Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS

Name Surname

Thesis title

Name of the department

Supervisor of the bachelor thesis:	Supervisor's Name
Study programme:	study programme
Study branch:	study branch

Prague 1970

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date .... signature of the author

Title: Thesis title

Author: Name Surname

Department: Name of the department

Supervisor: Supervisor's Name, department

Abstract:

Keywords: key words

Dedication.

Contents

Introduction

Motivation

Although the Moore's law still applies and the performance of computers is now greater than ever, there still exists some tasks which require a lot of computer power and despite parallelism their accomplishment takes a non-trivial amount of time. Among these tasks are for example computations of some special prime numbers, processing large data sets and many others.

It's natural to ask, how we can speed up such computations. Obvious possibility is to get better hardware, however this is limited by technical progress and can be quite expensive. Widely used approach is to parallelize the computation, that is different parts of the job are run concurrently (in parallel) which results in reduction of the processing time. Generally tasks can be parallelized on different levels. We will talk about parallelism on the level of whole computers, i.e. each computing node is represented by a standalone machine. These nodes are independent and their physical distance can be arbitrary long. The only condition which has to be fulfilled is that the nodes are connected together by the network so they can communicate and exchange data. This approach can be used with big tasks only, because the network communication is relatively very slow and for many tasks it would be a bottleneck. There are also different ways how the general term parallelism can be understood. We will consider so called data parallelism. That is, each computing node runs the same process, only the data differs. This is the main difference from the task parallelism concept in which different nodes can run completely different tasks.

Obviously, not all tasks can be parallelized easily or even are not possible to parallelize at all. On the other hand, there exists also jobs that can be split into smaller parts, each of which can be processed independently and the results then joined together in the end. Concurrent processing of such jobs is quite straightforward and allows use of the data parallelism. These jobs are suitable for our framework.

Goal of the thesis

The main goal is to implement robust framework which is able to split given job, distribute work among participating nodes, which process the assigned parts in parallel and, finally collect the results and join them together. The domain of suitable jobs is restricted to ones that are described in the preceding paragraph.

The concept of parallelization has been known for many years and many different approaches have been implemented. Some of them are discussed in the chapter 2. Nevertheless, the great majority of them uses some kind of hierarchy or at least some control node exists which has different functionality than the others. Main motivation of our work was to provide solution, in which all the nodes are equivalent, each can initiate the process or serve as a computing node. Even in our framework there exists always one node which controls the process while the other ones do the work. But the point is, that every single node can become the initiator (master) or serve as a slave or even both at the same time. Our framework was implemented with the application to the video encoding in mind. However its essence is generic, so it could be easily rewritten to be used with an arbitrary job which fulfills the above mentioned conditions.

The purpose is to implement framework which conforms to the peer to peer paradigm as much as possible. We also want to keep the logic in the node which have demands to be processed in order to ensure better control and more deterministic behavior. Another important goal is to keep the system from being error prone.

The system is supposed to run on the UNIX based operating systems. It uses standard socket API which is widely used and has been proven by thousands of applications. In order to make use of this standards the C++ language has been chosen for the implementation because it allows us to use this standard functionality.

Application to the video encoding

Video files are more or less sequences of images, so called frames. When the file is being played, the images are showed one after the other. They change many times per second so it makes the illusion of smooth video. There are different ways to code the video into the digital form so the need arises to convert the files from one format to other. Some of the well known codecs are for example *H.264* or *MPEG-2*. In order to convert the file, each frame has to be re-encoded, which takes quite a big amount of time. This makes the video files ideal for use of our framework.

Before we use it, some more details have to be revealed. Because we usually want to not only watch the video but also listen to some sound, the videos are usually accompanied by one or more audio files and they are packed together in some container. When we want to convert the video, the container has to be opened, the video extracted, processed and stored again. The audio files are not in our concern and we will simply copy them. Common container formats are *avi* or*mkv*.

There is another important thing. In order to reduce file size and decrease the requirements of rendering the video, the file actually contains only few frames which carry the whole information. These are usually referred to as the key frames. The rest contains only information how the image differs from the previous one. This approach greatly reduces the file size while preserves good ability to recover from errors, e.g. when some part of the file is damaged. Consequence of this fact is, that we cannot split the video file wherever we want, because if the chunk started with some non-key frame, the conversion would not be done properly.

The problematic of video encoding is quite complex and used algorithms are very sophisticated and advanced, so we did not reinvent the wheel and used third party software for the video encoding, namely the ffmpeg and ffprobe from the FFmpeg project. For the sake of simplicity, the program outputs always video files encoded with x264 codec and packed in the Matroska container. All of this software is free and published under the GNU General Public License or GNU Lesser General Public License in case of FFmpeg.

Terminology

To prevent misunderstandings there is a short list of frequently used terms:

- *master*(or *initiating node, initiator*) is a node which has work to do, i.e. file to encode. It initiates and controls the process.
- *task*(*job*) refers to work that should be done, specified by the initiator.
- *chunk* denotes one part of the splitted file, which is supposed to be encoded independently.
- *(computing) node* is one particular entity which communicates with the master and encodes chunks for it. Theoretically, there can be more such entities on one physical computer, because each node is determined by its communicating port and address.
- *neighbor* is considered every node which is in the database of the node we talk about.

1. Analysis

1.1 Known approaches

The idea of distributed computing was here since the 70's. Different approaches have been tried since then. Some of them are described in this section in order to show different concepts.

1.1.1 Client/server architecture

It's very common example of centralized system. In this pattern is one central node, known as the server and the rest of nodes, clients, are all connected to it. All the information and important computations happens at the server. The communication is typically initialized by the client, which sends a request, the server processes it and returns the desired result. Pros of this approach are relatively simple implementation and easy maintaining of the traffic. Also the data flow can be controlled easily. However the huge disadvantage is, that the server is by definition a single point of failure, i.e. if the server goes down, the whole system becomes unusable. Also, thanks to the asymetric nature of the system, there usually has to be more parts of the software, one to run on the server and the other to run on the clients. The system is dependent on the server's performance too.

1.1.2 Peer to peer architectures

Pure peer to peer network

Peer to peer paradigm is virtually the opposite of the previous model. Each node is completely equivalent to each other, can join the network at any time as well as leave it. This approach means great scalability and fault tolerance. On the other hand it requires more complex design and can encounter problems with security and management of the network. Our framework implements this model.

Hybrid peer to peer network

This approach combines the previous two. There exists one node which is dedicated to serve as a control server. Each node contacts this server and all the control information are goes through it while the data are exchanged in the same manner as in the pure peer to peer networks. This helps to get rid of problems with management and provide information coherence while scalability is preserved. However there is reintroduced the issue with the single point of failure.

Super peer architecture

Another try to improve the architecture considers the concept of the so called super peer. It is a computer, usually with slightly better performance than the other nodes, that plays the role of the control server, but only for some group of nodes. This forms some kind of autonomous groups (clusters) and has the same advantages as the hybrid network but the potential super peer failure is not so crucial. Also each super peer can have backup so the robustness of the network can be very good.

1.1.3 Summary

We have shown different implementations of the idea of distributing the work. Each have its own specifics and may be suitable for some application. We have chosen to implement our framework to behave like a pure peer to peer system as much as possible. Although the super peer architecture offers better control, we decided not to use it for several reasons. Mainly because our framework is supposed to be used in rather small networks where this paradigm could be quite exaggerated. Also we did not use the hybrid network because we definitely wants to avoid presence of the single point of failure.

1.2 Existing solutions

Since encoding of the video files is quite reasonable task, there was a few implementations of the similar issue. For example D. Hughes and J. Walkerdine from the Lancaster university published in their paper a solution which is using Lancaster's P2P Application Framework. They implemented a Java plug-in for this framework which uses Microsoft Windows Media Encoder SDK. Their approach was quite similar to ours and they achieved quite persuasive results. However their solution was usable in the specific environment only.

1.3 Framework description

1.3.1 Basic overview

The heart of the program is one executable file, that should be accompanied by the configuration file. This is discussed in detail in chapter 2. Main options that should be set are IP address and a number of port on which the program should listen. It is also essential to provide credentials (i.e. address and port) of some node which should be contacted by default. When more nodes are spawned, the network is formed and the computation may begin. Note that information about neighbors spread in a nondeterministic manner, it matters who is contacted by whom. So it's good to have one or more nodes which serve as super peers, i.e. the rest contacts only these nodes. Once the network is established, the computation can begin. If one (or more) nodes have tasks to be done, it can start the process. The file is then processed and divided into chunks in the initiating node. These chunks are distributed among neighbors, processed and returned back. Once the initiator has all chunks back, it joins them together and the process ends.

1.3.2 Neighbor maintaining

Each node maintains the list of its neighbors. The list is refreshed periodically, so the node keeps track of the current network state. Besides this main list exists another list, which contains potential neighbors. Generally each node that has communicated with the given one sometime in the past is added to the list of potential neighbors. The purpose of this list is to reduce amount of time spent with maintaining the neighbors list. That is, no more than required count of neighbors is maintained.

Neighbors are uniquely identified by the pair of address and communicating port. This is sufficient for the potential neighbors, because before the neighbor is added to the main list, it has to be contacted. Additional info is then added so more complex structure is needed for storing neighbors. This structure contains information about neighbor's quality, last known state etc. The quality of the neighbor helps to prefer one neighbor to the other when picking the one to contact. It is updated after each chunk delivered from the given neighbor and reflects the neighbors computation power together with the speed of the connection.

Initialization and discovery

The node is provided with the address and port of the neighbor which should be contacted by default, that is, when the neighbors list is empty. Each node has minimum count of neighbors which should be in its list. This number is checked periodically. In case that the list is too small, the list of potential neighbors is checked. If its empty, the node has to obtain more neighbors so it picks one neighbor from the list, contacts it and receives some suggestions. If the list is empty, the default node is contacted. Once the suggestions are received, the node adds the new addresses to the list of potential neighbors. When the list contains some potential neighbors, the node can contact them and add as regular neighbors.

Gathering neighbors

When the node has not enough neighbors, it can use another mechanism. This mechanism uses a flood technique to spread the request among the nodes in the network. The request is send to each neighbor, which spreads it further in the same manner. However the request is equipped with time to live value which is decreased after each hop so it does not spread forever neither too far. Once some node receives the request and is not busy, it contacts the initiator directly, so it can add him to the list of potential neighbors and then possibly as the regular neighbor.

Withdrawing from the network

When the node wants to leave the network it has to abort the process, if it is initiator. Then a special message is sent to each neighbor, which informs them so they can react accordingly. That is, if the neighbor has tasks to process for the leaving one, it removes them from the queue and then removes the neighbor itself. Removed neighbors are completely lost, they are no longer stored in either of the lists. The process of saying goodbye to neighbors is asynchronous, the neighbors do respond with some acknowledgement message, but the leaving node does not wait for it, because it could cause potential deadlock in case that the other node is unable to respond for some reason.

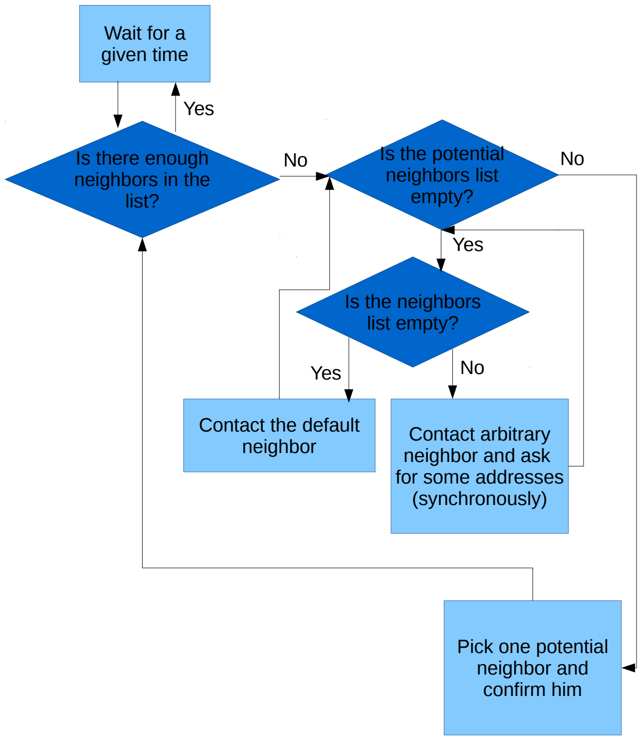

Figure 1.1: Maintaining the neighbors

Neighbor's failure

There is no guarantee that all the neighbors leave the network properly. The program itself can encounter error or be terminated violently. Another possibility is some unpredictable error of physical character, for example power failure, network problem etc. In those cases it's essential for the other nodes in the network to be informed about this fact. Especially it's very important for the initiator who had some tasks processed by this node. In order to ensure handling of this possibility the neighbor list is checked periodically. If some neighbor does not respond, it is removed fro the list and all the data connected with it are treated accordingly. Namely the chunks assigned to it are resent.

1.3.3 Distribution of chunks

Once the file is splitted, the chunks has to be distributed, processed and finally collected. To achieve this, we must deal with several issues, which are described further.

Life cycle of the chunk

Firstly, we have to keep track of every chunk's state. That is, we have to know whether the chunk is waiting in the queue, has been sent to process or have returned already. For easy manipulation each chunk is represented by the dedicated structure, which holds information about it as well as information essential for transfer. This structure is further described in chapter 3. From now on we will use the term chunk for both the physical file and the reference. Typical chunk's life cycle looks like this: The chunk is created and pushed to the waiting queue. Later it's popped out and transfered to the processing node. There it is enqueued for processing, then processed and sent back. Meanwhile the initiator holds the reference in the list of tasks being processed. In case of failure of the processing node, the chunk is pushed to the waiting queue again. Also when the chunk waits for return, it's checked periodically and if the computation takes too long, it's resent too. Note that this can cause the situation, when one task is being processed by more than one neighbor. However it's not a problem at all, because if the chunk returns more than once, it simply is not accepted, because it is not needed anymore. Once the chunk returns successfully, the reference is moved to another list, where it waits for completion of the task. When all the chunks are collected, the joining process may begin and the task execution ends.

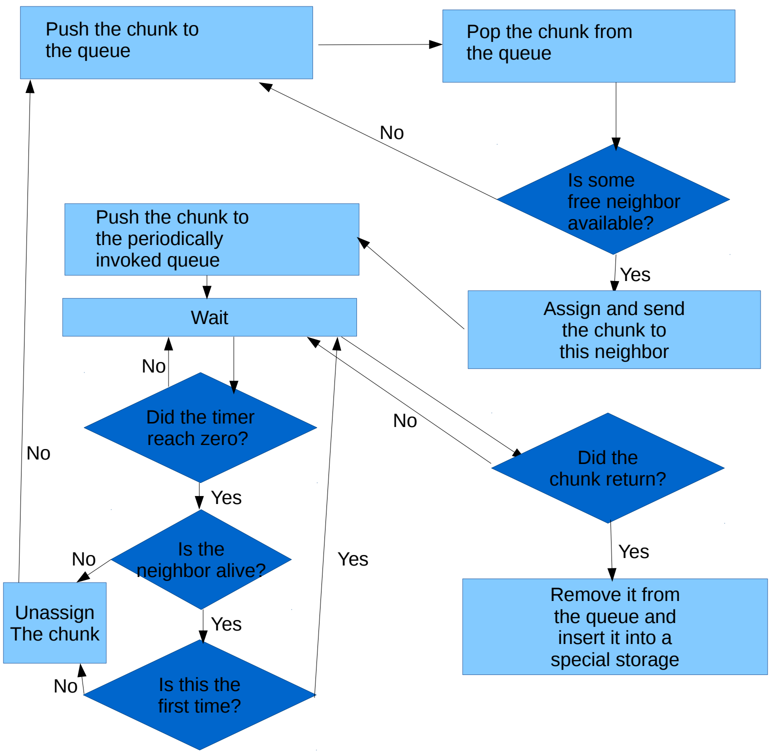

Figure 1.2: Processing a chunk - initiator part

Storing files

Tightly coupled with this process is the problem of storing the file. During the processing of each chunk four files has to be created. Firstly is created when the original file is splitted. This file can't be removed until the processed chunk returns, because it has to be available in case that the conversion fails for some reason. Another two files are created at the processing node, one for the input and one to store the output. The last file is created at the initiating node again to hold the processed chunk. This means, that the initiator has to has free disk space at least two times the size of the resulting file.

Picking neighbors

Last but not least we have to choose policy to whom the chunks are distributed. We want to achieve as big speedup as possible, while preserve rather small list of neighbors. When the chunk is popped out from the queue, the initiator looks for suitable neighbor. That is the neighbor has free status in the corresponding structure. If no such neighbor is found, the chunk is re-queued and another try is postponed. Also the gathering process described in the previous section begins. If some neighbor is available, the chunk is assigned to it and the transfer may begin. This mechanism may seem odd, but if it was the other way around, it could possibly cause problems. If free neighbor was chosen first, the process could block afterwards, because of the lack of chunks. Till some chunk become available, the picked neighbor's state could change. So it is necessary to process the actions in this order. The initiator keeps track how many chunks were send to the particular neighbor and it does not send more than specified count to one neighbor because it could potentially lead to delay. The flag indicating whether the neighbor is free helps to control the flow. Each time chunk is assigned to the neighbor, the flag is set to false value to prevent sending more chunks in parallel. It's set to true again after the successful completion of the transfer. When the neighbor is too busy, it can express it in the communication, so the flag is set to false to prevent overloading of the neighbor. The flag is also refreshed during every periodic check.

1.3.4 Security issues

The present implementation is possibly vulnerable to some security threat. This is caused partly by the pure peer to peer nature, because it is difficult to control traffic and authorize all nodes in dynamic environments like this. It also was not our aim to solve this issue. The framework is supposed to be used mostly in LAN's where all the peers are trustworthy. Otherwise it could be compromised easily. For example when the encoded chunk arrives, it is not checked whether it has been sent to this node or not. So a malicious chunk could be infiltrated causing bad output or even failure of the joining process.

1.3.5 Networking handling

The network communication is the most important part of the framework. Standard C library functions and structures were used which are conforming to POSIX.1-2001 standard. Although the program is intended to be used on the UNIX or UNIX-like operating systems, it should be portable to the Microsoft Windows systems as well thanks to the use of this standard. To preserve simplicity, all the network communication makes use of the TCP protocol. The system primarily uses the IPv6 addresses, but it can be run in the mode which uses addresses of the IPv4 family only. Further and more detailed information can be found in chapter 3.

1.3.6 User interface

To provide interaction with the user, the curses API is used. This library makes it possible to control the terminal screen. That means, the application does not require any special GUI libraries and is able to run interactively even on machines without the X server. The control is rather simple, offering possibilities to load the file, start or abort the process, show information about neighbors and so on. The interaction require only a keyboard, no mouse is needed at all. Concrete information together with some examples can be found in chapter 2. Detailed information about the implementation, namely the synchronization problems are discussed in chapter 3.

1.3.7 Problems, alternatives and possible improvements

Some alternative approaches were also considered during the designing. One of them was to don't use periodical checking at all. It was based on the idea, that there would exist permanent connection with each neighbor and the change of state would be indicated by the events related to this connection. However it was rejected due to the requirements connected with keeping the connection. Furthermore, the changes of ready state of the node would have to be check either periodically or the node would have to inform all its neighbors (even potential) about each change which would lead to another problems to deal with and is in contrast with passivity of the slaves anyway.

Another issue was whether use some more sophisticated way of distributing the chunks. Namely some kind of hierarchy was considered when the chunk references would be distributed to neighbors in packs, where it would be further splitted and distributed among neighbor's neighbors and so on, so a kind of a tree structure would be formed. The transfer of the file would be processed directly between the initiator and the leaf node. However this approach turned out to be complicated and brings many problems. For example in case of failure of some node which is high in the hierarchy a lot of chunks would have to be re-distributed. Also the initiator's ability to control the distribution would be reduced. Moreover, the advantages of this approach are not so significant at all, because the biggest portion of time is spent during transfers and processing the chunks and the time spent with distribution of references is not important at all.

Also the current implementation creates a separate connection for each data transfer. Alternatively, each chunk could be delivered and returned using the same connection which would lessen the demands of the communication. The connection's termination would also indicate problem with the chunk's processing or the neighbor itself. But the connection termination does not necessarily mean the failure of the process. Because it is desirable to avoid needless re-encoding of the chunks, this situation would has to be treated specially which would introduce additional problems. Also this approach does not fit very well to the current design which uses system of commands.

2. Installation and use

2.1 Download

First it's essential to get the source code. You can clone the code directly from the git repository using command

```
#git clone https://github.com/vojtsek/VideoCompression.git
```

Alternatively you can download the zip file and unpack it in some directory.

2.2 Requirements, installation and first run

To run the program successfully it's essential to have *ffmpeg* and *ffprobe* installed on your computer. Although technically it doesn't matter which codec is used, the program currently uses only H.264 standard for encoding of the output, so it assumes the *ffmpeg* has been compiled with the *x264* codec. Otherwise the program does not have any special requirements except standard libraries which should be available on all UNIX systems, so once you installed these programs, you can change to the directory containing the source code and run the installation script:

```
#cd DVC
#./install.sh
```

The installation script is a regular Bash script, so the Bourne again shell interpreter is required to run it successfully. It explores your computer, i.e. gets the IP address, finds location of *ffmpeg* binaries etc. Then it creates home directory for the program. The home directory contains data of the program's run. These include intermediate results as well as the final result and log files. The installation continues with generating the configuration file. This file is crucial for the framework. Before setting each option, the script prompts you for confirmation of the value. If you type nothing and just press the Enter key, the suggested value is used, otherwise the script uses your input. The resulting file is stored in the *bin* directory which is created during the installation. It's a plain text file so you can edit it anytime in the future. The script then continues with compilation of the program. If everything is all right, you can change to the newly created *bin* directory and continue. The directory *bin/lists* contains some supporting files that should not be change, otherwise the program could behave improperly.

The last step before you can run the program is to check the configuration. The configuration is saved in the *bin/CONF* file, which is created by the installation script. It's important to provide valid path to the *ffmpeg* and *ffprobe* executable and address with port of the neighbor that should be contacted initially. Otherwise you won't be able to join the network. The field *MY_IP* is not essential as long as the initial neighbor is alive - it will be recognized automatically.

Then you can finally run the program. Some of the settings can be changed by providing options, the available ones are listed below.

- -s … no address will be contacted initially
- -n address~port_number … node to contact initially
- -a address~port_number … address and port to bound to
- -h directory … path to the home directory
- -i file … file to encode
- -p port … listening port
- -d level … debug level
- -q quality … quality of encoding

If the string *ipv4* appears among parameters, the program will use only the IPv4 addresses, in which case should be the *CONF* file changed appropriately.

2.3 Using the program

When you run the program, the initial screen appears. You can choose desired action then using function keys. You can see the initial screen in the figure below. Available options are highlighted.

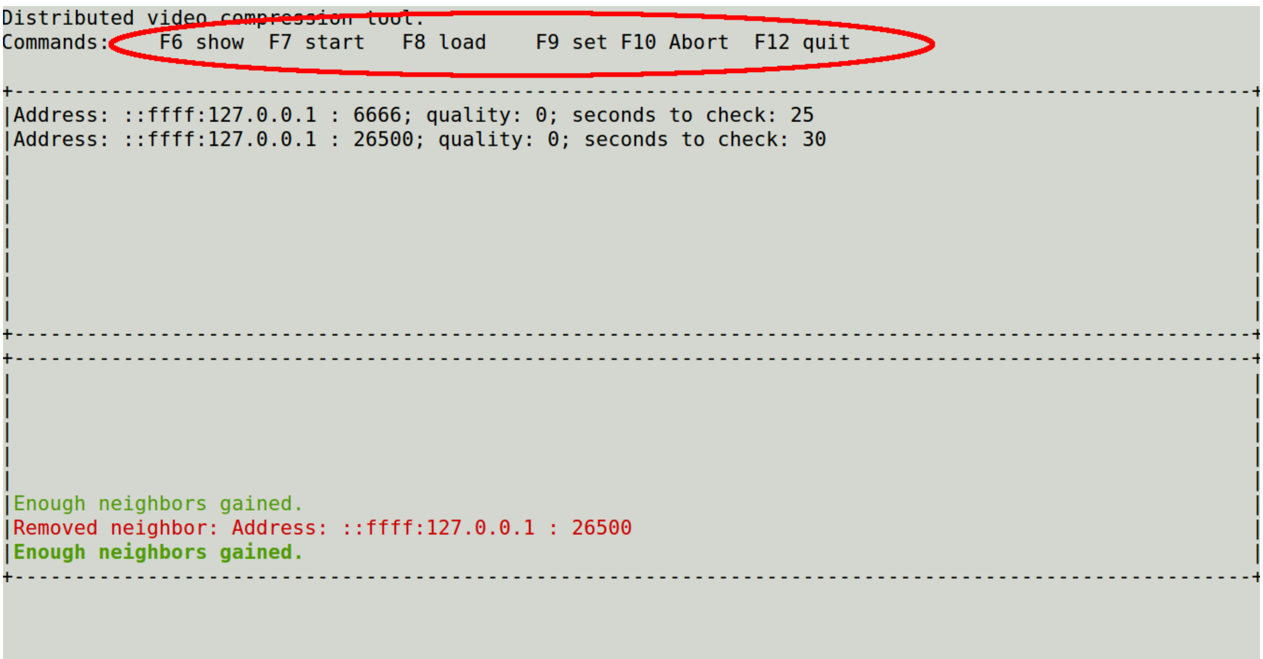

Figure 2.1: Initial screen after joining.

The important key bindings are listed in the table below.

First you should load the video file. When the corresponding function key is pressed, the program prompts you for the file location. You can type in the absolute path of the file. Once you use the file, it is stored in history which you can browse using up and down arrow keys.

Then you can set some parameters or show different information using F6 respectively F9 function keys. These keys provides set of options which you can choose from. When you are satisfied with the settings, you can start the

F6	Show information
F7	Start the process
F8	Load the file
F9	Set values
F10	Abort the process
F12	Quit the program
Up, Down	Traverse available options
Enter	Confirm the input

Table 2.1: Table of the control keys.

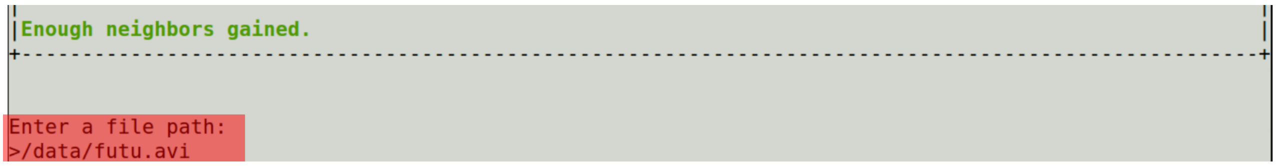

Figure 2.2: Loading the file.

process. The program then starts splitting the file and distributing the chunks while informing you about the progress. When the process is done, the file is joined and you can do further actions.

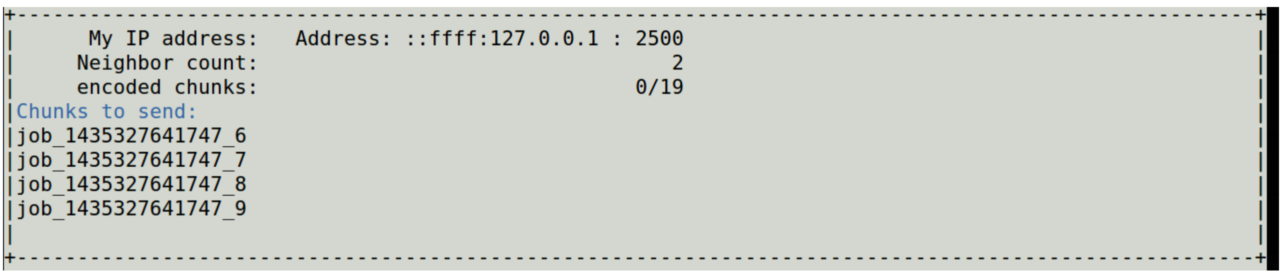

Figure 2.3: Overview of the process.

To obtain more information about what is going on, the *-d* option may be used which allows to specify level of debug messages that will be shown.

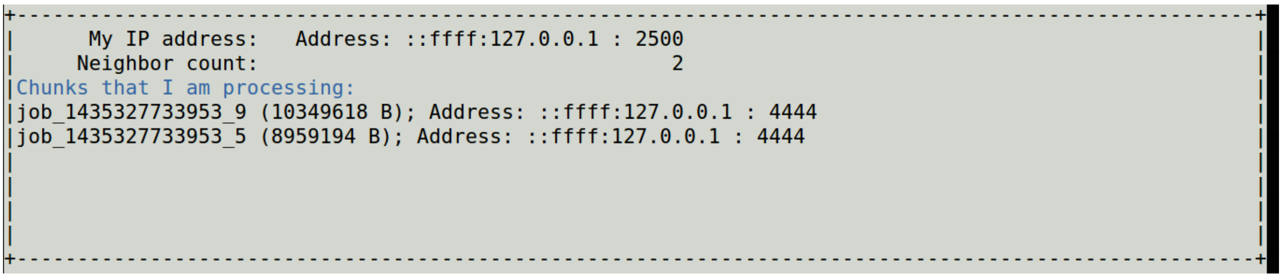

Figure 2.4: Processing of the tasks.

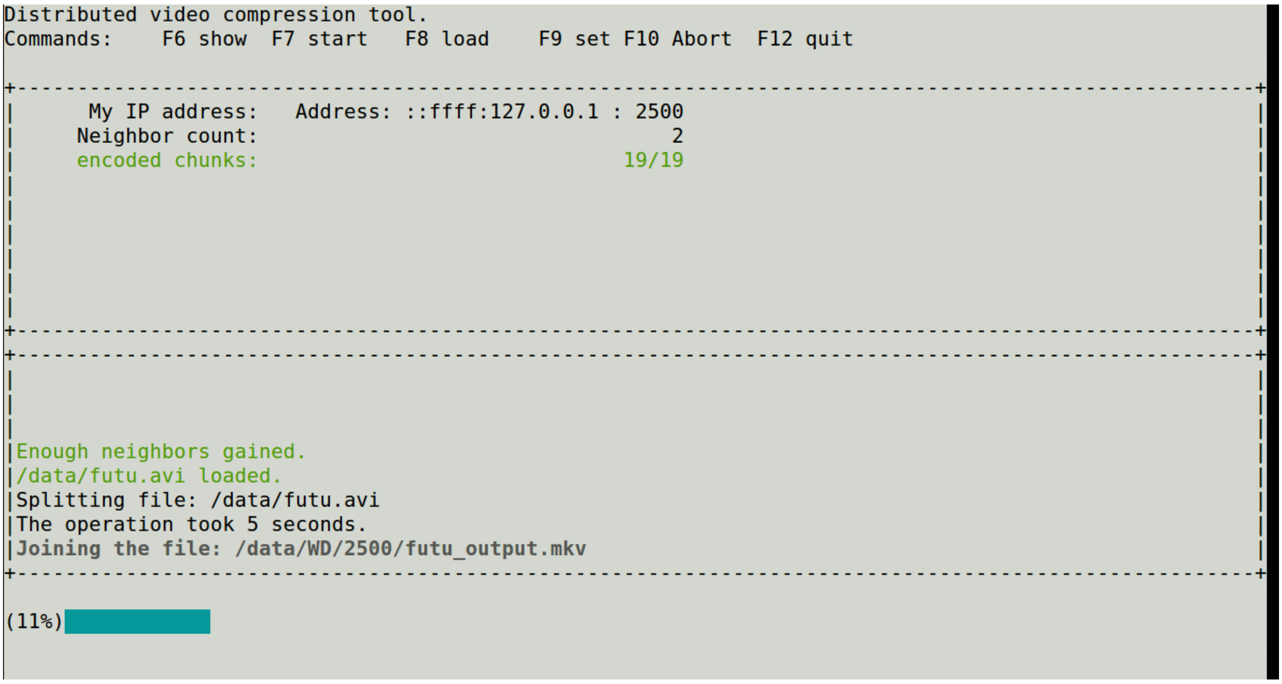

Figure 2.5: Joining the file.

3. Implementation

This chapter describes some used mechanism in more detail. It also introduces some classes and methods, but it is not supposed to serve as a detailed and full documentation. The Doxygen software has been used to generate documentation so the complete overview of the code can be found in the attached HTML document. The program is implemented in the C++ language. This allows the use of standard C library functions. Also the functionality provided by the C++ STL is exploited. Some of its containers are used as a base for containers with synchronized access implemented in the framework. Some of the functionality from the C++11 standard is also used, so the use of the program is limited to computers with compiler that supports this standard.

3.1 Networking

One of the most important issue is how to handle the networking. The chosen approach will be described in this section. As it has been said already, the program uses TCP for any kind of network communication. This is certainly good option when we want to handle data transfers, however, it can be considered unnecessarily demanding for simple tasks. However to preserve the implementation simple, we chose not to use UDP. At least the advantages of the use of TCP are exploited. To utilize the possibilities of the operating system, standard socket API is used. Information about addresses are stored in the *sockaddr_storage* structures which are suitable for storing both IPv4 and IPv6 addresses. This approach makes it possible to switch between both the versions easily. There are also some helper functions to work with address structures which are generic in the use of an address family. To provide easy manipulation with addresses, the structure *MyAddr* was created which groups the related functionality together.

3.1.1 Spawning connections

The ultimate class to handle the connections is the *NetworkHandler* class. It provides all the necessary functionality. Each instance of the program runs separate thread which binds to the given listening port and starts accepting connections. When the connection is accepted, new thread is spawned to handle the connection. When the program wants to make the connection, it provides structure referring to a given neighbor together with the commands to execute. to the *NetworkHandler* instance. The connection is then spawned and handled. Here we encounter the topic of commands. Each action consists of set of commands that implements it. Commands are instances of a given class. Each command has two parts. When one node initiates the connection, it provides the vector of commands to be executed. They are processed in the loop in the following manner: The command's *execute* method is invoked. It communicates over the network. Firstly it sends name of the command to be invoked in the peer node. The peer's thread loops too. First it reads the name of the command and then it invokes it. In that moment there are command methods running in both nodes and they can communicate. When they end, the receiving node waits for another action while the initiating node spawns next command from the vector, if any is present. Otherwise it ends the connection. This mechanism is used to handle all the network communication. What happens in case of problems is described in the section about handling errors.

Incoming connections are always handled asynchronously. Nevertheless, the outgoing connections may be handled in the synchronous way. In can be essential sometimes. For example if the node needs to obtain some potential neighbors because his list is empty, it has to wait until the action ends, because if it just sent the request and continued, it would probably find the list still empty.

3.1.2 Protocol

As it was said, the communication happens thanks to commands. Thread that handles the incoming connection is provided with respective file descriptor and address of the communicating node. First data that appears are considered as the listening port number of the node, so it can be identified in the neighbors list or added to the list of potential neighbors. Then the name of the command is sent, which is represented by the *enum* type. If no error occurs, the confirmation is send and the appropriate command's method is invoked. After returning from the method, another command is read. If there are no data left, the connection is closed.

3.1.3 Transferring the data

First problem every network application has to deal with considers the byte order. The POSIX sockets API provides set of functions to deal with it. Namely it's *htons* and *ntohs*, or *htonl* and *ntohl* respectively. These functions convert the native representation of short (long) data types to the network byte order. The framework uses these function.

Base transfer functions

Each of the following functions has basically two parts, sending and its receiving counterpart. Integers are stored as the *int64_t* type which ensures correct communication even between 32 and 64 bit nodes. Then they are sent using the mentioned converting functions. When the string is transfered, first is sent the length of the string, followed by the appropriate number of characters. Commands are sent as numbers, wrapper functions are used which converts between the *enum* type and *int32_t* explicitly. Sending the structures containing the addresses is managed by another special function. It converts the address to strings and sends it in this form, followed by the port number. This allows to handle both IPv4 and IPv6 addresses, the format is recognized during the reversed conversion.

Transfering files

The most delicate task is to transfer the files. The file is first check, and its size is determined. Then it is sent and then the function repeatedly reads part of the file to buffer and sends it until all file is processed. The count of sent bytes is compared with the actual file size in the end. The receiving side accepts the bytes and writes it to the file continuously, the file size is checked in the end. The data are first written to a temporary file and after the successful transfer it is renamed. This mechanism prevents inconsistency of the received files. Each file is referenced by the structure. Among other things this structure stores counter of unsuccessful sent tries. If the counter exceeds given limit, the neighbor is treated as invalid and his state is set to non-free. The file is also checked, if it is not valid for some reason the whole process has to be aborted because there is no way how to fix one specific file.

3.1.4 Handling errors

Unfortunately the network environment is quite error prone and all the action has uncertain results. Moreover, the communication can be interrupted at any time. Because of this it is important for the network application to be able to deal with different error situations.

Errors during the connection handling

Almost all the functions indicates error by the negative return code. These codes are checked so the error can propagate. If some error happens during the control communication, the loop handling the connection simply brakes, so the connection is closed. The commands are invoked in the try-catch block, so if the data have been corrupted or the synchronization has been lost, the next invalid command name raises an exception and the error is handled. The loss of the synchronization may be detected thanks to the obligatory confirmation of every command.

Other errors

If an error happens during the file transfer, the receiving side detects inconsistency thanks to checking of the file size, so the bad file can be removed. Generally, if any error is encountered during the communication, the corresponding execute method indicates it by its return value, so it can be propagated further. The signal handler also has to be set to cover situations when the connection is destroyed unexpectedly and the SIGPIPE is delivered.

3.2 Structures' overview

This section describes two main structures used in the framework. They common sign is, that they inherit from the Listener class, so they have to implement the invoke method. This fact makes it possible to use them as periodic listeners.

3.2.1 The TransferInfo structure

This structure serves for referencing the chunk. It contains flags used for transfer, addresses (source and destination), information about the video and path locating the physical position of the file. This field is important because it makes it possible to reference the file. It is changed during the transfers. The structure also contains information which help to determine the encoding process as well as statistics describing the result. These are used to compute and update the quality of the neighbor. The structure is also equipped with a pair of methods that transfers it over the network, which simplifies its usage. Periodic invocation decreases the timer. It is used when the referenced chunk is waiting for return. For the first time the timer reaches zero the neighbor is checked, if it is alive. If yes, the timer is send one more time. If the neighbor doesn't respond or the timer reaches zero for the second time, the chunks is resent.

3.2.2 The NeighborInfo structure

Instances of this structure are kept in the NeighborStorage class. They represent the neighbor, that is its address and listening port. It also keeps the information about quality of the neighbor. The time from last check is stored too. Periodic invocation causes the timer to decrease and possibly contact the neighbor to refresh the state.

3.3 Important classes

3.3.1 The NeighborStorage class

It is class used for storing the *NeighborInfo* structures. It provides several methods to maintain neighbors list while preserving synchronization. This is crucial, because the information about neighbor can change any time but it's desirable to keep our knowledge consistent. The class has one instance and helps to keep the information about neighbor at one place, so the manipulation can be controlled.

3.3.2 The NetworkHandler class

This class is used to handle all the networking issues. It provides functionality to spawn the connections or contact neighbors. It also hold the list of potential neighbors. This list differs from the neighbors list, because besides address and port, which are necessary, no further info is stored about potential neighbors. Also, most of the other classes have a notion about this list. When the lack of neighbors appears, simply the function *obtainNeighbors* is invoked, which uses the list internally. This class also handles adding of the new neighbors.

3.3.3 The Data class

It is a singleton class which helps to keep all the data at one place while accessible from anywhere in the program. It holds the instances of the *NeighborStorage*, *State* and all the queues used during the transfer and the encoding process.

There are more significant classes such as TaskHandler and WindowPrinter, which will be discussed in the corresponding sections.

3.4 Periodic actions

The framework uses a mechanism which invokes some actions periodically. Pros and cons of this approach are described in the respective sections, namely the alternatives. To implement this mechanism, separate thread runs that loops and once after each time quantum it invokes methods of structures that inherits from the *Listener* abstract class and are stored in the special queue. The time quantum is defined as a constant, so all timers actually express count of the quanta left. Obvious disadvantage of this approach is busy waiting that is used in the loop. Alternative approach could use a signal handler and setting an alarm. However, this could lead to not necessary asynchronous interrupts. Since the mutexes are used to avoid race conditions, a problem could occur if the signal interrupted some method holding a "bad" mutex.

3.5 Queues

As it is described in the corresponding section, the references to chunks are stored in different queues, depending on the state in which they are. Since the whole process is non-deterministic, different conditions may occur and more than one thread could need to work with the queue at one moment. This means, that some way of synchronization has to be provided. Because of this textit{SynchronizedQueue} has been created. It is a kind of a lock-free structure. Basically it provides usual functionality that could be requested from the queue, but ensures avoiding race conditions because only one thread at a time can access the underlying data. This is implemented by the use of textit{mutex}. The *pop* method also uses the conditional variable, so if there are no data that can be popped at the moment of invocation it blocks and waits for a signal.

3.6 Working with input and output

All the file operations are customized for use on the UNIX operating system. Nevertheless, there is possibility to implement respective functions to work on different operating systems easily.

File operations

Each chunk is stored in a separate file. For this reason several functions were implemented to allow easier work with the file system. These include functions for manipulation with the filenames, controlling files and working with directories. Also there is a generic function which spawns an external process. This function uses process forking, spawns the desired process and returns contents of its standard output and error streams. The function also accepts time after which it kills the process. The result of the process' run propagates in the function's return value so the caller can react accordingly.

Working with the video

The video processing is secured by the *TaskHandler class.* When it is loaded, some useful properties are obtained thanks to the *ffprobe* program. To allow easy processing, the output is in the JSON format which is then parsed with the help of the *rapidjson* library. This library consists of header files only and is distributed as a part of the source code. It is available under the MIT license which makes it suitable for usage. These values can then be showed using the F6 key. More importantly, it is used to compute the number of chunks that will be created. Note, that the number of chunks can change slightly during the splitting process. It is caused by the fact, that the teoretically computed count does not consider the positions of key frames. So the chunks can actually have different sizes and thus their count differs. This fact also implies, that each chunks has got slightly different size. When the process starts, the instance of *ffmpeg* process is spawned which splits the file. The resulting files are then stored in the special subdirectory of the working directory. For easy identification, each process is assigned a unique code determined by the time stamp. The chunks are then numbered in increasing order. The names are then stored in the referencing structures that are created at this point. Then the chunks are distributed as described in the section about distributing. Each processing neighbor encodes the chunks using the *ffmpeg* and then sends it back. The information about the chunk, such as the level of the encoding quality are stored in the referencing structure. Once the chunks are collected, first a list of the files to join is created and then the *ffmpeg* is used once more to join all the chunks to the output file.

An alternative approach to splitting the file was used during the development. Firstly the position of every split was computed. Then it was spawned one process per each chunk. The advantage was, that the distribution could begin after the first chunk was created and therefore some time was saved. However this approach turned out to be bad because of the existence of key frames. It does not allow to split the file at the arbitrary position so certain shift was observable in the resulting file.

User interaction

To provide interaction with the user, the *curses* library is used. User can provide input from the keyboard. There is set no delay of the input, so the buffering is disabled. This causes the need to handle the reading manually but also makes it possible to control the input and accept the commands immediately. The output is provided using the *WindowPrinter* class. This class stores the queue of lines of output and provides functionality to add or remove some. Each record holds the line together with the information of the style how to print it. The screen is divided into four parts. Each part spans the whole width. There is a line displaying available commands at the top. The biggest portion of the vertical space belongs to two windows. The first one displays different information about processing, neighbors or file properties. The second window displays the status changes, notifications and potentially some debugging messages. The most bottom part shows a prompt when user input is required.

Because there is usually a lot of threads that can cause the screen to refresh (this means the particular *WindowPrinter* instance is updated), it is important to allow only one graphical update at the moment, otherwise it could cause inconsistency of the graphical data. This is ensured by a mutex assigned to each *WindowPrinter* instance.

3.7 Synchronization

Because of the nondeterministic nature of the application, it is necessary to provide some kind of synchronization to ensure data and information consistency. This problem is solved by the use of mutex and conditional variable available in the C++ standard library. There are implementations of queue and map like structures which offer lock-free access. These classes uses containers from the STL and synchronization primitives mentioned above. These structures, namely the *SynchronizedMap* and *SynchronizedQueue* are used to store chunk references. Operations with the list of neighbors have to be synchronized too, because for example one thread could be working with the neighbor's reference while the other finds out that the neighbor is not responding and wants to remove it from the list.

Another area which have to deal with some race conditions is output which is displayed on the screen. The output is handled by the *curses* library which provides practically raw access to the screen. This means, that if more threads try to work with the screen at one moment, there is high possibility that they compromise each other and nonsensical data are displayed at the output. Moreover, this situation can also possibly result in the segmentation fault. To avoid these situations, the data that are supposed to appear on the screen are stored in the respective instances of the *WindowPrinter* and mutexes are used to allow only one thread to change the content of the storage or refresh the screen. This approach has a disadvantage, that each call of the routine that produces some output is possibly blocking

3.8 Error detection and recovery

During the process, various types of errors can occur. Errors connected with the networking are discussed in the respective section. Here we will describe other errors that could possibly happen.

Neighbor failure

If an unexpected failure of some node occurs, the other nodes which has it in their lists must react. If the communication is interrupted in the middle, there is no way how the other node can recognize the failure, so the command execution just fails. However this usually leads to repetition of the command which registers the failure. Generally the failure is noted when the try to establish a connection with the given neighbor fails. This can occur when processing a command, checking a neighbor or when the timer assigned with some chunk reaches zero. In every of these situations the same function is used, so the situation is always treated in the same way. The unresponsive neighbor is removed from the list. If it has some chunks assigned, they are queued for send. If it has chunks being processed by the current node, those chunks are trashed.

Chunk disappearance

After the chunk is send to be processed, it is pushed into special queue where it is checked periodically. If the time is up, the respective neighbor is checked. If it is responding, the timeout is set once again. If it does not respond, or the timer reaches zero for the second time, the chunk is queued for send. Also, in case of the neighbor failure, the neighbor is removed. This can lead to a situation, when one chunk is being processed by two different neighbors at the same time. However, after it returns for the first time, it is put into the dedicated storage, so if it returns afterwards, it is simply rejected. This approach could be improved by introducing new action which would cancel the chunk's encoding process at the remote node. It was not implemented though, because it is not clear on which node the process should be canceled. Connected with this problem is the situation, when one node receives by accident one chunk more times. The files are checked and what is more, the transfer uses temporary files so the worst scenario involves wasteful encoding of the chunk for the second time. Another issue which has to be solved is how to set the timeout. When the neighbor is involved for the first time, we have no information about its performance, so the timeout is set respectively to the size of the chunk, default multiplication factor is used. When the neighbor has already quality factor assigned, it is used to compute the timeout. So the quality coefficient can be seen as time needed to encode and transfer some unit of data.

Other errors

Errors can also be encountered during the manipulation with the video. Because all the video related problems are manipulated by the external programs, the mechanism is used, which can control the process. The approximate upper bounds are set for each task that is supposed to be executed and if the process' execution takes too long, the signal is send which terminates it, thus the error code is returned.

4. Experiments

The purpose of the application is to speed up the computation process, thus it should be verified, whether the improvement makes sense or do not. The improvement should correspond to the number of nodes involved in the computation. What we wish is, that the dependence is of some linear form, that is, the computation gets faster with every additional node and it improves by the same steps. In this hypothetical ideal case two nodes means two times faster computation and one hundred nodes means one hundred times faster achievement of the result. However, this is impossible for several reasons. At first, we must consider time that is taken by the division process. More time is needed for transfers and final join operation. Another problem raises because of the fact that the transfers are quite demanding themselves so when more transfers are ongoing at a particular moment, the initiator is more utilized and the process could be slowed down due to this fact. This also implies that the improvement does not raise constantly when adding more nodes. Finally, we must consider that in the real situation delays can appear due to technical reasons, network congestion or node failures.

4.1 Aproach to testing

If we want to obtain reasonable data, the measurements must be repeated several times to prevent deviations. Also we want to keep the measurements independent so its statistical processing is easier. Our approach to running the tests and gathering results is described in this chapter. Our main goal is to measure the improvement, but we would also like to measure the impact the particular setting has on the result.

Because of the number of tests, it is desirable that the testing process is automated. Because of that, special Bash script was used to run the tests. The script is tailored to be used at the testing laboratory, so it may need little modifications to work in some different environment. It is distributed with the source code of the framework. To allow automated and robust execution of the test, special functionality was added to the program. It is invokable by option given at the start time and causes the program to run in non-interactive mode, i.e. keyboard input is accepted, the program just processes given file and ends. This options ?? all the essential data are given when the program is started. To keep the measurements independent, all the instances (on every node) of the program are started at the test beginning and they are killed in the end. Communication with the remote nodes is handled by the ssh program. The testing script uses a special file which describes the particular run. Working example of such file together with explanations of the values is given below.

```
v6 // use IPv6
/afs/ms/u/h/hudecekv/futu.avi // location of the file to be re-encoded
2 // run the whole scenario twice
slower // quality of the encoding
10000 // chunk size [KByte]
```

```
2048576 // transfer buffer size
spawn u-pl1 2221 // spawn the program on the machine 'u-pl1', use port 2221
spawn u-pl2 2222
spawn u-pl4 2224
spawn u-pl5 2225
spawn u-pl6 2226
spawn u-pl7 2227
wait 10 // wait for ten seconds before next action
kill u-pl4 // kill the instance of program running on machine 'u-pl4'
spawn u-pl8 2228
spawn u-pl9 2229
spawn u-pl10 2230
```

Thanks to this mechanism, various scenarios can be run easily without the need of human interaction. The test data were collected by running the test ten times for the given count of nodes. The count varied from one to ten nodes involved. Each test was run once with chunks of 40 000 kB in size and once with 10 000 kB chunks. The same file was used each time as well as the encoding quality. Each test stored various results, among others the average times needed for transfer and encoding, number of chunks, quality and count of involved nodes. Because we had not the chance to run the tests in some dedicated network, the computation times may vary for the given setting depending on the current conditions. The tests showed however, that if we multiply the average time needed to encode one chunk by the count of chunks, the product corresponds to the time that would be taken by the normal encoding process. This allows us to deal with the problem, because we can compare the time with this computed estimation and the error will be minimal. The desired values have been gathered in two ways. Some of them, for example average transfer and encoding times, are measured directly in the program and then outputted in special file. The script just reads it from this file. The rest of the values is obtained in the script. ## Results Measurings showed, that the dependence between the number of involved nodes and the improvement is approximately logarithmic. To evaluate the data, simple linear regression model was used. Specifically, subsequent formula was used:

$$\frac{encoding\_time}{singletime} = x_1 *$$

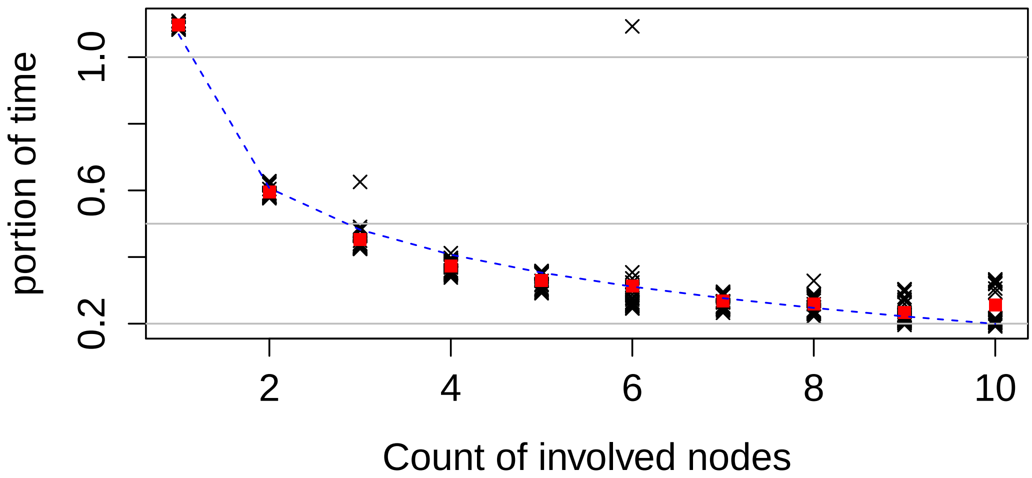

Figure 4.1: Achieved improvement

Conclusion

List of Figures

List of Tables

List of Abbreviations

Attachments