data must be copied between the processes' allocated memories. Threads do not suffer from this problem, because threads within same process share memory space between each other, so exchanging data is faster and easier. For our implementation we have chosen threads because of the mentioned advantages [11].

If we talk about parallelization in any way or on whichever level of abstraction we always need to handle hazards with a synchronisation. Each API implementation approaches to these problems differently. In Shared Memory chapter we will explain approach of chosen APIs.

4.1 Approach in Segmentation For Parallelisation

As was mentioned in chapter before, given algorithm is processing image in a stream. It does not need full frame to start processing. Our application starts actual processing after enough pixels are available (number depends only on SE) in stream. This fact means that simple division of a frame on n-parts can not be applied because image processing would be excessively delaying output stream. Only the frame, which has already all pixels available, could be processed by using this approach.

Another approach to parallelisation is to parallelise frame according to the received lines. Each stream packet supplies two lines of scan. We can create threads that would simultaneously process two different pairs of scan line. Since processing algorithm may require data from lines before those two processing, in one batch, many critical sections and more memory would be needed to handle

such synchronization requests. This would lead to decreased performance in bigger SEs. This problem can be suppressed be stacking more lines into one batch. Certainly it would speed up processing but it would grow up memory requirements and stream delay would be higher as well, all depending on number of stacked lines.

Delay would be smallest if we do not need to wait for any other lines and start processing after receiving first pixel of the frame. This approach denies use of any solution proposed above. Final speedup, of whole application, is what counts in the end. As mentioned in chapter before, dilatation and erosion is usually applied several times on a single image. We propose solution that does not divide each dilatation/erosion filtering into several parts and run them simultaneously but paralyze the whole sequence of dilatations and erosions in a way, that each dilatation/erosion would be one parallelized element; thread. This approach eliminates problems with complex synchronization because each dilatation/erosion is computed serially. Another advantage of this solution is its minimal latency. Only one disadvantage of this solution is its minimal speedup if number of erosions/dilatations applied on the image is smaller than number of available computing cores. For example, if only one dilatation is applied on the stream, there is no speedup at all because there is no parallelized part. However this problem is not critical, because it is very uncommon to use such low count of dilatation/erosion passes.

4.2 Old vs. New Parallelisation

One of our goals was to improve former parallelization, which was done using the UNIX pipes. Since we have been developing a new parallelization for Windows based systems, former UNIX pipes were rewritten to Windows' ones using WIN32 API.

4.2.1 Pipes

Original parallelization used UNIX pipes and was implemented for Linux. Each execution was done through shell script because parallelisation consisted of numerous executions of same program with different parameters. Program itself could done only one dilatation/erosion thus more programs running simultaneously created parallel execution of given sequence of erosions/dilatations. Shell script also ensured that the programs could communicate between each other using pipes. Simple pipe character ("|") was used. Pipes use pipe files to store data exchanged and those files behaves as FIFOs.

Parallelisation based upon pipes is not the most efficient one, because lot of the processor time is wasted on handling (read, write operations) pipes. Those operations ale slow, because pipes use files to store data. Input/output operations on files are always slower than operations within the shared memory stored in RAM. Another factor that slows pipe operations is the fact that all pipe operations

are handled by the operating system and program needs to wait while they are executed by the operating system.

The only advantage of piped solution is its fast and simple implementation. Developer does not need to implement parallelization inside the program.

4.2.2 Shared Memory

Usage of the shared memory access in parallelized application is the fastest way for the exchanging of data among parallelized elements. In our application we have chosen this way of exchanging data because we wanted to obtain the best results in speedup.

Before actual implementation we had to decide, which parallelisation API to use in our application. There are many APIs' implementations to use. The most used and supported are: Intel® TBB (Intel Threading Building Blocks), PThreads (POSIX Threads), OpenMP (Open Multi-Processing), MPI (Message Passing Interface), UPC (Unified Parallel C). In the final stage, before actual programme coding, we have been choosing among Intel TBB and PThreads. Next two sub sections briefly describe each of these APIs.

4.2.2.1 Intel® Threading Building Blocks

Intel® Threading Building Blocks is mainly developed and maintained by Intel® Corporation. For the long time it used to be proprietary solution for parallelisation. Nowadays it is free to use and entirely open source.

Intel's approach to thread creations is to decouple programmer from targeted machines as much as possible. TBB itself handles number of the threads running simultaneously depending on number of cores and their overall load. Programmer only specifies parts of the program that may be parallelised and critical sections for data synchronisation. Developer should not handle individual behaviour of each thread by himself because this is done by TBB. Application developed using TBB arbitrarily decides how many threads to create and when rearrange them, during the run. Simply TBB is very useful for programmers developing multi-threaded applications for broad range of PC systems.

4.2.2.2 POSIX Threads

POSIX Threads is standardized library for UNIX type operating systems. Fortunately, version for Windows is available too. This C language programming interface provides functions for the parallelised programming. In contrast to TBB, in PThreads programmer must specify the thread's creation point and must decide by his own, how many threads to create.

Thread creation is appointed by code and on different machines program will create exactly the same amount of threads with same properties. Function used for creating new threads is `pthreads_create()`.

After the thread creation, programmer must be able to control threads. For this purpose PThreads' provides certain ways how to lock or even put to sleep the thread. Locking is important in cases where two different threads are accessing shred data. Locking is preventing rewriting variable by one thread so other does not produce incorrect results. `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions serve this purpose. Another option for programmer is to put to sleep one thread, if it is waiting for some condition. When the condition occurs, thread is waked up by another thread and continues in code execution. In this case programmer uses `pthread_cond_wait()` function for putting thread to sleep and one of `pthread_cond_signal()`, `pthread_cond_broadcast()` functions for waking up. PThreads also provide functions for thread scheduling but we have not used them in our application, because we did not need to.

4.2.2.3 Implementation

F our implementation, we have chosen shared memory model devised with PThreads [12] [13] [14] [15] [16]. Application has many options and user can find specification in attached program manual. Here we will only explain the implemented parts and their main functionality. Application processes images using dilatation/erosion. User can select source of those images, file or IR camera stream. Depending on this, different threads are executed. Output is always displayed on the screen using OpenGL.

In a beginning, we must propose base idea of our application. Application during the execution creates numerous threads depending on parameters. Thread creation is done in `main()` function. Here, program creates threads for the input, output handling and threads for image processing. Input thread takes care of reading data either from the file or from the socket. Output thread contains the OpenGL initialisation, functions for displaying and loop; `glutMainLoop()`. Each instance of the image processing thread embody single image processing algorithm, either dilatation or erosion.

File & Stream Thread

Input thread handles primary data sources. Our application can process data either from the file input or from FLIR ThermoVision™ A320 camera stream. Images from file must be in Portable Grey Map [17] and properties of stream were described earlier in this document.

Still image is generally read form the file on the local hard drive, saved in Portable Grey Map format. File is read into static array at once. Array has the exact amount of elements as image pixels. This array is data source for first processing thread.

If we use the camera stream as a data source, application must at first establish connection with the camera, initialise stream and after that actual streaming may happen. Process of the creation and administration of the stream is described in the second chapter. Streaming is real-time, thus data from it must be stored all the time with no regard on capability of data processing. Streaming uses two

buffers for image data storage, one buffer can store only one frame. Application needs two buffers to ensure that no data from the stream are lost. Processing thread starts reading of a new frame from buffer that is not filled yet, but for writing of the next scan, second buffer will be used so that data read by processing thread are not overwrote. Stream is received till program is not halted.

OpenGL Thread

OpenGL thread handles output of the processed images. It contains usual OpenGL initialisation with call-back on display function. In our application only 2D images are displayed. OpenGL is optimised for this purpose thus uses only 2D functions. For best performance of this thread many unused OpenGL features are disabled during the initialisation. After the initialisation, thread runs in OpenGL loop and calls-back registered functions. OpenGL thread runs until the application is closed.

Display function handles calculation and projection of each image. It is called in certain time intervals and each time it sends actual image data to draw buffer. If new frame is processed and ready to be drawn, draw flag is set by the processing thread. Display function on each run checks for this flag and when it is set, rewrites old data in draw buffer with new ones. This applies for the both, still image processing and stream processing.

Only difference is in handling contrast of new data. For still image no contrast adjustments are done. However stream must be adjusted because of the low contrast ratio. Pixels have values in 14-bit range (0~16383). In this range whole camera's temperature range (-20ºC ~ +120ºC) is stored. Contrast is poor because usually, when recording on camera, we do not have such temperature diversification. In a room crowded with people there is usually temperature ranging from 25ºC to 38ºC, which is 13ºC span. Difference of the 13ºC can be stored in very few bites and unadjusted picture would have very low contrast ratio. Picture is adjusted in a way that temperature range, in our example 13ºC, is spread across entire variable range. Thus in displayed window there are gray colours ranging from pure black to pure white pixels.

Image Processing Thread

Number of created image processing threads is equal to sum of dilatations and erosions applied on image. Each image processing thread applies only one pass of dilatation or erosion.

Usually, set of dilatations and erosions are applied on a single image thus image processing threads need to exchange data between each other. Thread reads and write processed pixels into double ended queue. If the thread has no pixel to read, it puts itself to sleep using `pthread_cond_wait()`. After preceding thread, in processing sequence, writes new pixel to shared variable, thread is waked up using `pthread_cond_signal()`. Then it reads new pixel and eventually writes processed pixel into variable shared with the following processing thread. This way of data synchronisation is not limited in number of threads.

To ensure that no two different threads write and read simultaneously from one variable, PThread locking must be used. Usage of locks provides exclusive access to locked variable. Thread that locks variable can safely use variable without fear of corrupting accessed data. If variable is locked by one thread and another one wants to access the same variable, second process will have to wait until previous one is done and unlocks the variable. Extensive variable locking or demanding operations upon locked variables may lead to overall slowdown of the application because individual threads spend more time waiting for variables than processing data. So it is important to lock only shared variables and only when it is necessary for the shortest time as possible. In our application each read/write action upon the shared variable is protected. `pthread_mutex_lock()` function is used for locking and `pthread_mutex_unlock()` function is used for unlocking the shared variable. To reduce problems with threads waiting for variable, only one read and one write operation is done during one processing circle and only simple pop and push operations are used to minimize time spent by thread in the critical section. Usually value of shared variable is copied into local variable and subsequently shared variable is unlocked and local variable is processed [Annex F].

Thread that is first or last in processing sequence evidently does not have image processing thread to read or write from. These threads, besides processing, are handling also input and output actions. The first thread reads data from buffers handled by file or stream thread. If stream is used as a source, the first processing thread needs to decide from which buffer to read. Thread always chooses the buffer that does not contains all image pixels yet, thus contains most recent image data. The last thread in processing sequence writes pixels into buffer dedicated for synchronisation with the display function. When display buffer contains complete frame data, flag for drawing is set and the OpenGL thread display new image to the screen [Annex G].

On figure [4.1], bellow, there is simplified example of the implementation. We can see input thread followed by chain of the processing threads and on the end the output thread. From the scheme it is evident that each thread is dependent on the previous one. Dependency is not complete but partial. In order to start processing data in one thread, preceding one must provide some data. The parallelisation is achieved when processing threads can run simultaneously. In our case it is, when one thread provides data to another before ending processing the whole image. This can be resembled to numerous hydroelectric power plants on one river. Power plant on lower part of the river can produce electricity after it is filled with some water from the upper hydro. On the end and in the beginning of the chain there are treads that provides interfaces with the input (stream, file) and the output (display) resources.

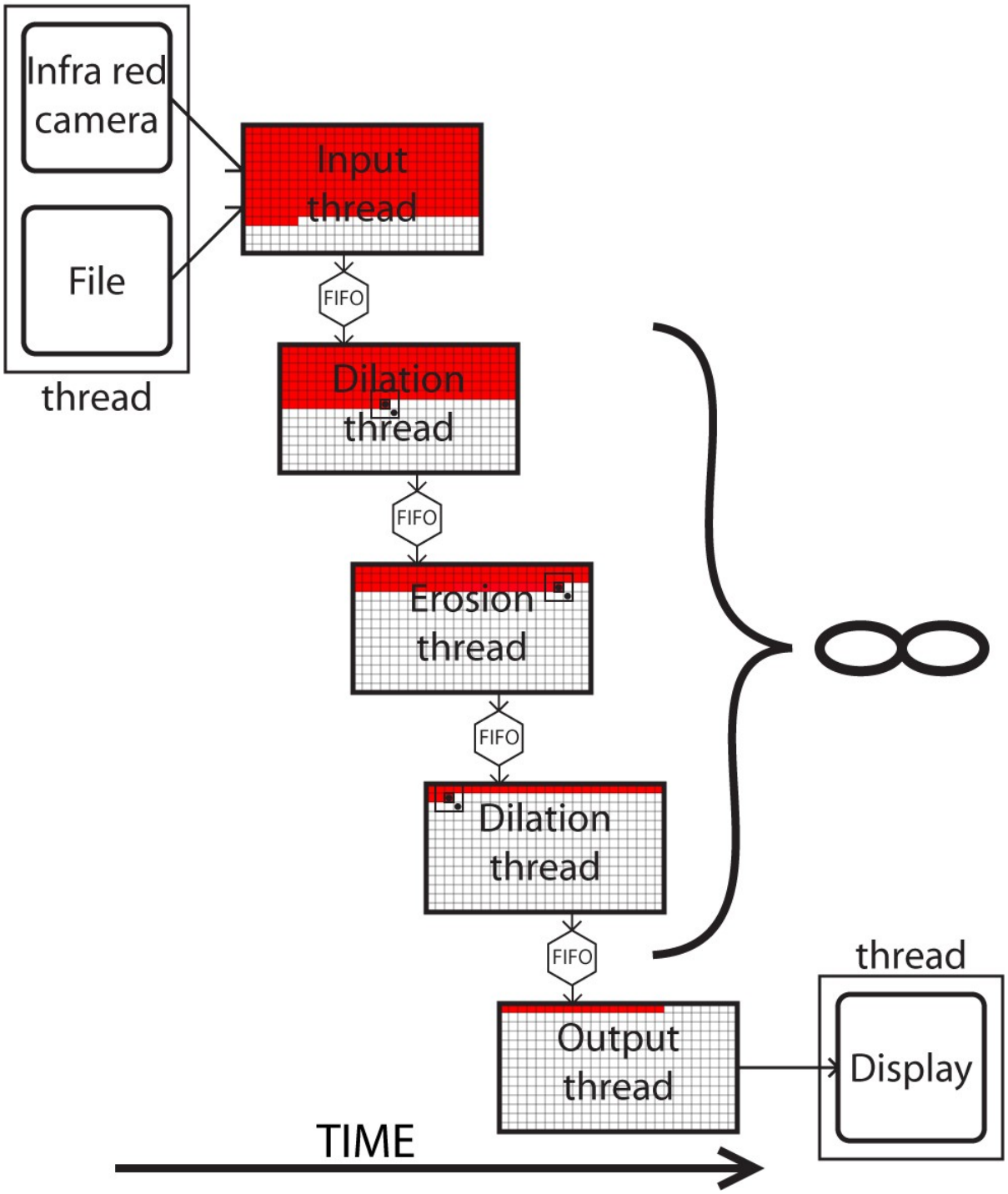


Figure [4.1], Illustration of implementation

4.3 Experiments & Measurements

This chapter presents benchmarks that were made on our application. Benchmarks show us very valuable information concerning the speed and the efficiency of the application. Measurements were done in few ways to obtain complex review of application's behaviour with various parameters.

In following charts we present performance comparison of two different principles of data synchronization. Performance of the former implementation based on pipes is compared against the new implementation, which is using shared memory. Three parameters were taken into consideration: image size, size of the structuring element and number of applied filters. We have measured run time of the application with different values of these three parameters.

Benchmarking was done on three machines. First was my personal notebook, second was notebook belonging to Computer Science Department at ESIEE and third machine was ESIEE's Opteron server.

My notebook is Toshiba Satellite M45–S331. This notebook is equipped with Intel® Pentium® M 730 [18] processor with core speed of 1.6 GHz. This processor supports Enhanced Intel® Speedstep Technology [20], but it was disabled during the benchmarking. In addition laptop has 1536 Mbytes of RAM and Toshiba 4200 rpm hard drive. Microsoft Windows XP Home Edition 32-bit is used as operating system.

Second device is notebook HP Compaq 8510w Mobile Workstation. Notebook is equipped with Intel® Core™2 Duo T9300 processor running at frequency of 2.5 GHz [19], 4096MBytes of RAM in two equal modules and HDD Seagate Momentus with 200Gbyte, 7200rpm and 16 Mbytes cache. Installed operating system was Microsoft Windows Vista Business Edition (Build 6000). This system is not ideal for testing purposes and benchmarks can be misrepresented because of Intel® Core™2 Duo T9300 processor. This processor supports Enhanced Intel® Speedstep Technology [20] and could not be disabled. This feature enables processor to be more energy efficient by dynamically changing its clock speed and voltage depending on processor's load. If processor is idle clock speed is much lower than during the load. This phenomenon can possibly affect benchmark results, especially application's runs with small run times. In my opinion impact of this feature is insignificant, it does not alter results in more than few percent span.

Third system is based on AMD's Operon processor, exactly on AMD Opteron 285 [21]. This processor has 4 cores and it is the most powerful processor in our benchmark. It also has biggest memory, 16384 Mbytes. On this system Microsoft Windows Server 2003 Standard Edition 64-bit is installed as operating system.

All presented values are average values of 10 measured runs. All measured values are presented in milliseconds, thousand milliseconds is equal to one second.

4.3.1 Image Size Dependency

In this part we will present application speed depending on image size. In figure [4.1] we present measured values for different image sizes, while size of the structuring element and the number of openings filter passes are constant. In benchmark following image sizes, in pixels, are used: 800×600, 1024×768, 1600×1200, 2400×1800, 3200×2400 and 4000×3000. Structuring element with size of 3 pixels is used and 2 opening filters are applied.

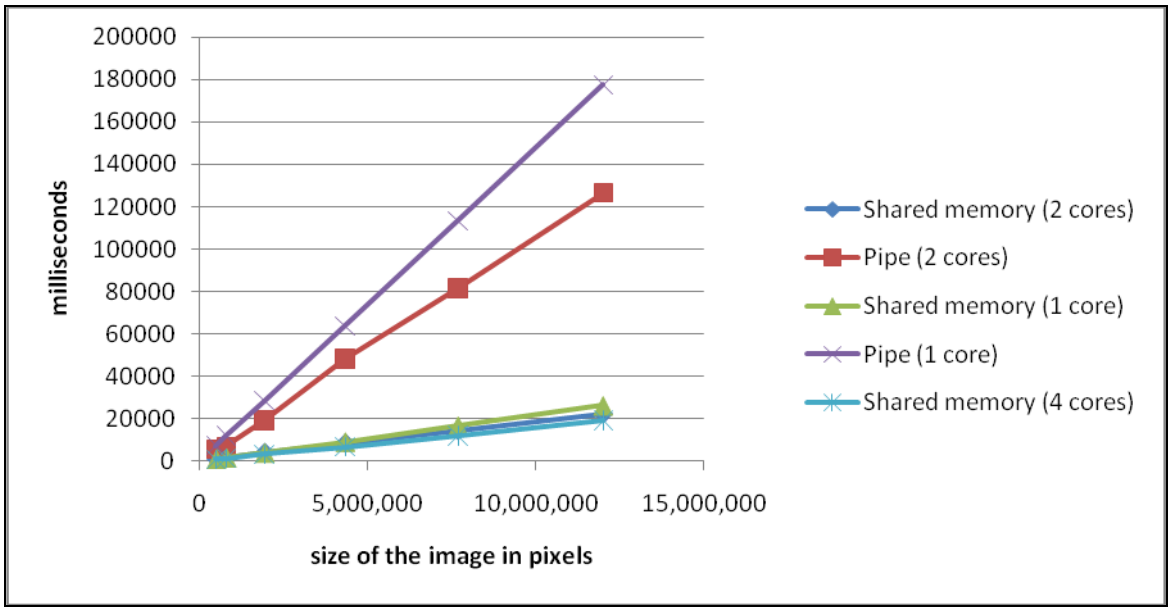


Figure [4.1]

From the graph is evident that execution time has linear function and with the rising image size, function is rising. We can say that running time is linear proportion of processed pixels. This applies for both, shared memory and pipe synchronization. Only difference is in slope of the linearity. Pipe has much bigger slope so with rising pixel count execution time is rising faster than in shared memory case. This is because the pipe version spends proportionally more time in thread handling than the shared memory version. The average speed up in this case is around 7 times.

4.3.2 Structuring Element Size Dependency

In this part we will display application speed depending on size of the structuring element. In figure [4.2] we present measured values for the different sizes of the structuring element, while the image size and the number of openings filter passes are constant. In benchmark following sizes of the structuring element, in pixels, are used: 3, 10, 20, 30, 50, and 80. The image size is 800×600 and 2 opening filters are applied.

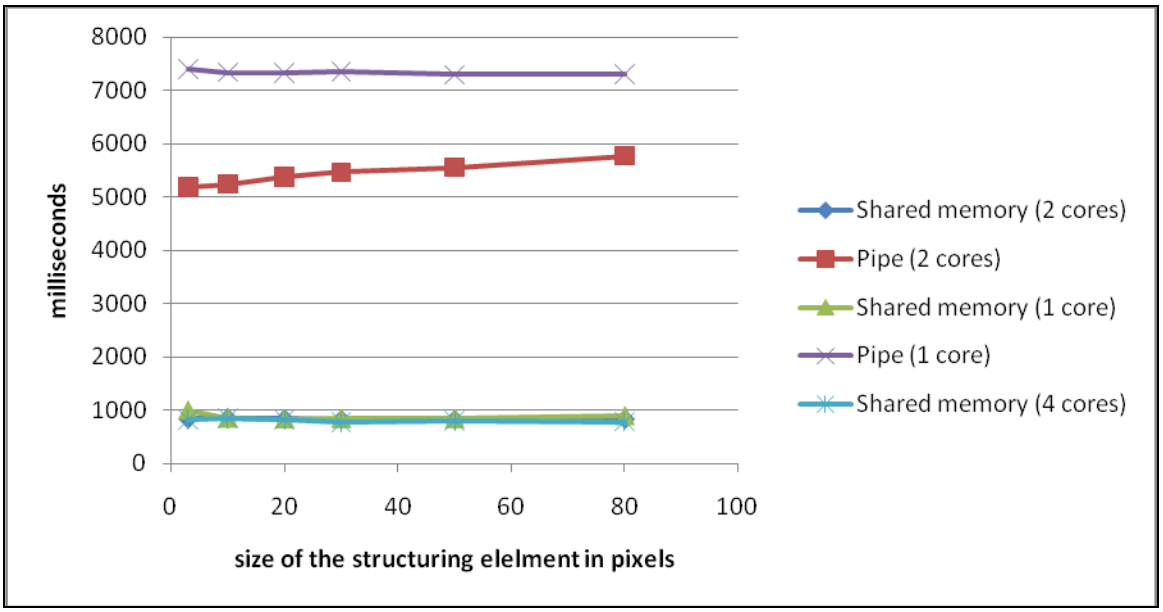


Figure [4.2]

Graph shows us that execution time has constant function, thus with rising size of the structuring element, execution time is the same. Conclusion from this benchmark is that, that execution time is not dependent on the size of the structuring element. Pipe is again slower in this case, because of mentioned synchronisation troubles. The average speed up in this case is around 7 times.

4.3.3 Used Filters Dependency

In this part we will display application speed depending on number of the applied filters. In figure [4.3] we present measured values for different count of used filters, while image size and size of the structuring element is constant. In benchmark following counts of used filter are applied: 2, 4, 6, 8, 10 and 12. Opening filter (dilation + erosion) is always used in this benchmark. The image size is 800×600 and the structuring element with size of 3 pixels is used.

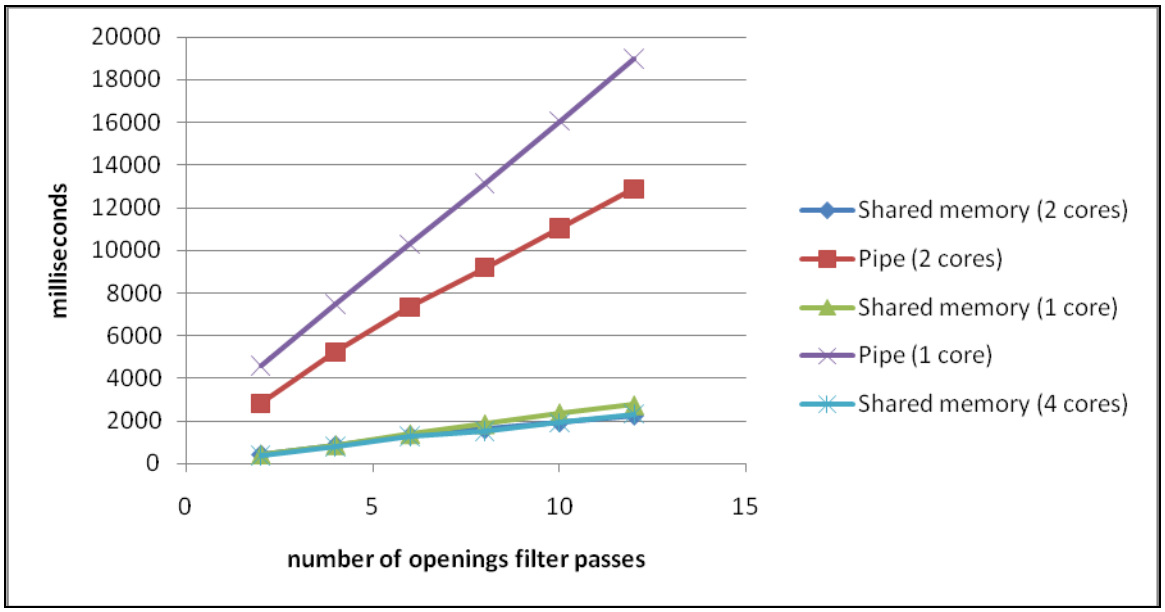


Figure [4.3]

From graph we can see that execution time has linear function and with the rising number of the applied filters is rising. It is clear and it was expected that with the rising number of applied filters execution time is prolonging. The application's running time is clearly dependent on the number of applied filters on processing image. As in the first benchmark, application that uses synchronization via pipes is slower and its function has bigger slope, nevertheless cause stays the same. The average speed up in this case is around 7 times.

5 Conclusion

The main task of my bachelor thesis was to setup processing chain, which in final stage contained: infra red camera, parallelised processing algorithm and real-time display algorithm.

Provided FLIR ThermoVision™ A320 infra red camera was brand new in time of my arrival and nobody has ever worked with such device before. Because of this my first task was to find the way how to communicate and acquire data from camera. This task was struggle for me because of complexity used protocols. Despite of those facts after weeks of work I was able to communicate and stream data from camera and I could display image from camera. Asset for me from this part is better ability of gaining and processing information concerning one topic that I am focusing on. This is result of solo questing of foal. In this part there was nobody who could help me with streaming protocols because no one has worked with such matter.

Another goal was to provide meaningful display for user. For this I have chosen OpenGL because I have worked with it before. I have utilised my earlier knowledge and I supplemented it with OpenGL optimization for given purpose. This time it was, 2D image rendering. I especially appreciate part because I am really interested in OpenGL and processor time saved by OpenGL optimization speeded up image processing.

Last and main task was parallelization of image processing algorithm. I was very glad that I can work on such matter and there was not time when I would regret it. I have never really parallelised any algorithm so beginning was all about reading. From this I learned a lot and without it I would hardly parallelised something. After the theoretical part I have developed and implemented parallelisation of given problem. I believe, I did best what I could, with me knowledge and skill. If, it is good enough you have to decide for yourself. I think that presented benchmarks are sufficiently persuasive. I believe that my implementation of given application noticeable improved run speed and usage of system resources. According to benchmarks, application runs around seven times faster than before, what I find as a great success.

Computer Science Department at ESIEE Paris deals with surveillance and object recognition. Infrared camera can improve recognition accuracy especially in bad light conditions. I believe that my camera interface helps to implement infra red camera to other applications. In time writing of this report, my camera interface was already used in one other application. In case of mathematical morphology I see possibility of improving and accelerating processing algorithms, because in this field direct implementation of operations such as dilation or erosion requires enormous computational power.

During designing and implementing this application I have always used way that was best from my point of view. Now when all implementation is done and application is working, I see that it may be better if I would do some things little bit differently. For example not to underestimate theoretical

part of development or ensure that old code is perfectly working before adding new one. It is not possible to absolutely eliminate these phenomena but it is essential to suppress them as much as possible. By doing so, development of application is faster and less stressing.

6 Annex

Annex contains various source code examples, on which document references. Annexes A through E are from file “network.cpp”. Annexes F and G are from file “processing.cpp”.

6.1 Annex A

```
101     sendbuf.append("OPTIONS * RTSP/1.0\r\n");
102     sendbuf.append("CSeq: ");
103     r=sprintf_s(tmp_buf, "%d", seq);
104     sendbuf.append(tmp_buf, r);
105     sendbuf.append("\r\n\r\n");
106     seq++;
107     // Send an initial buffer
108     r = send( ConnectTCP, sendbuf.c_str(), (int)sendbuf.length(), 0 );
```

6.2 Annex B

```
139     sendbuf.append("DESCRIBE rtsp://");
140     sendbuf.append(source);
141     sendbuf.append(" RTSP/1.0\r\n");
142     sendbuf.append("CSeq: ");
143     r=sprintf_s(tmp_buf, "%d", seq);
144     sendbuf.append(tmp_buf, r);
145     sendbuf.append("\r\n");
146     sendbuf.append("Accept: application/sdp\r\n\r\n");
147     seq++;
148     // Send an initial buffer
149     r = send( ConnectTCP, sendbuf.c_str(), (int)sendbuf.length(), 0 );
```

6.3 Annex C

```
182     sendbuf.append("SETUP rtsp://");
183     sendbuf.append(source);
184     sendbuf.append("/sid=");
185     sendbuf.append(SID);
186     sendbuf.append(" RTSP/1.0\r\n");
187     sendbuf.append("CSeq: ");
188     r=sprintf_s(tmp_buf, "%d", seq);
189     sendbuf.append(tmp_buf, r);
190     sendbuf.append("\r\n");
191     sendbuf.append("Transport: RTP/AVP;unicast;client_port=");
192     r=sprintf_s(tmp_buf, "%d", DEFAULT_PORT_UDP_C);
193     //r=sprintf_s(tmp_buf, "%d", atoi(stream_port.c_str()));
194     sendbuf.append(tmp_buf, r);
195     sendbuf.append("-");
196     r=sprintf_s(tmp_buf, "%d", DEFAULT_PORT_UDP_C+1);
197     //r=sprintf_s(tmp_buf, "%d", atoi(stream_port.c_str())+1);
198     sendbuf.append(tmp_buf, r);
199     sendbuf.append("\r\n\r\n");
200     seq++;
201     // Send an initial buffer
202     r = send( ConnectTCP, sendbuf.c_str(), (int)sendbuf.length(), 0 );
```

6.4 Annex D

```
237     sendbuf.append("PLAY rtsp://");
```

```
238     sendbuf.append(source);
239     sendbuf.append("/sid=");
240     sendbuf.append(SID);
241     sendbuf.append(" RTSP/1.0\r\n");
242     sendbuf.append("CSeq: ");
243     r=sprintf_s(tmp_buf, "%d", seq);
244     sendbuf.append(tmp_buf, r);
245     sendbuf.append("\r\nSession: ");
246     sendbuf.append(session);
247     sendbuf.append("\r\nUser-Agent: fsrtsp\r\n\r\n");
248     seq++;
249     // Send an initial buffer
250     r = send(ConnectTCP, sendbuf.c_str(), (int)sendbuf.length(), 0);
```

6.5 Annex E

```
480     sendbuf.append("TEARDOWN rtsp://");
481     sendbuf.append(source);
482     sendbuf.append("/sid=");
483     sendbuf.append(SID);
484     sendbuf.append(" RTSP/1.0\r\n");
485     sendbuf.append("CSeq: ");
486     r=sprintf_s(tmp_buf, "%d", seq);
487     sendbuf.append(tmp_buf, r);
488     sendbuf.append("\r\nSession: ");
489     sendbuf.append(session);
490     sendbuf.append("\r\nUser-Agent: fsrtsp\r\n\r\n");
491     seq++;
492     // Send an initial buffer
493     r = send(ConnectTCP, sendbuf.c_str(), (int)sendbuf.length(), 0);
```

6.6 Annex F

```
190     pthread_mutex_lock(&mutex_proc[index_proc-1]);
191     //if(!buffer_proc[index_proc-1].empty()){ //divne
192             //printf("read(%d)\n", index_proc);
193             Fx = buffer_proc[index_proc-1].front();
194             buffer_proc[index_proc-1].pop_front();
195     //}
196     pthread_mutex_unlock(&mutex_proc[index_proc-1]);


230     pthread_mutex_lock(&mutex_proc[index_proc]);
231     buffer_proc[index_proc].push_back(dFy);  // output
232     pthread_mutex_unlock(&mutex_proc[index_proc]);
```

6.7 Annex G

```
255     if(index_proc == num_proc){
256             pthread_mutex_lock(&mutex_draw);
257             draw = true;
258             pthread_mutex_unlock(&mutex_draw);
259     }
```

7 Glossary

IR	– infra red
GUI	– graphic user interface
RAW	– format with no compression
RTSP	– Real Time Streaming Protocol
SDP	– Session Description Protocol
SE	– structuring element
MAX	– maximum
MIN	– minimum
ASF	– Alternative Sequential Filter
API	– application programming interface
TBB	– Intel Threading Building Blocks
OpenMP	– Open Multi-Processing
MPI	– Message Passing Interface
UPC	– Unified Parallel C

8 References

[1] FLIR SYSTEMS, http://www.goinfrared.com/cameras/camera/thermovision-a320-infraredcamera/

[2] H. Schulzrinne, A. Rao, R. Lanphier, Real Time Streaming Protocol (RTSP), RFC 2326, April 1998.

[3] M. Handley, V. Jacobson, Session Description Protocol (SDP), RFC 2327, April 1998

[4] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson, A Transport Protocol for Real-Time Applications (RTP), RFC 1889, January 1996

[5] L. Gharai, C. Perkins, RTP Payload Format for Uncompressed Video, RFC 4175, September 2005

[6] P. Dokladál, E. Dokladálova, Dilation by Flat Rectangles: An Optimal Memory, Zero Latency Algorithm, not finished yet

[7] D. Lemire, Streaming maximum-minimum filter using no more three comparisons per element, Nordic Journal of Computing, 13(4):328-339, Winter 2006

[8] E. R. Dougherty, An Introduction to Morphological Image Processing, 1992

[9] S. R. Sternberg, Grayscale morphology, Comput, Vision Graph, Image Process, 35(3):333-355, 1986

[10] J. Serra, Image Analysis and Mathematical Morphology, volume 2. Academic Press, New York, 1988

[11] J. Reinders, Intel Treading Building Blocks, O'Reilly, July 2007

[12] B. Barney, POSIX Threads, https://computing.llnl.gov/tutorials/pthreads/, February 2009

[13] B. Nichols et al., Pthreads Programming, O'Reilly and Associates

[14] B. Lewis, D. Berg, Threads Primer, Prentice Hall

[15] D. Butenhof, Programming With POSIX Threads, Addison Wesley (www.awl.com/cseng/titles/0-201-63392-2)

[16] S. Kleiman et al., Programming With Threads, Prentice Hall

[17] Pnmtotiff User Manual, netpbm doc at SourceForge, March 2005

[18] Intel® Corporation, Intel® Pentium® M Processor 730, http://ark.intel.com/Product.aspx?id=27586&processor=730&spec-codes=SL86G,SL86M

[19] Intel® Corporation, Intel® Core™2 Duo processor SP9300, http://ark.intel.com/Product.aspx?id=36691&processor=SP9300&spec-codes=SLB63,SLGAF

[20] Intel® Corporation, Enhanced Intel Speedstep® Technology, http://www.intel.com/support/processors/sb/cs-028855.htm

[21] AMD, AMD Opteron, http://products.amd.com/en-us/OpteronCPUResult.aspx?f1=Third-Generation+AMD+Opteron%e2%84%a2

9 Attachments

Attachment 1.: CD with source codes and detailed benchmarks results

Attachment 2.: Brief application user manual

Attachment 3.: Processor's architecture overview