University of Bath

MPHIL

The Development, Implementation and Analysis of a Real-Time Parallel Algorithm of Sliding Discrete Fourier Transform

Tsimashenka, Iryna

Award date:
2011

Awarding institution:
University of Bath

University of Bath
Department of Computer Science

The Development, Implementation and Analysis of a Real-Time Parallel Algorithm of Sliding Discrete Fourier Transform

Iryna Tsimashenka

Submitted in part fulfilment of the requirements for the degree of
Master of Philosophy in Computer Science
University of Bath, 2011

Abstract

Audio processing is an interesting and challenging area due to the strict requirements of high-level human ear perception. Although audio processing is a part of digital signal processing, there has not yet been any real-time parallel implementation of the essential signal processing tool. We have developed, implemented and analysed a Sliding Discrete Fourier Transform algorithm using a particular vector parallel processing. This investigation is focused on speeding up real-time parallel implementation of the SDFT algorithm. It has developed and implemented some real-time parallel algorithms with different methods of data copy in order to find the fastest algorithm. In order to achieve a whole signal processing tool for professional use in the audio processing field a real-time parallel algorithm of Inverse Discrete Fourier Transform has been developed. In order to speed up this algorithm some versions were developed with different methods of data copy. All algorithms were tested and analysed. Real-time parallel algorithms were achieved with promising results. The signal processing tool of the SDFT and the IDFT algorithms was attained.

Acknowledgements

I would like to thank the following people:

- My supervisors, Prof. John ffitch and Dr. Russell Bradford for providing stimulus during my research.
- My parents for their great support and love.
- The members of the Computer Science Department, relatives and friends for constructive discussions and help during the writing of this thesis.

The best time to plant a tree is twenty years ago. The second best time is now.

Chinese proverb

Contents

1. Introduction....9
2. Literature review and background....15
   - 2.1 Computer Music review....16
   - 2.1.1. Early history of Computer Music....16
   - 2.1.2 Signal Processing and Computer Music Review....17
   - 2.2.1 Signal Processing and Computer Music in the 20 century....18
   - 2.2.2 Centres and venues for research in Music Technology....20
   - 2.2 Parallel computing review....21
   - 2.3 Background....23
   - 2.3.1. High Performance Computing in Music Technology....23
   - 2.3.2 Goertzel algorithm....29
   - 2.3.3 Sliding Discrete Fourier Transform....30
3. Environment and methodology....33
   - 3.1 Environment....33
   - 3.2 General methodology of the parallel development and implementation of the SDFT and IDFT algorithms....42
   - 3.3 Analysis of the successive algorithm....48
   - 3.4 Analysis of the synchronous data copy of the SDFT algorithm....50
   - 3.5 Analysis of the asynchronous data copy of the SDFT algorithm....55
   - 3.6 Analysis of the asynchronous chunk data copy of the SDFT algorithm....58
   - 3.7 Analysis of the asynchronous data chunk for SDFT and input of the SDFT algorithm....61
   - 3.8 Analysis of the asynchronous chunk data copy of the SDFT values in the IDFT algorithm....64
4. Results and Evaluation....67
5. Conclusions and Future Work....71
6. Bibliography....74
7. Tests of variations of the SDFT algorithm....78

List of Tables

Table 1. Results of successive DFT and IDFT algorithms. 78
Table 2. Results of synchronous data copy of SDFT and IDFT algorithms. 79
Table 3. Results of asynchronous data copy of the SDFT algorithm. 80
Table 4. Results of asynchronous chunk data copy of input in the SDFT algorithm. 82
Table 5. Results of asynchronous chunk data copy of input and previous values of the SDFT in the SDFT algorithm. 84
Table 6. Results of asynchronous chunk data copy of SDFT values in the IDFT algorithm. 86

List of Figures

Figure 1. CSX600 processor [25]. 35
Figure 2. Execution units [25]. 36
Figure 3. Communications between CSC and host processors [25]. 38
Figure 4. Description of mono type * poly type [25]. 39
Figure 5. Description of poly type * mono type [25]. 40
Figure 6. Order of algorithms and data movements. 43
Figure 7. Image of storing complex number in mono and poly memories. 45
Figure 8. A synchronous order of storing and coping information in the Mono memory. 53
Figure 9. A synchronous order of storing and coping information in the Mono memory. 56
Figure 10. An asynchronous order of storing and coping chunks of information in the Mono memory. 59
Figure 11. An asynchronous order of storing and coping chunks of information in the Mono memory and within the chunk in the Poly memory. 62

List of Charts

Chart 1. Dependency of the time delay on the window size in the successive algorithms of the DFT and IDFT. 49
Chart 2. Dependency of time delay on the window size in the synchronous data copy algorithm. 54
Chart 3. Dependency of time delay on the window size in the results of asynchronous data copy implementation of the SDFT algorithm. 57
Chart 4. Dependency of the time delay on the window size of the asynchronous algorithm with chunk data copy of previous SDFT for calculation of the current SDFT..... 60
Chart 5. Dependency of time delay on window size. 63
Chart 6. Dependency of time delay on the window size of the asynchronous algorithm with chunk data copy of SDFT for calculation of the IDFT..... 65
Chart 7. Evaluation of SDFT algorithms. 68
Chart 8. Evaluation of IDFT algorithms. 69

Chapter 1

Introduction

The mysteries of Time have unceasingly engaged the human intellect. For many centuries men have mused on the nature of Time with a philosopher's eye; they have disputed Time's cause of uncontrollable changes, and attempted to catch the essence of Time. The sexagesimal numerical system originated by ancient Sumerians was passed down to Ancient Babylonians; this system is still used for measuring time. In the Renaissance, the period gave rise to modern science concerning nature and humanity. Time was comprehended no longer as an ephemeral substance, but became an essential trait of the Universe. There developed a full definition of Time's behaviour, which could be studied by scientists and thinkers. The German philosopher Immanuel Kant in [1] contests that Time is not derived from experience, but is a precondition of experience.

However, even today no explanation for certain of Time's features can be found in fundamental physical laws. Time still conceals many mysteries that researchers have attempted to explain. There are many thoughts about what Time is; Albert Einstein, one of the most influential scientists of all time, concedes in [2] that Space, Time and Substance interact continuously. Time nevertheless remains a mystery and an indivisible issue for humanity.

Scientists have not learned how to manage Time, yet they have studied how to take advantage of it, and computers are effective tools in the achievement of this goal. Computers have accumulated information and knowledge over many years. However, only in the last few decades has the creation of parallel processors led to a promising opportunity to develop real-time applications, and therefore to seize the impetus from the World of Computer Sciences.

The use of a concurrent process that communicates by message-passing has its roots in operating system architectures studied in the 1960s [3]. There are many cases in which the use of a single computer would be possible in principle, but the use of parallel systems is beneficial for practical reasons.

Depending on the scientific problem, parallel researchers use different kinds of architecture - heterogeneous or homogeneous multi-core computers. In the area of High Performance Audio Computing a vital aspect of research is Digital Signal Processing, which in the last decade has become crucial, in the light of time delay. This thesis is about a detailed description of the research, which has been carried out.

While computers have been advancing, developing into clusters, multiple processors and vector accelerators, there has barely been a change in the audio computing model. Recently a specific focus was proposed under the title of "High Performance Audio Computing (HiPAC)"; this research fits directly into this description.

Therefore High Performance Audio Computing (HiPAC) has become an essential research line in the twenty first century. This is due to the fact that efficient software is a significant problem in bridging the gap between the IT industry and its users in music technology. It is imperative that parallel programming researchers should create an advanced level of the HiPAC, as it is in danger of being left behind. The main focus is on HiPAC that exploits

the processing power of all processors in parallel to efficiently support audio programming in parallel computers.

Digital Signal Processing (DSP) is a very wide field, with a variety of applications, especially DSP used in multimedia. In essence, it has made a huge improvement in picture processing. In a comparison of picture and sound fundamental features, the sound technology is unambiguously more erratic and has high latency requirements. For instance, considering a delay in video processing, a program just repeats the previous image, and the eye smoothes the video, whereas in sound processing any delay will give a click sound. Consequently, the general view of information is damaged. Also, it should be noted that Audio Computing is lagging by a decade behind Image Processing owing to the lack of computer performance and now, with the ability to use multi-core computers, there is a chance to improve this notable area.

There are different approaches of frequency-domain processing of signal, which use spectral processing tools like: Discrete Fourier Transform, Short time Fourier Transform, Instantaneous Frequency Distribution, proposed by T. Abe in [4] and Sinusoidal Modelling. Victor Lazzarini in [5] has explored these methods from basic principles. These tools are used in music technology as a conversion from a time domain signal into a frequency domain representation. V. Lazzarini showed in [6] a variety of implementations of frequency-domain digital audio effects, as well as in Csound particularly.

For this particular research the choice was made to investigate the parallel DFT algorithm, owing to the fact that in literature one can more usually find the optimised algorithm of DFT, which is Fast Fourier Transform (FFT), yet it needs a certain number of transform size, which is power-of-two as well as other restrictions; transform size in the DFT analysis does not have this limitation. Although, David John Wheeler suggests a FFT algorithm with complexity of $O(N*logN)$, this algorithm will not feature in our consideration.

One approach of this research is to develop a real time parallel DSP algorithm. A null hypothesis is proposed to convert from the time domain to the frequency domain as well as the inverse operation within 10 milliseconds, as this limit is imperceptible for experts in audio technology. For the DFT inverse calculations were performed by using the Inverse Discrete Fourier Transform (IDFT). Due to the fact that a larger transform size implies precise values in the frequency domain, it is imperative to find a balance between transform size or window and time delay, which takes calculations (DFT and IDFT conversions) of one point. In the analysis of parallel algorithms, more attention is usually paid to communication operations than computational steps. Therefore, communications were the purpose of this investigation.

In addition to these approaches, the input signal was stored with double precision, which takes for each point 8 bytes of memory, instead of the usual 4 bytes. This method of information storing was adopted for the purpose of making the rounding error marginal. From the audio point of view this error gives rise to noise. So, significant errors in single precision give noise approximately 2 minutes from the beginning, in comparison with double precision calculations, which give noise in 2 hours.

Concurrent programs are more challenging to develop due to parallel architecture introducing implicit kinds of errors and a variety of restrictions in coding, which influence program performance. Communications between concurrent processors and shared memory were the greatest obstacle in the project; due to the form of parallelism, which is data parallelism. Data synchronisation emerges in parallel or distributed computing only if a parallel algorithm requires processes of execution calculations on data in synchrony, each node keeps its own copy of data. Data must be copied back and forth coherently with one another. Here, for calculation of SDFT data has to be transferred from different places in shared memory and only then can calculations be done, consequently this parallel problem arises in the current project.

The project is concerned with a parallel slowdown. Owing to the procedure of data sending it is the most expensive in terms of time due to the physical features of the communication tools. It is necessary to find the most appropriate size of sending information from mono to poly processors and back. If the size is wrong then the accelerator needs more time for communications and eventually communication overheads arise. In this case, it is necessary to find equilibrium between the amount of data, which has been sent, and the time the communication needs.

In order to achieve real time for a parallel program it is imperative to choose a computer, which is appropriate for calculations and communications in the introduced algorithm. The Clearspeed CSX600 accelerator was chosen for the execution of a SDFT parallel implementation. Clearspeed has both a mono execution unit and a poly execution unit. The Poly unit contains 96 processors; each processor has its own poly memory. As the first part is a fine-grained algorithm, nodes must communicate with shared memory many times, whereas the second part of the algorithm was IDFT, where nodes needed to communicate with shared memory once per loop cycle and the amount of data which was required was larger than in SDFT. Also we introduced an example of parallel program with straight calling data from shared memory. This was done due to the specifics of audio processing requirements, such as low latency and the considerable amount of data, which must continuously be calculated. We needed to answer a crucial question concerning which implementation is faster for the fine-grained vector accelerator: whether to copy a massive amount of data from shared memory to poly memory or to do the calculation on shared memory. This question emerged due to each processor element typically having faster access to its local memory than access to shared memory, however this statement was checked by developing and implementing 3 algorithms of IDFT as well as analysing these results in the chapter “Methodology and Environment”.

Several algorithms were developed, implemented and analysed, and the results of each were compared. The aim was to find the fastest DFT algorithm as well as the fastest IDFT. More than four hundred tests we performed in order to find the balance between loaded

data and delay. Detailed tables with test results can be found in the appendix of this thesis. In the chapter “Background and Literature Review” we show the links between computer music and computer science throughout the history of these areas’ development. The background to this research is also demonstrated in this chapter. There are two sections in the “Methodology and Environment” chapter: “Environment” and “Methodology”. The Environment section contains explicit review of the Clearspeed architecture and the software provided with the Clearspeed vector accelerator, where the project has been programming. The Methodology contains a description of the developed algorithms, and charts of experiment results, where data has been analysed. In the “Results and Evaluation” chapter we evaluate the results from all algorithms in order to find the fastest algorithm, which conforms to the requirements. The conclusion, applications, results and future work can be found in “Results and Future Work” chapter.

Chapter 2

Literature review and background

The structure of this chapter evolved by virtue of the fact that to the best of our knowledge there is no documentation that contains a sophisticated history of modern computer music in the 20th century. We divided this chapter into three main sections: a Computer Music review, a Computing review and the Background to this research. In the first two sections we attempt to show how the areas of Computer Music and Computing have been closely interwoven with each other throughout their history. The Background is the last section in this chapter, where we talk about the research, which was done previously, and how it is connected to our work.

2.1 Computer Music review

2.1.1. Early history of Computer Music

The Greek philosopher Pythagoras first examined music as a science in the sixth century BC, when he considered the beauty of a sound, and found a harmonic overtone series on a string. Many significant thinkers and scientists of Ancient Greece and Rome discovered that sound can be represented as a mathematical equation. Among them were Aristotle and Galileo Galilei. Considered to be a father of acoustics, the music theorist Marin Mersenne contributed musical tuning and *"The first absolute determination of the frequency of an audible tone (at 84 Hz) implies that he had already demonstrated that the absolute-frequency ratio of two vibrating strings, radiating a musical tone and its octave, is 1:2."* in his work [7], [8].

John William Strutt, Baron Rayleigh, was the first to summarize outstanding contributions to music theory in his book [9] as well as to look at sound from both mathematical and physical points of view. J.W. Strutt describes in detail the most significant discoveries and observations of research that influenced sound theory in the 18th and 19th centuries. In the first volume of [9] Baron Rayleigh looks at sound from a different point of view: *"The sensation of sound is a thing sui generis, not comparable with any of our other sensations"*. Here J.W. Strutt summarises many distinguished physical experiments, which were carried out by François Arago and others, about the velocity of sound. Jean-Daniel Colladon and Jacques C.F. Sturm were also investigating the propagation of sound. Baron Rayleigh also summarizes many other experiments regarding the intensity of sound and the generation of a musical note by *"revolving a wheel whose milled edge is pressed against a card"*. He also describes in detail a "Siren", a remarkable voice-production invention of Charles Cagniard de la Tour. In this volume J.W. Strutt also develops his own investigations and observations, about recognition of sound, analysis of notes and many other fundamental tools of sound.

In the second volume of [9] sound is treated mathematically by Lord Rayleigh. He combines vibrations of air with an equation of continuity and Lagrange's theorem, Poisson's equation, law of reflection, and the Fourier transform. He describes a first theoretical explanation of the velocity of sound experiments made by Newton. This significant book contributes copious points from mathematics and physics to the theory of sound, and in it J.W. Strutt opens up a new era of sound.

The twentieth century was a crucial moment for audio technology and acoustics; it was an exuberant time of sound applications in many other areas of science. The first scientist to use acoustics as an individual science was an American physicist, Wallace Clement Sabine (1868 – 1919), a founder of architectural acoustics. W. C. Sabine improved acoustics in the Fogg Lecture Hall in 1895 [10] and collaborated in the building of Boston's Symphony Hall in 1900 [11].

2.1.2 Signal Processing and Computer Music Review

In essence, signal processing is a field where applied mathematics is combined with electrical engineering, and which considers analysis or operations of signal in discrete or continuous time in order to conduct an operation of signal. A signal can be sound, image, radio, electrocardiogram and many other things. Each kind of signal imposes constraints on features of an operation, which deals with this signal. Sound processing is a signal processing with imposes constraints on latency. The most common examples of operations' applications on sound analysis are: spectrum analysis, filtering, smoothing, modulation, and wavetable synthesis.

The area where mathematics, computer science and music composition are joined together is called computer music. It includes several new technologies: digital signal processing, music synthesis, computer composition, sound design, acoustics, psychoacoustics and

many others. In the twentieth century, when personal computers became an everyday phenomenon, and with increased numbers of home recording systems, the definition of computer music changed to mean everything that is created by using a computer.

2.2.1 Signal Processing and Computer Music in the 20 century

In the late forties the first few stored-program computer machines with floating point units were built in the United Kingdom. At the Computer Laboratory in Cambridge one of the world's first practical stored program electronic computers "Electronic Delay Storage Automatic Calculator" (EDSAC) was built in 1949. The world's first taught course in computer science was also offered here four years later. At that time one of the earliest tasks for computer machines was the calculation of Mersenne primes, which took approximately nine hours to run. Also, at the University of Manchester, the Ferranti Mark 1 was installed. A machine with a nickname - "Babe" [12] - this computer faced another first calculation: that of the highest factor of a number.

In 1948, the first scientist to make a sound on the computer was Christopher Strachey, who was a mathematics master at Harrow, a private school in London. There are two stories about the first sound computer program. The first is that Strachey wrote a program and ran it on EDSAC and surprisingly, the computer started producing a sound: the sound of the national anthem. From that time, in the mid-fifties, scientists started to view computer things differently – musically. The second story is that the first musical use of a computer was at Victoria University in Manchester, with the first musical rhythm of "Baa Baa Black Ship" [12]. Also, Christopher Strachey visited Bell Laboratories and other research centres in the United States. He worked on both Ferranti and EMI Groups. Later he worked at the University of Cambridge and in the middle of the sixties he became the first director of the Programming Research Group at the University of Oxford, where he later became the first Professor of Computer Science [13]. Christopher Strachey launched a new era of computer music.

Nevertheless, it is an American researcher, Max Vernon Mathews, who is considered to be the father of computer music. His interests included audio processing, synthesis, and many others aspects of computer music. After achieving his ScD, he started work at Bell Labs (formerly known as AT&T Bell Laboratories and Bell Telephone Laboratories), where the first computer music language MUSIC I was created in 1957. Max Mathews started a new epoch of digital sound generation by creating a family of MUSIC languages. Afterwards he created MUSIC II and MUSIC V. Also Max Mathews was the first to teach a course of computer music at Stanford University, where he later became a professor [14].

In the eighties John Robinson Pierce, an American scientist who worked at Bell Labs from 1936 till 1971, created many outstanding contributions to microwave technology, and radio and satellite communications. For the latter he was awarded an Edison Medal. At Bell Labs Pierce also worked in collaboration with Max Mathews [15]. After resigning from Bell Labs, he became professor of electrical engineering first at the California Institute of Technology and then later joined the Centre of Computer Research in Music and Acoustics (CCRMA) at Stanford University, where he did outstanding research in computer music. The most prominent creation of J. Pierce and M. Mathews was the Bohlen–Pierce scale, an alternative to the octave musical scale [16].

American scientists were not alone in contributing their research to the field of computer music. Many researchers and composers from the Old World introduced a variety of significant investigations as well. One of the earliest Europeans, who started work at Bell Telephone Laboratories was Jean-Claude Risset, a French composer who worked with Max Mathews at Bell Labs in the sixties. Risset worked on brass synthesis, pitch paradoxes, synthesis of new timbres and the sonic development process. He also wrote many articles about computer music. Risset was chair of the computer department at IRCAM in the late seventies and worked as a composer at the Media Laboratory at MIT. He received a considerable number of prestigious prizes and grants [17].

2.2.2 Centres and venues for research in Music Technology

IRCAM (*Institut de Recherche et Coordination Acoustique/Musique*) has developed many contributions, which have influenced the computer music world. Founded in 1970 by French President George Pompidou and Pierre Boulez, it was opened in 1977. IRCAM has nurtured many remarkable researchers, among them John Chowning, Luciano Berio, Pierre Boulez and Jean-Claude Risset. Many notable research concepts, music languages, environments and technological contributions have been created there. This institution still remains a home for many contemporary researchers and composers [21].

From the fifties significant scientists worked at Stanford University's Center for Computer Research in Music and Acoustics, which was founded by John M. Chowning. The main research is aspects of computer music, which are brought together from music, computer science, physics and engineering areas. There are groups like "Music, Computing, and Design", "Signal Processing", "Music in Virtual Words" and many others. In the beginning it was created as a high end research centre *"A multi-discipline facility where composers and researchers work together using computer-based technology both as an artistic medium and as a research tool."* [18]. Here Chowning worked on frequency modulation (FM) synthesis algorithm (1967), which he invented by an accident of testing a variety of vibrato: *"Chowning found that when the frequency of the modulating signal increased beyond a certain point, the vibrato effect disappeared from the modulated tone, and a complex new tone replaced the original."* [19]. Chowning patented his discovery in 1975 and licensed it to the YAMAHA Corporation. The first commercial device with FM synthesis implementation on it was a digital synthesizer DX7, which came into the world in 1983. Yamaha patented their hardware implementation and established a monopoly in the market of musical hardware technology [20]. Chowning's invention of FM synthesis is a strong example of the long tradition of transferring research from laboratories to industrial applications.

And lastly, ICMC (*International Computer Music Conference*) - the multi-disciplinary nature of this conference, which covers composition, computing and digital signal processing, makes it one of the most prominent widespread proceedings where researchers now show their inventions in music technology and audio processing [21].

2.2 Parallel computing review

In this subsection we are going to look at parallel computing for several reasons. Firstly, we want to show how music technology is closely interwoven with concurrent computing. Secondly, we attempt to discover what kind of parallel computer will be commonly widespread in the near and distant future, because this acquisition of a future general parallel computer will enable the construction of a high performance computer for the purposes and requirements of music technology. This knowledge will be a key factor in producing high quality software under the title of the HiPAC. Thirdly, we attempt to elucidate the heading of audio software for the music community and share gained experience from our research in the real-time parallel algorithms and their realisation in Music Technology.

Improvements in modern audio technology would not occur without the creation and modernization of a strong engineering, scientific and analytical tool – the computer and its development to the parallel computer. The idea of an Analytical Engine, the first mechanical general purpose computer, was conceived by Charles Babbage [22]. Babbage's invention was remarkable and initiated the whole of computer development [23], [24].

The origin of the multiple instructions and multiple data (MIMD) parallel computer came from Charles Babbage's creation the Analytical Engine [29], [30]. The first company to start the development of the mass-produced computer was IBM in 1954, supporting the

high-level programming languages FORTRAN, LISP and MUSIC. Under the "IBM 704" project there were 704 researchers. Among them were pioneers in computer science, including the well-known computer architect Gene Amdahl and the designer of FORTRAN John Backus [31], [33]. In 1962 using the IBM 704 computer American physicist John Larry Kelly and Max Mathews synthesized speech for the first time. This was a song "Daisy Bell". It was one of the most illustrious moments at Bell Labs [32].

During the later fifties and sixties the foundation for the modern parallel computer was created. A trend became apparent that whenever a new architecture design was introduced, just a few computers were produced, which confirmed the fact that parallel computing was built only for scientific computations. In 1965 Gordon Moore, an American scientist, published Moore's Law. It concerned a long-term trend in processor development. Moore said that over every 18-24 month period, the number of transistors and, consequently computer performance, would be doubled [38]. Moore's Law for desktop computers stopped working at the beginning of the 2000's when chips reached their physical limits in processing speed. Also for HiPAC Moore's Law used to work only partly because of the differences and requirements of audio processing, described in the "Introduction" chapter. In 1966 Michael J. Flynn introduced four classifications of parallel computers and programs, which are called Flynn's taxonomy. This classification is based on a number of concurrent nodes and the data flow needed for implementation of an algorithm or available in a classified computer [39]. In the HiPAC a Single Instruction Multiple Data classification is used, or its subclass called a Single Program Multiple Data, which lets the data flow asynchronously.

2.3 Background

2.3.1. High Performance Computing in Music Technology

Many significant challenges in audio processing have hitherto been avoided as being computationally prohibitive. There have been many positional papers and comments about the advantages of using multi-core computers [3], [52]. In [51] it was suggested using multiprocessors only for the calculation of a specific area of tasks and the author of [51] did not believe that ways of programming would ever change, a view which is very debatable, but if it were true then it would be very challenging to convert general users like music composers towards parallel programming. Also it was proposed to find another style of computing due to the fact that for programmers these changes are very challenging [51].

I believe that looking back into the history of how computers were created and enhanced it can be seen that there was always a temptation to develop a faster computer by multiplying processors and using those processors in parallel. Therefore it would not be any new way of making a computer faster to make it parallel. Peter Van Roy in [50] recalls dataflow programming as well as designing decentralized systems. An overview of many-core processors in the past and present was proposed by David Wessel in [49]. Where a different kind and level of music software was compared, highlighting the most popular current software (CSOUND, Max/MSP, FAUST and etc.), he said that a huge majority of these languages do not support parallelism. On the other hand there have been several concurrent developments from the seventies and the eighties (IRCAM Signal Processing Workstation, multiple Motorola 56000-based Audio Media Nubus Cards, DigiDesign's DSP-Farm). The author discusses the architecture of parallel computers to be in use only for computer composers, saying that it is unclear whether it should be homogeneous or heterogeneous architecture but it is clear that the most must be made out of concurrency to make real-time practical [49]. These papers are positional, so the authors do not present any research results.

John ffitch et al in [48] describes how crucial High Performance Computing for Audio Technology is. The main necessities here are real-time processing as well as low latency. This is due to the imperative of improving sound processing, audio synthesis and music composition, where deep investigation needs to be done in a digital audio stream. Also they propose different ways of researching HiPAC, including accelerators and multi-core computers: *"...the study of new advanced processor architectures to enhance audio synthesis, processing and music composition…*" John ffitch et al emphasize significant differences between High Performance Computing and HiPAC: *"Rather than considering the use of supercomputers and mega clusters we are concentrating on what will within a relatively short timescale be consumer grade hardware. The emphasis has to be on affordable low latency real-time processing".* Also they claim that getting further speed from a one core consequently requires high power consumption. As a result, it is not profitable in the light of high energy expenses and it is a source of noise creation in music applications. The authors also affirm their hypothesis in audio processing definition, that it is serial, as well as that the implementation of audio processing involves many identical independent calculations, so the structure of an algorithm is highly data parallel. Some challenging tasks are defined due to the fact that calculations can be done in real-time, and which require highly powerful computers, which are going to be built in the next generation for general purpose.

In [48] there is also included a condensed investigation into parallel hardware. It describes the differences between SIMD (Single Instruction Multiple Data) and MIMD (Multiple Instruction Multiple Data) models. The aim of the investigation in [48] is to find out how to design general parallel audio computers in the light of parallel audio processing needs. Nowadays, fine-grained concurrent architectures of SIMD look like a vector addition of a general CPU, and also SIMD includes graphic accelerator cards; these improvements are important to all computer users. A distinction of the new SIMD accelerator is that it is developed on one chip and the architecture is monolithic. Also in order to support and make faster performance growth chip producers have already implemented the procurement of

many units for supporting new concurrent processors. Manufacturers have already developed chips with a performance of one teraflop, but they will be produced commercially only in 10 years' time.

Nowadays a serious market has been built for SIMD parallel architectures with the most noticeable SIMD accelerators being the Tesla system series from nVidia and the Clearspeed CSX600 and CSX700. The Tesla accelerator introduces the first general-purpose product, not specified for graphics. Secondly, the floating-point Clearspeed CSX600 and CSX700 accelerators are presented. In CSX600 each card contains one chip and the newest version of CSX700 unites two CSX600 processors. In essence, each chip contains 96 floating point processors in SIMD style. The main advantage of Clearspeed is low power consumption, which is 30W per card. The most recent release of CSX700 is significantly cheaper due to the implementation of a PCIe-based microprocessor. This fact makes Clearspeed CSX700 vie with nVidia Tesla accelerators. Detailed studies were also done on the Clearspeed processors during the programming, and they are presented in the chapter of "Methodology and Environment", which also contains detailed information about Clearspeed architecture.

Audio processing imposes interesting challenges on computing. So, there is no general rule on how much speed up can be obtained from parallelism. Amdahl's law [47], states the following formula:

$$SpeedUp = \frac{1}{(1-S) + \frac{P}{S}}$$

Where, S is a sequential part of the algorithm and P is a parallel part of the algorithm and also $P+S=1$. Speedup is a time span of a ratio of sequential portion's time span plus time span of parallel part takes. It is mainly estimates of overly conservative value in parallel computing. Hitherto audio processing is without any parallelism evaluation, due to the fact that it has high concurrency only in data, and the complexity of calculation is low, consequently the concurrent part of a task will mostly depend upon hardware features. There-

fore, new paradigms of audio processing must be created. The authors argue in [48] that audio processing must pay attention to computation requirements. *"Even if they are time-consuming today, they will not be in only a few years, so that we* ***must*** *start investigating them now...in the HiPAC program the study of no-compromise algorithms – rather than make simplifications to an algorithm purely for reasons of slowness, HiPAC considers such algorithms in as pure or "ideal" a form as possible, especially where that ideal form may lead to musically useful and novel behaviour."*

It is shown as an example that the Sliding Phase Vocoder can make use of HiPAC [46]. The implementation of SPV is based on SDFT, and they show that the process of SDFT calculations is essentially parallel between the bins and they expect that implementation SDFT on the Clearspeed microprocessor will be yielding in terms of real-time performance and latency.

Very new impressive investigations have been done by Yann Orlarey at Grame [45]. The first investigation is a Jack audio server with low latency, which was previously based on a sequential machine and it was upgraded to a data flow model with usage of a lock-free programming technology on multi-core computers. The Jack system works similarly with a server with natural parallelism, when clients do calculations concurrently even if those depend on shared variable. An activation model is needed to activate clients simultaneously and accurately.

Also the authors underline that Jack has sequential and parallel components. If a parallel component exists it means that clients can be executed concurrently on different processors. The data-flow model helps here to describe this system: if all inputs become accessible then a node in data-flow graph is able to run. The activation counter is used by each client to find out the number of input clients on which it depends. The activation is transferred from client to client during all executions of the server. The authors suggest the way of sequential graph execution, utilizing pipeline techniques. Their suggestion is dividing an audio buffer into components. But this division means that during running smaller

buffers there will be more context-switching between processors, which causes a time delay, so it is imperative to find the equilibrium between the size of divided buffers and the number of available processors in the project [45].

Another project is a language for High Performance Audio Applications – Functional Audio Stream or "Faust". Faust is designed to be implemented efficiently as C/C++ plug-ins for audio applications [44]. It is used for real-time signal processing with transformation into C++ language; also it does not depend on any DSP library. A programming model of a language is a combination of mathematical semantics with block-diagram syntax. Recently, the authors started a design of a parallel compiler and they show an example of their work, where they found it challenging to balance data copy and communication overheads. Thus efficiency is still an essential domain of the research for achieving real-time performance. The work done by Y. Orlarey in [45] is promising; however, unfortunately, no results of their experiment are shown. Thus it is impossible to compare with the results of our project.

A parallel implementation of a partitioned convolution and a non-negative matrix factorization (NMF) were done in The Parallel Computing Laboratory at the University of California Berkley by Eric Battenberg, David Wessel and Adrian Freed in [43]. This NMF is a component of a music information retrieval (MIR). A parallel algorithm of a partitioned convolution a frequency delay line (FDL) was used in [42]. The main approach was to reach real-time requirements and to harness GPU cards in audio processing.

Eric Battenberg in [43] shows an algorithm to speed up NMF. Implementations were done on consecutive and concurrent ways in order to optimize and analyse the algorithm towards real-time. The author presents performance results of an execution time for several implementations of NMF on various architectures. It was done in order to fully use the potentialities of a multi-core CPU and highly parallel graphic processors.

Versions were written in the following languages: MATLAB, C, OpenMP and CUDA. The MATLAB implementation was done on Intel MKL BLAS Core2 Duo T9300 CPU while CUDA was run on GTX 280 GPU and 8600 GTS GPU. Results have shown that the CUDA implementation gave the best execution time; it runs over 30 times faster than the implementation of MATLAB. And the OpenMP implementation executed in Core i7 920 runs in 7 times faster than the MATLAB version. However, the authors say that to write a program in CUDA takes 10 times longer than in OpenMP. Therefore CUDA can be used only in massively intensive calculations.

Due to an implementation of an effective parallel code being the hardest problem in the utilization of parallel systems, the authors suggest looking at an idea, "Selective, Embedded, Just-In Time Specialization", (SEJITS) by Catanzaro et al. [41]. The idea of it is that a computer music composer will write an example of non-negative matrix factorization in a scripting language like RUBY. The composer must also express operations in time-frequency representations. This script code is a portable code, which would be run on SEJITS and if a parallel implementation of the code is accessible, so the system generates an optimal code on appropriate layer language like C, C++, OpenMP or CUDA. As a result of the project, the authors recommend SEJITS since this system is the most appropriate way of using parallel systems for general users including computer music composers. But they also say that SEJITS must have some features like meta-programming and introspection. Developers must also create efficient criteria for the layer programming. The results of the NMF show the vigour of modern GPUs in order to supply an essential acceleration. Lastly, Battenberg et al say that the example of Music Information Retrieval or MIR systems of SEJITS has not given real-time results and they cannot change it at all [43].

2.3.2 Goertzel algorithm

In this section we shall summarise the main sources of the SDFT algorithm: a Goertzel algorithm and the SDFT algorithm. We shall discuss the research, which was the main motivation for this work.

The Goertzel Algorithm [37] is a method for presenting a Discrete Fourier Transform from time-domain to frequency-domain signals. This technique reduces communication costs in $O(n^2)$. This algorithm was created only for frequencies with particular features. The main difference of the Goertzel algorithm from the DFT algorithm is that it depends on predetermined frequency and some values from the time-domain frequency. As it says in [36] and [35], the Goertzel algorithm computes a complex DFT value for every N input element. Luckily, audio signal processing satisfies these requirements because the time interval between bins is fixed. So for calculation of each bin it is necessary to do only two additions and one multiplication and therefore this method performs fast calculations.
This algorithm calculates the *k-th* bin of DFT with window size N, using the following equation:

$$F(x) = \sum_{j=0}^{N-1} x(j) e^{-2\pi ijk/N} \qquad \textit{Equation 1}$$

Where input frequency *x(j)* must be an integer, also index *k* is in an interval: $0 \le k \le N-1$. The output of the DFT algorithm and the Goertzel algorithm is the same. Even if it is chosen to use the Goertzel algorithm instead of the DFT algorithm, the first value must be defined by using equation 1, and then utilize SDFT with all its advantages.

In the Goertzel algorithm, a *z*-domain transform function is:

$$H(z) = \frac{1 - e^{-2\pi jk/N} z^{-1}}{1 - 2\cos(-2\pi kn/N) z^{-1} + z^{-2}}$$

where a single z-domain zero is on $z = e^{-2\pi jk/N}$ and conjugate poles at $z = e^{\pm 2\pi jk/N}$. The zero/pole pair cancels each other at $z = e^{-2\pi jk/N}$ [34].
For the Goertzel filter the time-domain different equations are:

$$v(n) = x(n) + 2\cos(2\pi k/N)v(n-1) - v(n-2)$$

$$y(n) = v(n) - e^{\frac{-2\pi jk}{N}} v(n-1)$$

In fact if it is necessary to calculate S different inputs, it is necessary to implement the Goertzel algorithm S times. Also, one of the main advantages of this algorithm is N does not need to be a power of two. This algorithm is also used for recognition of Dual-Tone Multi-Frequency (DTMF) signals, produced from a telephone by pushing buttons on a keypad [34].

2.3.3 Sliding Discrete Fourier Transform

An algorithm of Sliding Discrete Fourier Transform (SDFT) computes DFT with exactly the same precision as the Goertzel algorithm. The Goertzel algorithm calculates a DFT for every single element of the input array. As opposed to the Goertzel technique, the SDFT algorithm can estimate more signal bins with less data and computation. Also in SDFT the input rate is the same as the output rate [34]. In essence, the SDFT algorithm computes the current value of DFT in time=t and with a frame = N, when a new input sample comes, the frame moves by one element in the time =$t+1$ algorithm calculates a new quantity considering the new element of input and the previous value of SDFT [34].
The DFT starting at time t with window size = N can be represented as follows:

$$F_t(k) = \sum_{j=0}^{N-1} x_{j+t} e^{-2\pi ijk/N}$$

where $F_t(k)$ is a *k-th* value of SDFT in frequency domain. The algorithm of SDFT in the time $=t+1$ depends on the value at the time$=t$:

$$F_{t+1}(k) = \sum_{j=0}^{N-1} x_{j+t+1} e^{\frac{-2\pi ijk}{N}} =$$

$$\sum_{j=1}^{N} x_{j+t} e^{\frac{-2\pi i(j-1)k}{N}} =$$

$$(\sum_{j=0}^{N-1} x_{j+t} e^{\frac{-2\pi ijk}{N}} - x_t + x_{t+N}) e^{\frac{2\pi ik}{N}} =$$

$$(F_t(k) - x_t + x_{t+N}) e^{\frac{2\pi ik}{N}};$$

whereby the window always moves only by one bin. Not surprisingly this algorithm exists, due to the fact that when the analysis window moves by a sample the most bins of discrete frequency in the window will be the same. Most recently, SDFT implementation issue was highlighted by John ffitch et al in [28]. This paper was the initial point of this research along with [48], [27]. Also they say it is not necessary that the analysis window remains power of two.

In [28] John ffitch et al consider the essential question about sliding a window. They explain that in order to minimise blur and increase frequency resolution an envelope window to the sample period is applied. Therefore, it is not possible to do it in SDFT in time domain but it can be done after applying SDFT. So windowing is used in frequency domain. This windowing is discussed only for the purpose of showing integration of SDFT into Csound.

Thanks to the work of Richard Dobson and Victor Lazzarini, Csound has a well-built streaming Phase Vocoder. John ffitch et al. in [48] explain the application of SDFT in Csound: *"The process here is to construct a new DFT frame when sufficient samples have been obtained, with a restriction that this cannot be more than once per k-rate frame. Internally the f-variables have a structure to maintain the bin values, window size and type and various housekeeping data. The SDFT implementation in Csound reuses this struc-*

ture, so from an elementary user point of view the introduction of the sliding option has no syntactic change." Basically, the SDFT is a technique with an overlap of 1 bin [28].

In order to reconstruct the signal it was proposed to use a fundamental Discrete Fourier Transform:

$$x_t = \frac{1}{N}\sum_{j=0}^{N-1} F_t(j)e^{2\pi ijt/N} \qquad \textit{Equation 2}$$

In equation 2 each frame was as a representation of a single sample because there is a transform for each sample. It is debatable which sample each frame represents. This question means that latency depends on choosing the right sample in the frequency: there are non-latency versions, but very expensive, or other versions which seem very complicated.

The authors say that this process of transformation is not fast and it is necessary to investigate SDFT parallel algorithms in order to achieve real-time performance. They also say that they have achieved *"out-of-real-time"* implementation and further research is necessary in the field of High Performance Audio Computing [28]. This paper was one of the main motivations for this research, owing to the researcher's belief in the strong future of parallel computing.

Chapter 3

Environment and methodology

3.1 Environment

Nowadays there are many discussions about supercomputers for audio processing, and it is not clear which kind of architecture fits best for a particular audio processing task. In order to get a better understanding of the actual needs of parallel algorithms for audio processing it is necessary to take some steps. The first step is to create a parallel algorithm, secondly, implement and analyse it, and lastly, to conduct experiments in order to find out how changing programme variables affect the time delay. A push must come from the area of HPC by computer scientists and mathematicians in order to give freedom in music science promotion.

It is reported in the “Literature Review” chapter about the general purpose GPU, which is produced by NVIDIA. These kinds of processors were not a part of this investigation because they were not readily available when this research was started. An explanation is

also given regarding the research done for non-negative matrix factorisation on several parallel computers and utilising several parallel languages and extensions. Researchers of this project believe that the Clearspeed CSX600 computer is an appropriate machine for audio processing. Therefore, the main interest of the research was to develop, implement and analyse the algorithm of SDFT on the Clearspeed CSX600, which was chosen because it has 96 floating point vector processors and the power consumption of parallel processors is less than 30W per card. So, the null hypothesis was proposed to investigate whether it is possible to develop the SDFT algorithm, which can fit in the Clearspeed CSX600 machine, and to get real-time performance out of this parallel algorithm.

ClearSpeed Technology Ltd. was founded in 2001. There are two offices in Bristol, UK and San Jose, California. The company mainly creates accelerators and vector parallel processors, which are used to perform tasks with a large data set, high accuracy or in a short period of time. The company has concentrated on achieving less power, density and heat problems on HPC [26]. Therefore, this machine was chosen in order to develop, implement and optimise the SDFT algorithm.

To confirm the proposed hypothesis studies were performed on the Clearspeed parallel computer. Basic subjects for achieving successive results were: Programming on C^n parallel extension of C language, a Clearspeed architecture overview, acquaintance with C^n Standard Libraries, and an acquaintance with debugging in GDB. A complementary research problem was the studying of properties of a given parallel system.

Clearspeed provides essential material about a general comprehension of the CSX600 parallel processor architecture. The most relevant was information about the execution units of the processor. The authors introduce the main concepts of parallel processing, particularly, SIMD parallelism; and they demonstrate different features of C extensions, which are used by the CSX processor.

Firstly, in [72] the CSX600 processor architecture was described. Figure 1 from the [72] is a high-level image of the architecture of the CSX600 processor.

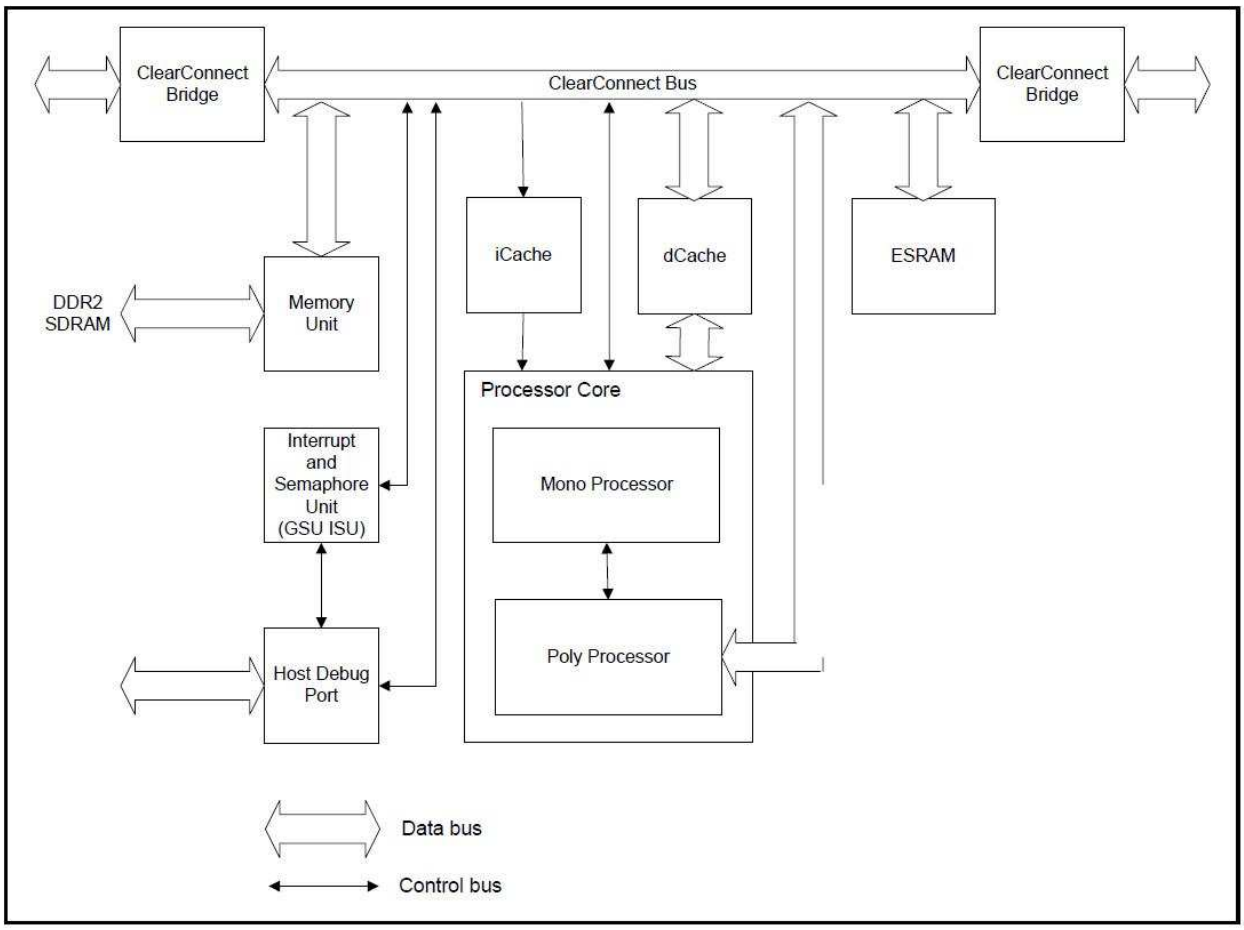

Figure 1. CSX600 processor [72].

The parallel machine is composed as a Single Instruction Multiple Data multithreaded processor core, an embedded SRAM with integration onto a single processor, high-speed interfaces and an external DRAM interface. An interconnection between all subsystems on the chip uses a ClearConnect Bus on the chip network.

From a programming point of view, the central parts of the parallel processor are execution units. There are two central parts: the mono execution unit and the poly execution unit which is an array of processing elements and each instruction is executed by either the mono execution unit or the poly execution unit. A mono execution unit contains SRAM, which has 256 Kbytes of memory. The vector processor core includes 96 processor elements (PE) and each PE can execute parallel, addition and multiplication operations.

There is also support for a fully pipelined operation with execution of one instruction per cycle [25].

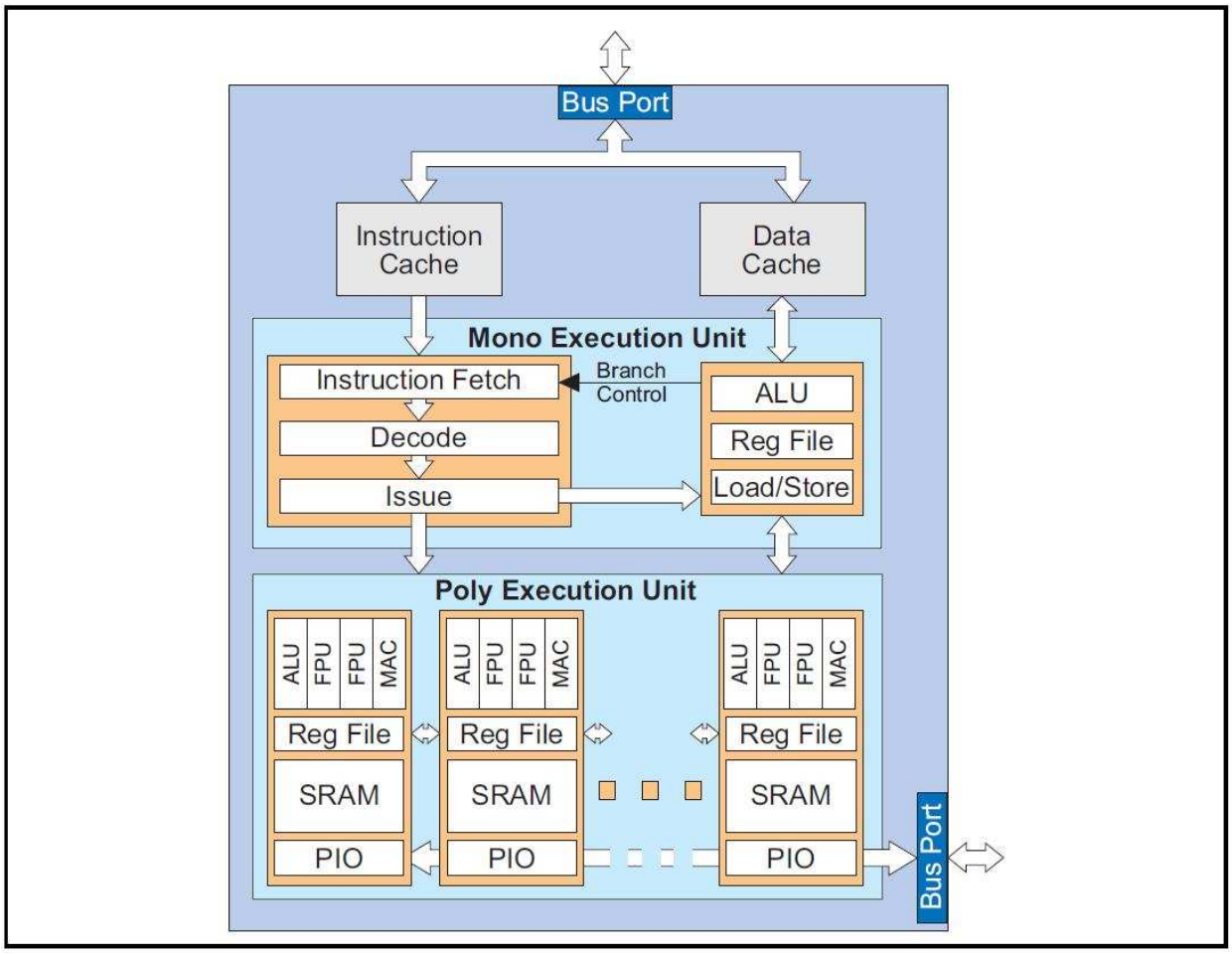

Figure 2. Execution units [72].

The high performance of the CSX600 processor comes when the data is processed concurrently in the poly execution unit. As stated above, the poly execution unit is an SIMD array with 96 processor elements, where PEs are connected as a vector array by the connection path. This means that each PE computes the same programming code but on different data. Each processor element contains 6 Kbytes of SRAM, a register file and an ALU (arithmetic logic unit), 32 + 64 bit FPU (floating point unit) [72].

Secondly, a programming model is described and the C^n concurrent language, which is a C language extension. As shown, there are two execution units: poly and mono. Therefore,

there are mono and poly memories with two different memory spaces. These spaces have different basic types of variables: a mono variable, which has only one instance and is stored in the mono memory; and a poly variable, which stores in each PE with different values on each processor. In programming for CSX processors multiplicity specifies were introduced, which let a programmer direct the domain, where the variable will be stored. It is possible to use poly variables in order to specify PEs, on which the code will be executed.

Developers of Clearspeed showed the basic examples of using C^n language. SDK includes all necessary tools for writing, compilation, debugging and running the code on Clearspeed accelerators [72].

They showed a good description of the process of running a program on CSX processors. An executable file can be run on the processor, this demands that while a program is running on a host computer it is loading executable code to the CSX processor and after that communicates with it. Generally, the program runs on the host computer and part of it runs on CSX processors. Communications between the CSX component and the host computer can be done by using a device driver. This module provides input and output services between CSX and host processors [72].

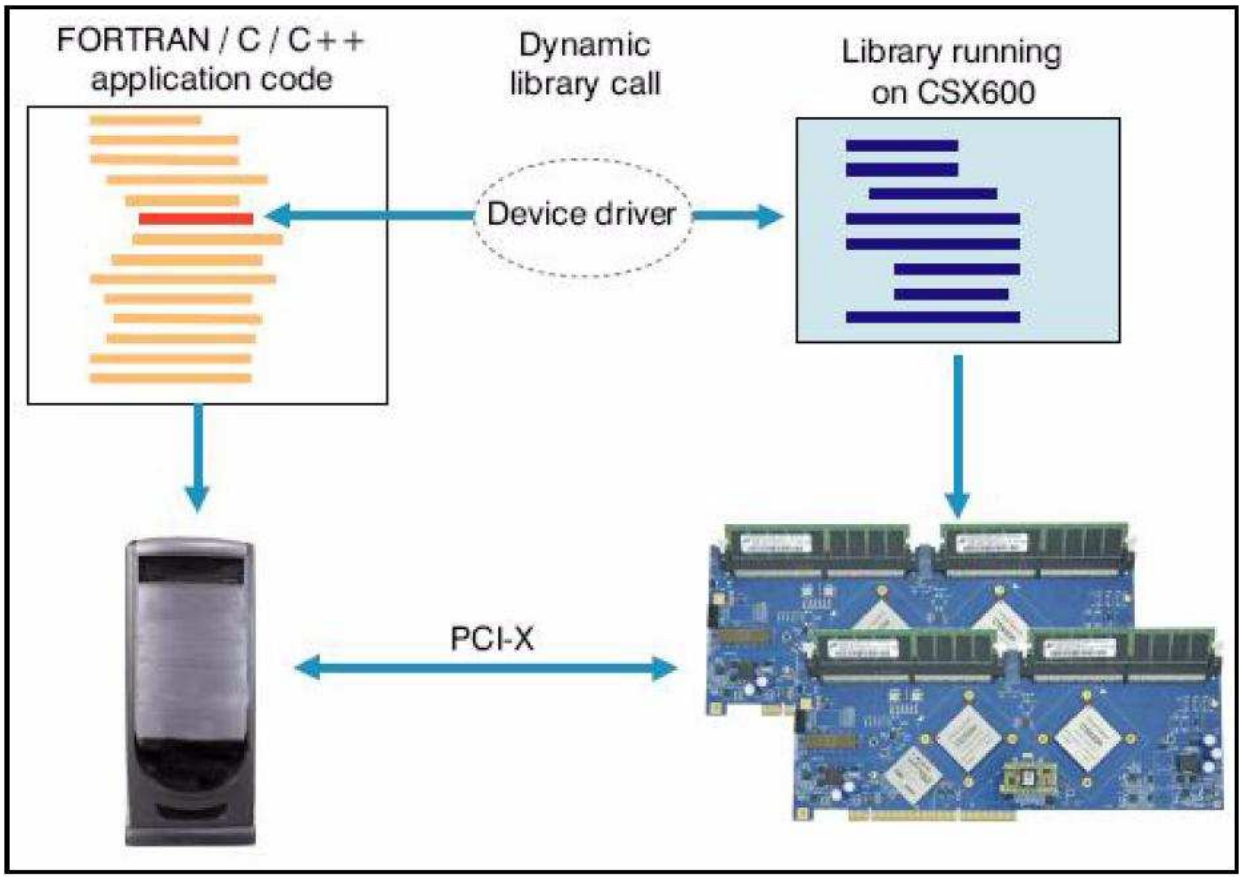

Figure 3. Communications between CSC and host processors [72].

An explanation is given of the C^n language and its features, which is mainly based on ANSI C. There is a description of basic types, pointer types, array types, struct and union types. Due to having mono and poly variables, understanding of pointer types is more difficult. There is the possibility of creating a pointer *mono type* * *poly type*, this means that a poly object points to an object in mono memory, which is shown in figure 4. This is used only for data transfer library functions because any manual use of this type of pointers can involve a serious performance delay.

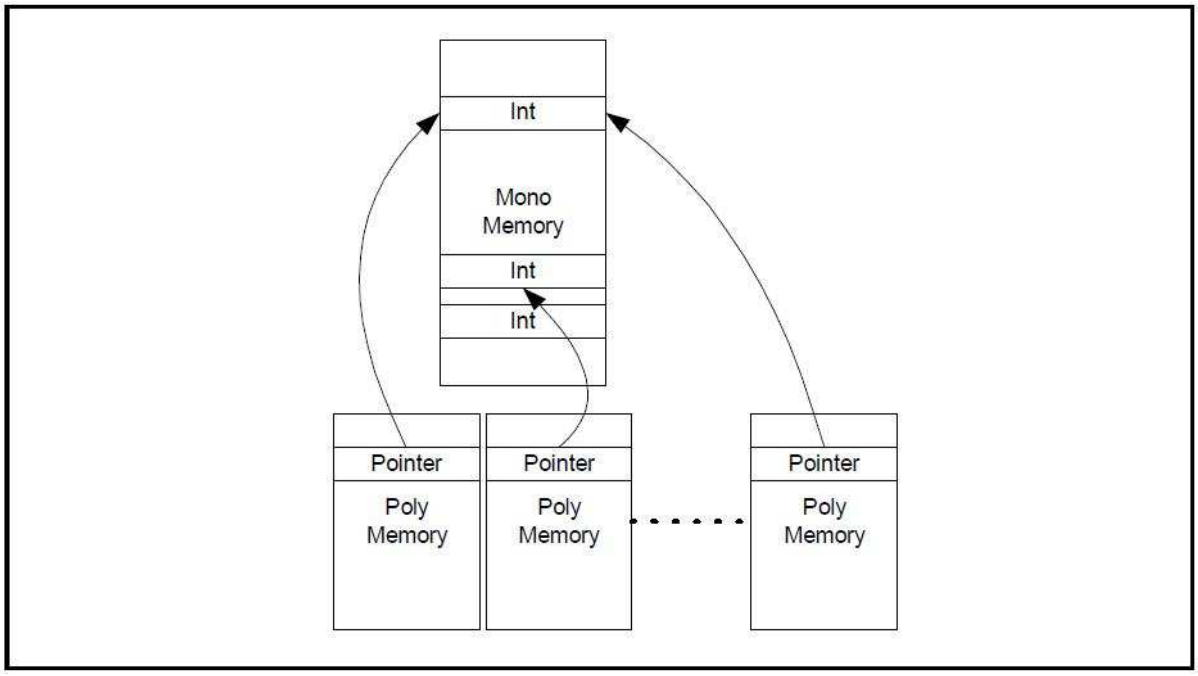

Figure 4. Description of *mono type * poly type* [72].

Conversely, a pointer *poly type * mono type* can be created, this means that a mono variable points to a poly variable. It is possible to implement this due to the fact that a poly variable is stored on each PE at the same address. The figure 5 from [72] shows the connections for this pointer.

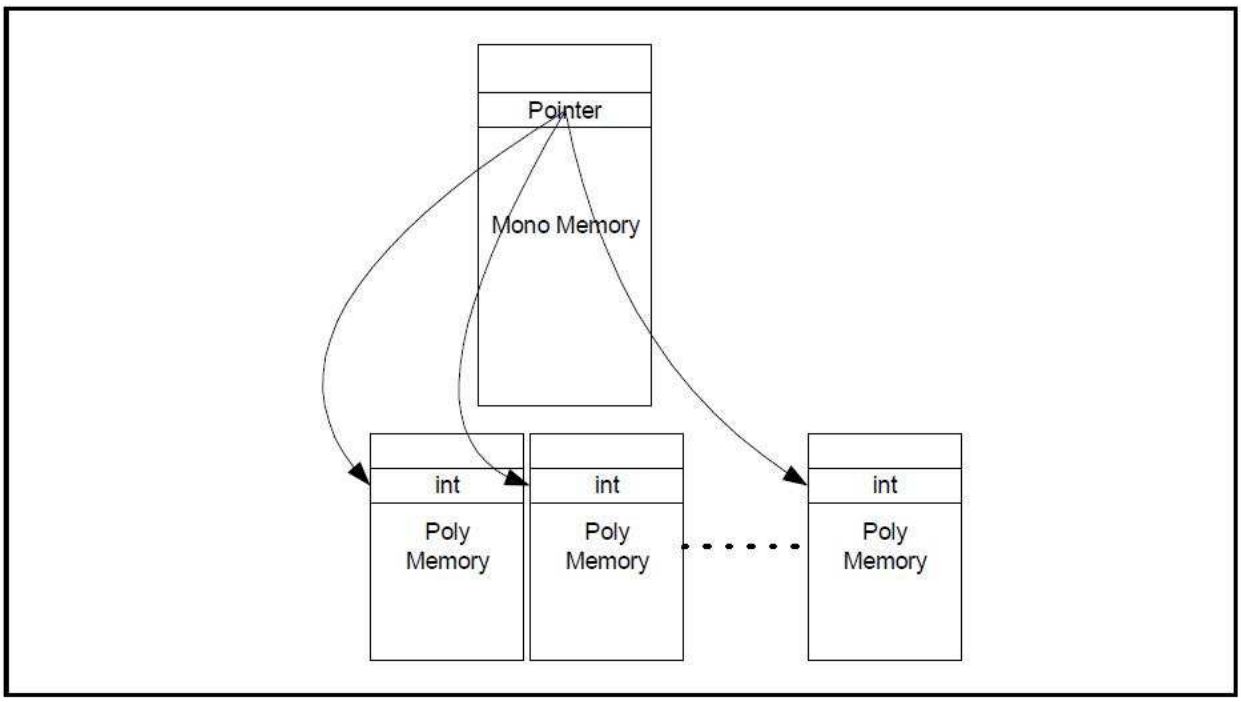

Figure 5. Description of *poly type * mono type* [72].

There are some restrictions in order to avoid an extra complexity, for instance, the use of a goto statement. Owing to mono and poly data stored in different spaces, it is illegal to assign or cast a poly pointer to a mono pointer or vice versa. This was done because the poly memory uses 16 bit pointers, while the mono memory uses pointers with 32 bit and above. It is legal to mix poly and mono variables, [72] presents some rules about legal mixing, it is also concerned about poly conditionals, loops and statements. For instance, implementation of mono and poly variables in *if*-condition may look like:

```
poly short penum;
penum = get_penum();
if (penum < 10){
...
}
else {
...
}
```

There is a mix of mono and poly variables in the pseudo code, which is presented above. There is a poly condition in *if*-statement, and if this condition is met, then the first branch is executed by the enabled PE. Otherwise, the second branch is executed by the disabled PE. However, for any mono variable there will be executed two branches of *if*-condition, due to the fact that the mono execution unit is always enabled for poly conditions.

Input and output operations for the data transfer between poly and mono sections of memory are also introduced. The CSX processor supports different techniques for data movements. There are synchronous functions of data transfers between different memory sections. For the correct transfer of information, PEs specify the source or destination machine address and only elements, which have been enabled and can transfer the same amount of data from or to a location in the poly memory. There are also asynchronous versions of the data transfer functions, which let the programmer execute an input or output copy on a detached thread so calculations can be performed in parallel with data transfer. Asynchronous data transfer functions use semaphores for synchronisation, data coherency and completion of data [72].

There are a few examples of C^n programs: concurrent calculation of a Mandelbrot set and synchronous and asynchronous data distribution in [72]. Finally, developers described some useful programming and debugging hints in order to avoid basic problems and get more performance during the first experience of programming for the CSX processor in [72].

During research of the project on the Clearspeed CSX600 computer, it was found, that the parallel computer is supposed to permanently coordinate a use of shared resources, but not conflicts, which occur between parallel processors. Also, a fault-tolerance is the foundation of some difficulties that occur during programming on this parallel system. It is difficult to recognise that the system does not work entirely. Hanged processors were quite difficult to identify, because when PEs hanged the whole system remained to work clearly but it was producing incorrect data. Yet it was easy to identify a dead halt, which ap-

peared only if all nodes including the processor in the mono execution unit were halted. In order to restore the entire system, it is necessary to reboot the whole system and call the command *csreset*, which reinitializes the board and its processors. These difficulties emerged only at the beginning of the project implementation. Therefore, in order to remove this obstacle, it is strongly recommended to use a simulator during compilations of first versions of the projects.

3.2 General methodology of the parallel development and implementation of the SDFT and IDFT algorithms

In essence, a process of digital signal processing starts when a signal comes from a microphone or a musical instrument, when the signal is converted to a sequence of real numbers, which is called a time domain digital signal, and so a process of transforming the signal starts from a real value sequence. This sequence is used as an input array to the SDFT algorithm, which transforms the time domain signal into a complex frequency domain signal. The next stage is utilization of an application, which can be anything, for example pitch shifting or filtering. The next stage is an IDFT algorithm, this method is complementary to the SDFT. The input of the IDFT is the complex frequency domain signal, which has been transformed during execution of the application. And lastly, the output of the IDFT is a real time domain signal, which goes directly to a speaker. The development of the IDFT was done as extra work in order to complete an analysis tool and give free play to a computer music composer's imagination.

The following image shows the order of data movements from the microphone to the speaker.

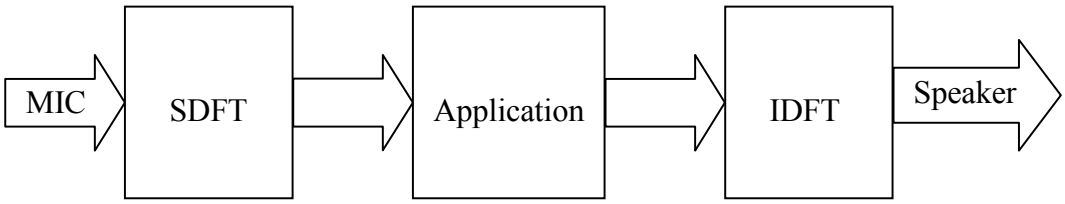

Figure 6. Order of algorithms and data movements.

The input comes directly from the microphone and it is treated as an array of real numbers in double precision. The reason for using 8 bytes of memory for storing each element of input is described in the "Introduction" chapter.

The project runs on a SIMD parallel computer and is programmed in C/C^n languages. This method was optimised in order to carry this out in real time. This investigation benefits the application of parallel processing to sound synthesis, and aspects of sound analysis and modification.

The main aim was to find equilibrium between the time delay and the window size. Due to window size *(N)* being variable it is possible to get a range of times for the purpose to find out the marginal time that it takes to calculate the algorithm. Tests were done with a value of *N* in the range from 64 to 4096 elements. The N is a significant figure, which determines how precise the sound can be so the larger the window size the more accurate will be the resulting output. But with a large window size program it could take a long time for copying data and the marginal window size could take longer time for calculations due to the context switch time delay, which is needed for the processor from a mono execution unit to switch from the current task to the next one. Also during the balancing of the window size and time, it was necessary to find the smallest value of time in the SDFT algorithm.

The hardest part of the algorithm was memory management, which is significant in the transfer of data accurately from mono to poly memories and back. It was difficult to manage each parallel processor to find its own absolute address in memory spaces for data "copy from" or "copy to". Also sometimes it was confusing to do management calculations in bytes so that the physical address in memory cards must be calculated precisely.

The task is subdivided into parts that are calculated individually between each concurrent processor and then all the results are put back together to make the final result. So, this architecture of tight coupling, as well as the high communication rate between shared memory and processors, does not permit use of all the performance from asynchronous chunk data copy, due to specific problems, which are shown below in the analysis sections.

An analysis requires taking into account the granularity of the architecture and semantics of the parallel extension C^n. Thus in the experiments chunks of data need to be of sufficient size to overcome the communication costs.

Algorithms mostly differ from each other due to data transfer management between poly and mono memory. During tests the number of SDFT input elements *(S)* was chosen to get wall clock time within the scope of 1 to 10 minutes. Tests had to run for a sufficiently long time in order to avoid timing noise during calculations, so that results are more precise on average.

DFT calculations were done on the Poly Execution Unit with direct data calling from Mono memory. Each processor sends instructions with specific variables and calls for results directly without copying of input data. This method of calculations is most appropriate due to the fact that this process is done just once. It means that the result of DFT formula is a complex number, which is treated as two real numbers and stored in the buffer consecutively:

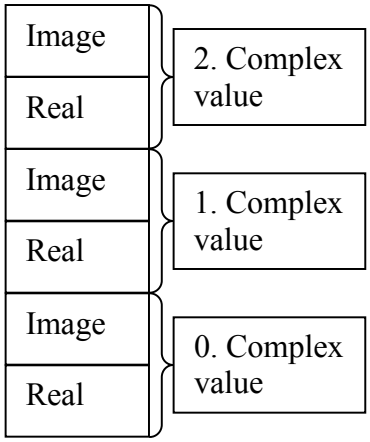

Figure 7. Image of storing complex numbers in mono and poly memories.

As described in detail in the "Literature review" chapter, the original idea of SDFT came from the Goertzel algorithm, where the next value of DFT depends on current value and two values of input:

$$F_{t+1}(x) = (F_t(x) - x_0 + x_N)e^{\frac{2\pi ik}{N}}$$

There are several steps for correct algorithm implementation:
At the first stage of the algorithm the calculations of the initial DFT are achieved using the classic Discrete Fourier Transform equation:

$$F_k(x) = \sum_{j=0}^{N-1} x_i e^{\frac{-2\pi ijk}{N}} \qquad \textit{Equation 3}$$

At the second stage, direct calculations using SDFT formula are used, as described in the "Background" chapter. In the implementation of the algorithm a *"t-loop"* loop was executed *S* times: in each consecutive time, two input values $\{x_0; x_N\}$
were copied and the previous value of SDFT. Because input real values are stored consecutively: $\{x_0; x_1; x_2; \ldots x_{N-2}; x_{N-1;} \ldots\}$, it is necessary to copy the input from two different places in the Mono memory. So it means that data copy functions were calling twice. Time

costs were essential in this loop and also it was expected a marginal amount of time was consumed for calculation of this algorithm.

The third stage is for implementation of an Inverse Discrete Fourier Transform. Data, which is used for calculations, is stored only in one place and it does need to call the function for data transfer just once. Yet the equation, for implementation of the IDFT is:

$$x_t = \frac{1}{N} \sum_{j=0}^{N-1} F_t(j) e^{\frac{2\pi jk}{N}} \qquad \textit{Equation 4}$$

It is conspicuous from equation 4, that it is necessary to transfer N complex double precise elements from the Mono memory to the Poly memory. So owing to the fact that the Clearspeed CSX processor is a fine-grained vector accelerator, the results of IDFT algorithm might be slower than the SDFT implementation.

In the analysis of asynchronous algorithms a complementary buffer was introduced, due to the fact that real time performance was required. The buffer inherently avoids overlapping by a small fraction of a sample rate. This fraction was usually 1 or 2 sample rates.

A poly variable *k* was introduced, whose value varies from *0* to *N-1*. This poly variable was presented in order to correctly manage input and output transfers of all processor nodes. Each parallel node has its own value of *k*, which is calculated with a library function *get_penum()*, the output of this function is a number, which varies from 0 to 95. Due to the Clearspeed CSX600 having one chip with 96 processors, the number of processors is fixed. A loop was written for a *k*-distribution, because the window size is normally greater then the number of processors. Consecutively it is necessary to distribute correctly all elements from the Mono memory. This *k*-distribution loop will always take part in concurrent calculations later.

It was developed, implemented and analysed following variants of the original SDFT algorithm:

1. the successive SDFT algorithm;
2. the synchronous data copy of the SDFT algorithm;
3. the asynchronous data copy of the SDFT algorithm;
4. the asynchronous data chunk copy for the input of the SDFT algorithm;
5. the asynchronous data chunk for the previous SDFT and input elements of the SDFT algorithm;
6. the asynchronous data chunk copy for the SDFT elements of the IDFT algorithm;
7. the asynchronous data calculations in the shared memory with results' copy of the IDFT algorithm.

During the analysis of algorithms, tables of calculation times were created for the SDFT loop calculations, and DFT loop calculations were done only for tests, whose wall clock time fits the requirements described above. Times of DFT loop calculations were calculated with the initializing of input data, some variables and calculation of DFT of first N input elements, which are calculated in the "Initialisation" column.

In the rest of this chapter each algorithm will be described in detail; there will also be presented a pictorial diagram of the algorithm, and a graph of the experiment's results of the time delay for a range of window sizes for one sample. Additionally there are tables from the experiments' results in the "Appendix" section. The results of experiments vary by $\pm 0.01s$ and in the result tables there are presented examples of experiments, not the average time delay, since the round error of $\pm 0.01s$ is negligible and acceptable. Also in the charts there are two series of time delays, one is the algorithm which is described in each section and another is a whole project, which are the SDFT and the IDFT algorithm implementations. The IDFT algorithm was calculated by equation 4 on a mono execution unit but each PE calls for its own results according its own input data, which are poly variables. After this calling the results were stored on the poly memory on each PE, this means

that computations after calculations on each vector at time = *t* where done, results were copied back to mono memory asynchronously.

3.3 Analysis of the successive algorithm

The aim of implementation of a successive algorithm of DFT and IDFT was to compare the results of concurrent implementation with successive implementation and show that a parallel version has better performance so it will triumph over the consecutive programming; and computer music society will be prevailed upon to use High Performance Audio Computing.

For implementation of DFT equation 3 was used and for implementation of IDFT equation 4 was used. The followed chart shows the results of the successive algorithm.

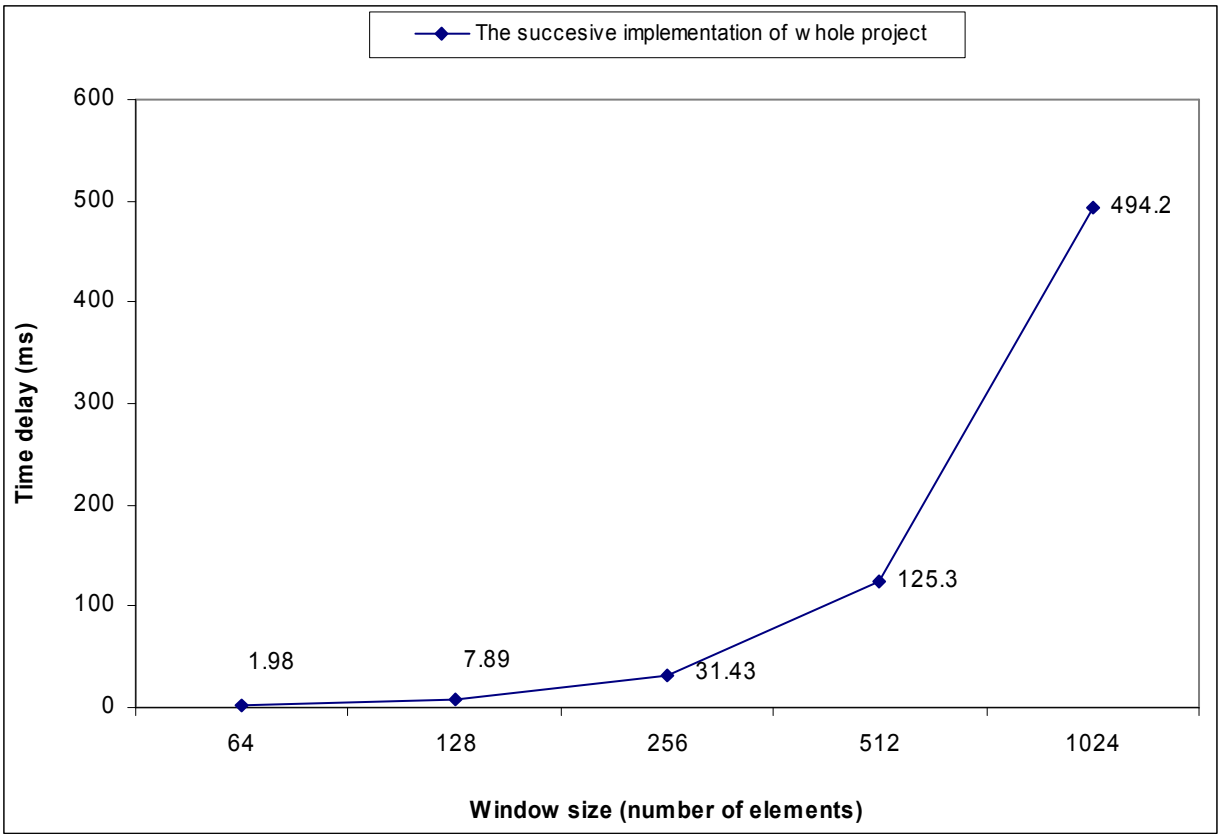

Chart 1. Dependency of the time delay on the window size in the successive algorithms of the DFT and IDFT.

The chart below provides an overview of the time delay depending on the window size that the project runs within 10 ms only with window size equal to 128 elements. The next window sizes are 256 and 512, and time delays are 31 and 125 ms respectively. When the number of elements in the window size reached a 1024, the time delay significantly increased to 494.2 ms.

However, in order to achieve professional quality for the tool it is necessary to have a window size that is more than 128 elements. Also results from DFT and IDFT were not shown separately because the main interest of this project is the time delay out of the SDFT parallel algorithm and parallel implementation of the SDFT and the IDFT algorithm. So at the end of this chapter consecutive and concurrent implementations will be compared.

3.4 Analysis of the synchronous data copy of the SDFT algorithm

Perhaps the simplest model of parallel computing is synchronous data copy, where all nodes operate in a lockstep manner and this fact prevents data from overlapping. It also makes it easy to maintain data integrity. During each communication round all parallel nodes receive the data from the shared memory, perform local calculations, and send messages back to the shared memory. It was reasonable to develop a synchronous algorithm, since the architecture of the machine is homogeneous and specifics of the algorithm are that the amount of data, which was used in input and output operations was the same in each PE.

The synchronous data copy algorithm provides a simple example of SDFT calculation. In order to implement this algorithm standard *memcpy()* library functions were studied.
The next stages of the algorithm describe a data movement from the microphone:

1. Input sends N elements, which is enough to find an initial DFT. Values of DFT are stored in the buffer in Mono memory, so this data is available for every PE. Input data is also stored in Mono memory. All values are ready and available to calculate the SDFT algorithm in synchrony.
2. Firstly, input is copied by *memcpy()* function in the t-loop. In the next stage in the *k*-distribution loop an appropriate value of previous DFT is copied, this value can be founded in the buffer by poly number *k*.
3. Calculations of SDFT are done synchronously in parallel, and in order to copy data back to mono memory poly variable *k* is used.

Figure 7 displays an illustrative example of data management in *time* = *i+1*. It shows that in this time the next input value x_{i+N} is coming to the buffer, which is stored in Mono memory. This input is sent to poly processor elements in synchrony with the previous value of SDFT $F_{i,k}$, where k is a poly variable. Parallel processors perform some operations concurrently and send the result to the mono memory. The result is a SDFT value in

time = i+1: $F_{i+1,k}$. Once the buffer has received all results in *time = i+1*, all values of SDFT are sent to the next stage – IDFT algorithm. This is a stage of calculations showing one element of the input stream.

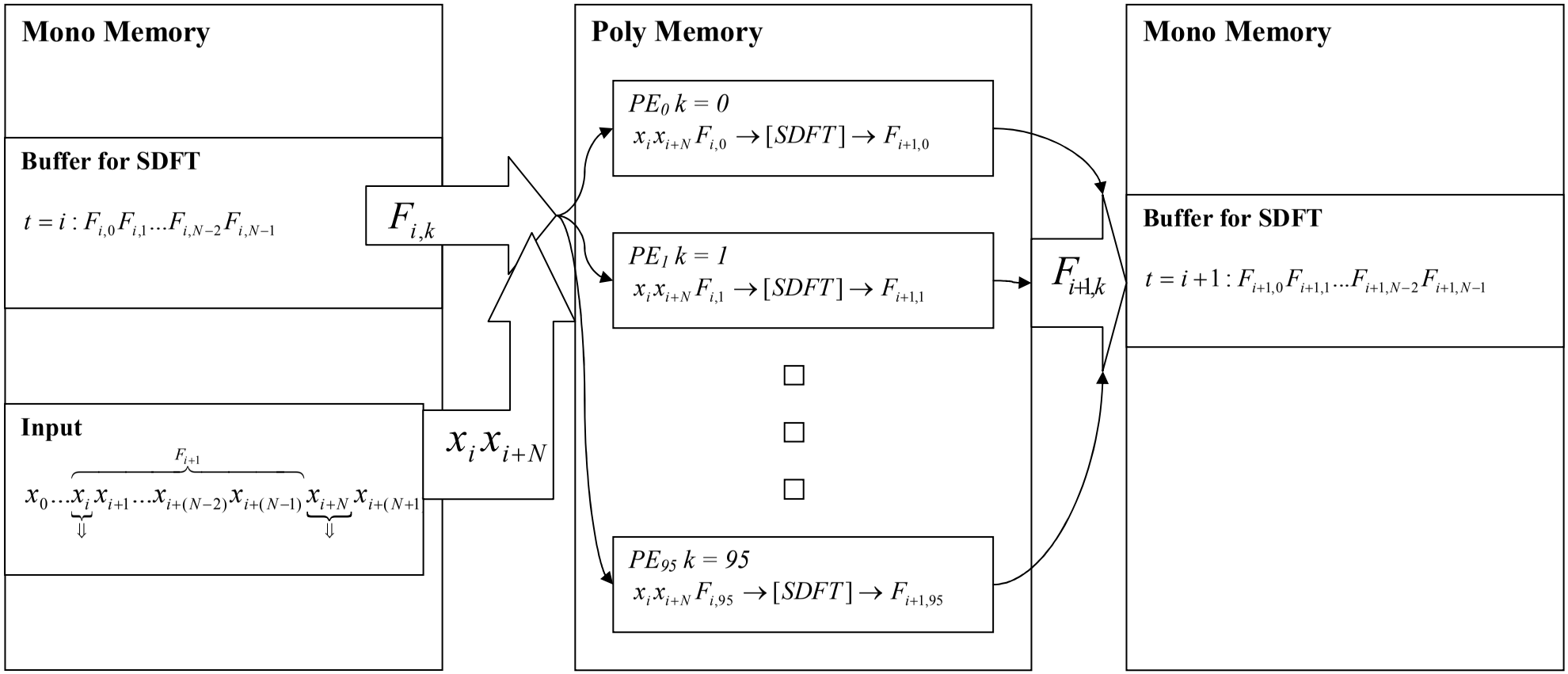

Figure 7. Description of the data copy of the synchronous algorithm of SDFT.

As previously noted, the most challenging part of implementation is data management to make input and output data coherent. In this version all concurrent processors work in synchrony and some management is done by library functions in an exchange of valuable time. In figure 8, there is a view of two-dimensional array from the mono memory, where the SDFT values are stored.

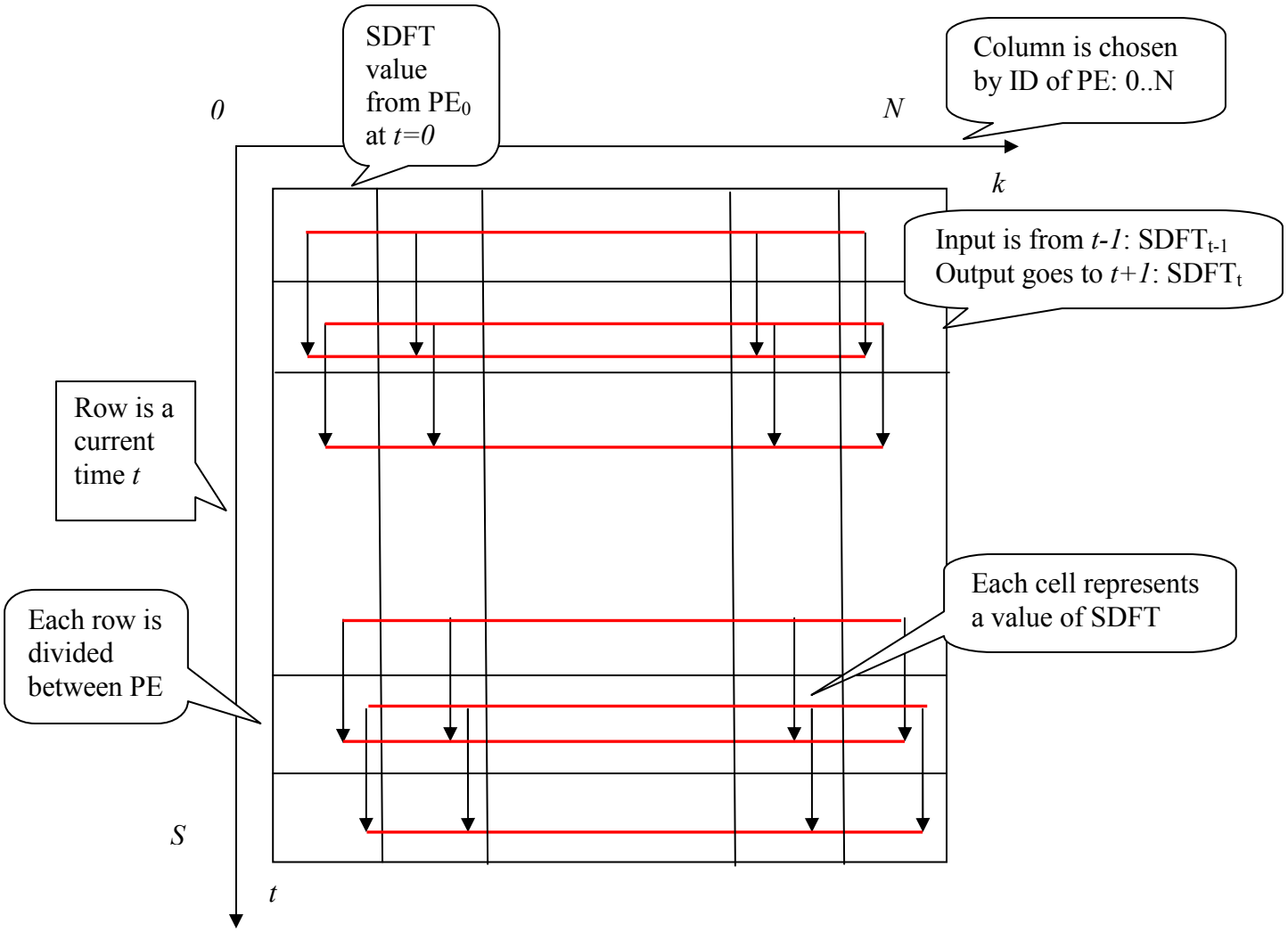

Figure 8. A synchronous order of storing and copying information in the Mono memory.

So each PE, according to its ID copies a piece of data from the specific place in the array. The figure shows how data moves inside the array, red lines show that data is copied in synchrony, so if the processor finishes a data copy at any stage, it must wait for others to finish the same procedure. In the array at the first stage data is copied from the first row and moves

down, where columns are chosen by the ID of each processor element, so at the time = *t*, each processor element copies the SDFT value from the previous row and proceeds with the calculations, the output data is stored at the row = *t* and so on. The project was tested on a finite number of elements in order to optimize performance of the algorithm during implementation and analysis. After analysis of the synchronous data copy of the SDFT algorithm the dependency between window size and the time delay was found, which is shown below in a Chart 2.

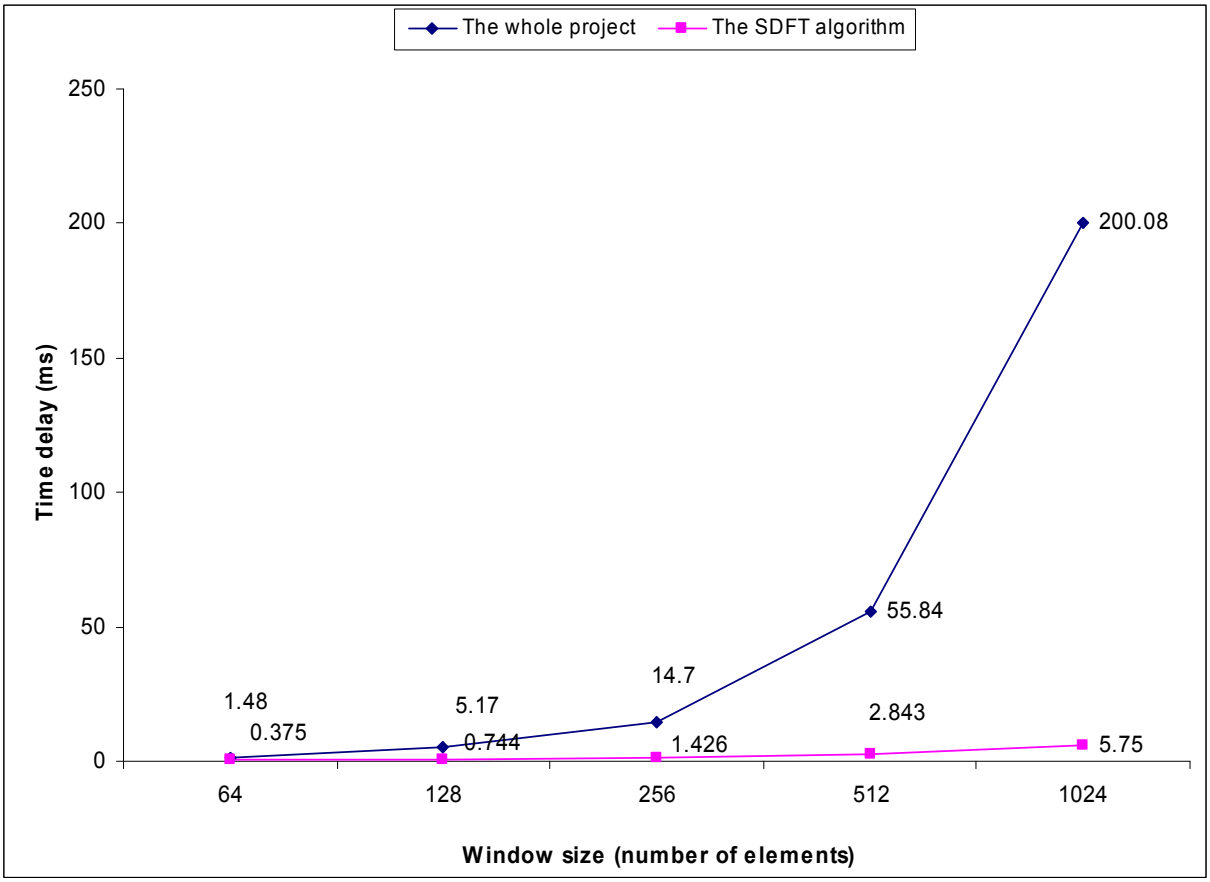

Chart 2. Dependency of time delay on the window size in the synchronous data copy algorithm.

Chart 2 reveals dependencies of time delay on the window size. It can be seen that time, which is needed for calculations, is increased with window size growth. Between 64 and 128 elements the time increases slowly, but with 256 elements the figure rises a little over 10 ms. In 1024 window size the time delay rises very sharply and reaches 200.08 ms. By contrast,

the table from the appendix shows that time needed for SDFT is significantly lower than IDFT time. For instance, with window size equal 128 SDFT calculates 9 times faster than IDFT. To sum up, the synchronous version of the IDFT parallel algorithm gives a very good speed up which is 5.75 ms per sample and N = 1024.

3.5 Analysis of the asynchronous data copy of the SDFT algorithm

After developing and implementing the synchronous version of data copy of the SDFT algorithm, the asynchronous version of data transfers was developed and implemented. The main difference here is using library functions of asynchronous data copy: *async_memcpym2p, async_memcpyp2m,* whose meaning are asynchronous data copy from mono to poly memory and asynchronous data copy from poly to mono memory respectively. These functions are similar to synchronous, but using these library tools allows performing them on a separate thread and after the thread calculations in order to continue running. In order to synchronize computations of threads and operations of reading or writing it is necessary to use semaphores. Functions of asynchronous data copy take an extra parameter of the identity (ID) of a semaphore [72].

Generally two buffers are used as a double buffering technique and the functions of asynchronous data copy are applied. The principle of this method is while one buffer (background buffer) is being filled the data from a second buffer (active buffer) is being calculated.

There are some general steps, which facilitate using this technique:

1. First data are copied into the active buffer.
2. Wait, until the background buffer becomes empty
3. Next data are copied into the background buffer
4. Wait, until the active buffer becomes full and then do calculations on the data
5. Copy data back from poly memory to mono memory

6. Swap the two buffers
7. Go to step 2, until the input data finish

Some algorithms of asynchronous data copy of the SDFT were developed. The main difference between these algorithms was the data management in order to find the balance between the window size and time delay.

Firstly, the asynchronous data copy algorithm was developed and implemented. The amount of data, which was transferred between mono memory and PEs was one element. In figure 9 is shown the asynchronous way of data management in the mono array, where data is stored. Each PE copies to its SRAM the piece of information independently of others, executes a calculation on it and then the PE copies it back to mono memory. Red lines show that transfers do asynchrony and if any parallel processor finishes data copy to its SRAM, this processor can do calculations and does not need to wait for others.

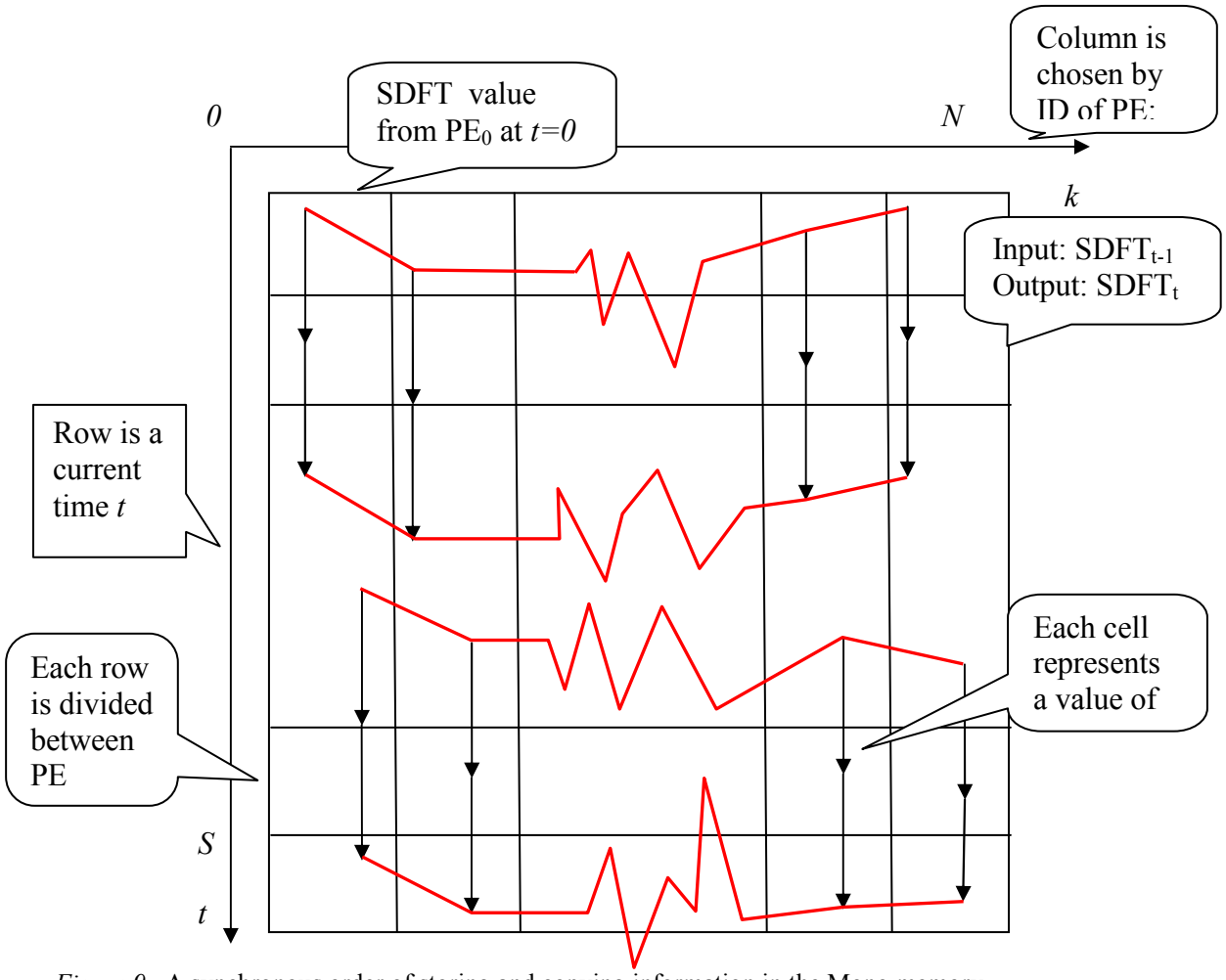

Figure 9. A synchronous order of storing and copying information in the Mono memory.

After implementation of the SDFT algorithm many experiments and analyses of it were made. Chart 3 shows the result data achieved using this new algorithm.

Chart 3. Dependency of time delay on the window size in the results of asynchronous data copy implementation of the SDFT algorithm.

Chart 3 shows the asynchronous implementation of the SDFT algorithm. With the window size equal to 512 and above time for calculating an element is sharply increased. So with the window size equal to 64 and 128 elements the time for calculating an element is 0.34 and 1.30 ms respectively. The window size with 256 elements also shows a marginal time delay, which is 3.83 ms. However, the time results grow considerably with N equal to 512: it takes more then 15 ms to calculate SDFT for an element and around 70 ms for the calculation in total. In the chart above, the last number of elements in the time domain digital signal is

1024. The asynchronous SDFT algorithm needed 55 ms in order to compute a sample and approximately 250 ms is the total time of calculations in the project. In conclusion, the results of this asynchronous version of the SDFT algorithm show that there are some aspects, which can be filled in order to make the project faster.

3.6 Analysis of the asynchronous chunk data copy of the SDFT algorithm

Two asynchronous algorithms of the SDFT with chunk data copy were developed. The first to be developed was the SDFT algorithm with a chunk data copy of the previous SDFT. Input elements were transferred one by one. Here, a variable was introduced, which defines the size of chunk, which is indicated as *T,* which means number of copied samples per calculation round, where $N / T = k, k \in N$. This algorithm was implemented, tested and analyzed. Figure 10 shows the chunk of data is moved by each parallel processor. The chunk is a piece of vector from array, this vector is chosen by time = *t*. And each PE proceeds with this chunk of data independently from the others, so it means PE communicates with the mono execution unit only to copy data to poly SRAM or copy results back to mono memory asynchronously. In the each poly SRAM the parallel processor executes calculations with the chunk of data and then copies it back to mono memory.

Dividing the input frequency by chunks means that the number of involved PE is N / T so if T is large then not every PE takes a part in calculations it means the computer is not using the whole power, which is available. For instance, if $N = 1024$ and $T = 128$, than $N / T = 8$, so only 8 processors will be involved in this kind of calculation.

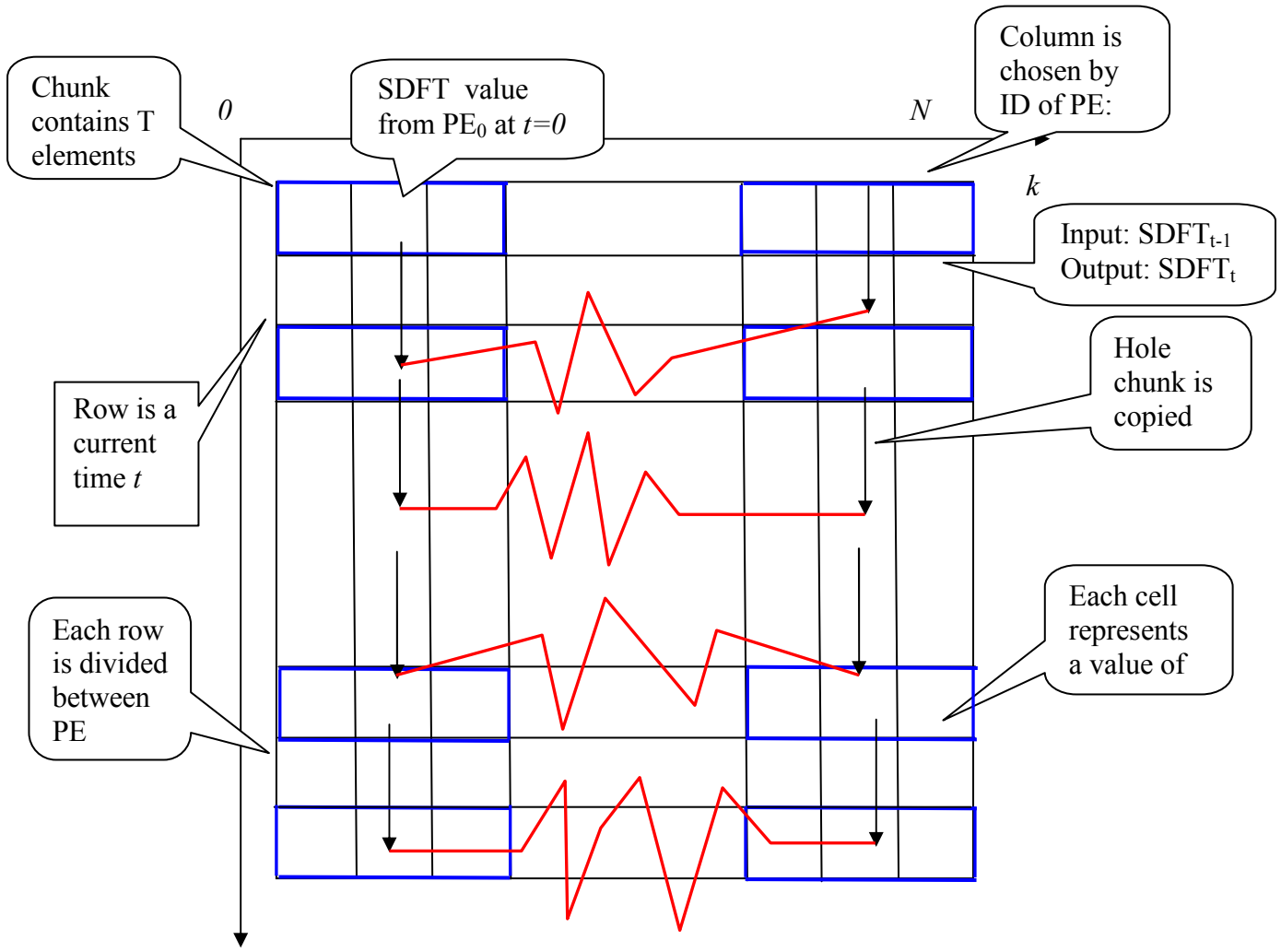

Figure 10. An asynchronous order of storing and copying chunks of information in the Mono memory.

Chart 4 shows the dependency of the time delay on the window. Tests were conducted where it varied with window size and for each window size it varied with chunk size.

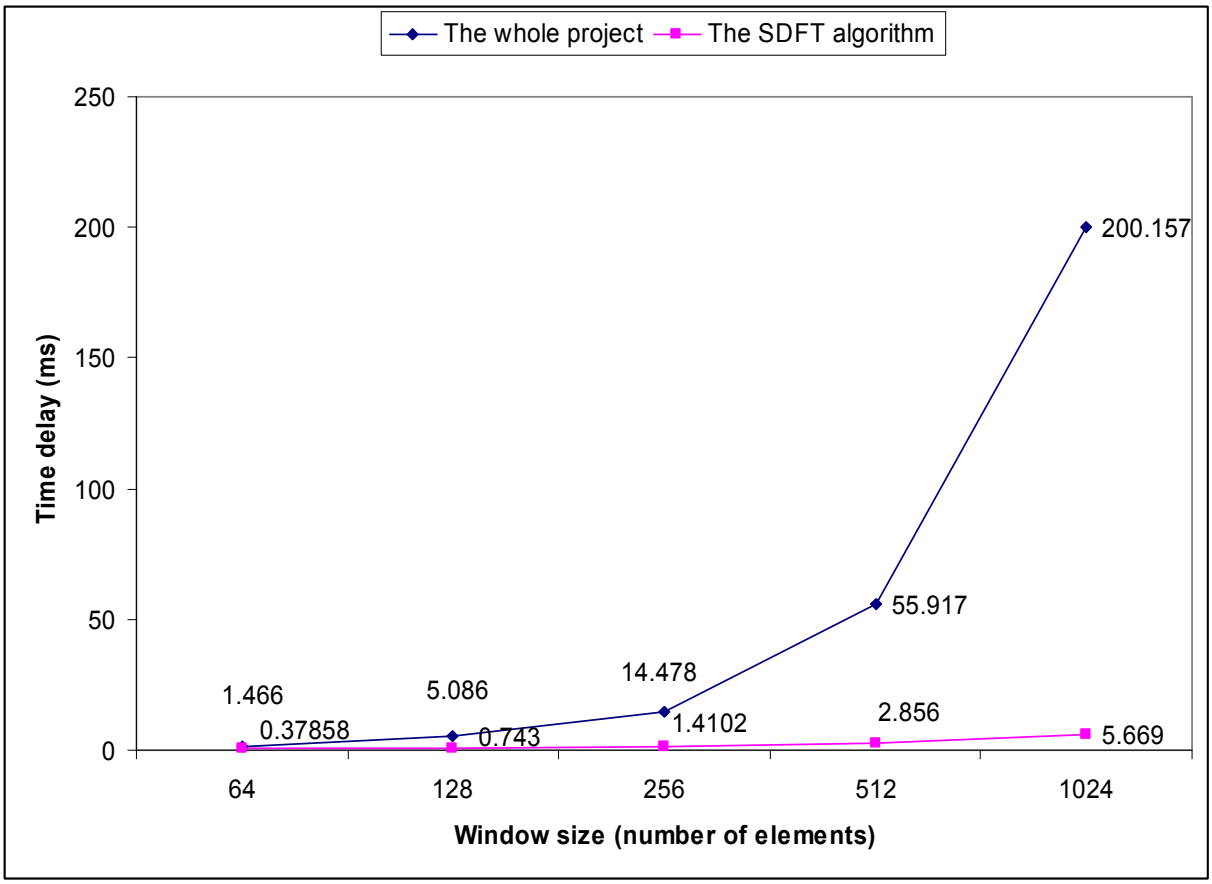

Chart 4. Dependency of the time delay on the window size of the asynchronous algorithm with chunk data copy of previous SDFT for calculation of the current SDFT.

Chart 4 shows that the performance of the SDFT algorithm rose dramatically. Due to the fact that here, a chunk data copy was used, and the algorithm shows the best performance with one element in the chunk. Also the algorithm was optimized in the light of results from the previous version. The SDFT algorithm with an input number of time domain elements equal to 64 and 128 calculates a sample in less than 1 ms. The algorithm with 256 and 512 samples per window performed the task within 3 ms. With the number of input elements equal to the tenth power of two the SDFT algorithm computes in 5.66 ms. To sum up, this algorithm provided good results, 5.66 ms were needed per sample with the window's size equal to 1024 samples, so this means that it was chosen as the right way for optimization and software acceleration.

3.7 Analysis of the asynchronous data chunk for SDFT and input of the SDFT algorithm

The last version of the SDFT algorithm was developed with a full chunk data copy implementation. It means that the input frequency sent in chunks of samples into the SDFT implementation. Also in each loop it was proceeded T elements, so if the current time = t, then the next time will be $t+T$ and the next T samples will come. In comparison with previous algorithms, where the frequency domain signal was stored in the buffer within the one row, in this algorithm output data from the SDFT algorithm were stored within T rows. The data manager in each PE must find the exact row and column for storing a current value of the output. If in the algorithms above PEs are transferred data within one row, then here, a small two dimension array with $T*T$ elements is copied. Also there is a limitation on the amount of copied data due to each SRAM having only 6 Kbytes of memory with a stack for input equal to 3 Kbytes and another 3Kbytes being taken for calculations and using libraries. So finding the necessary chunk of data in mono memory and finding the exact place where it is needed to copy it back became a big issue here, but it was solved, tested and analyzed.

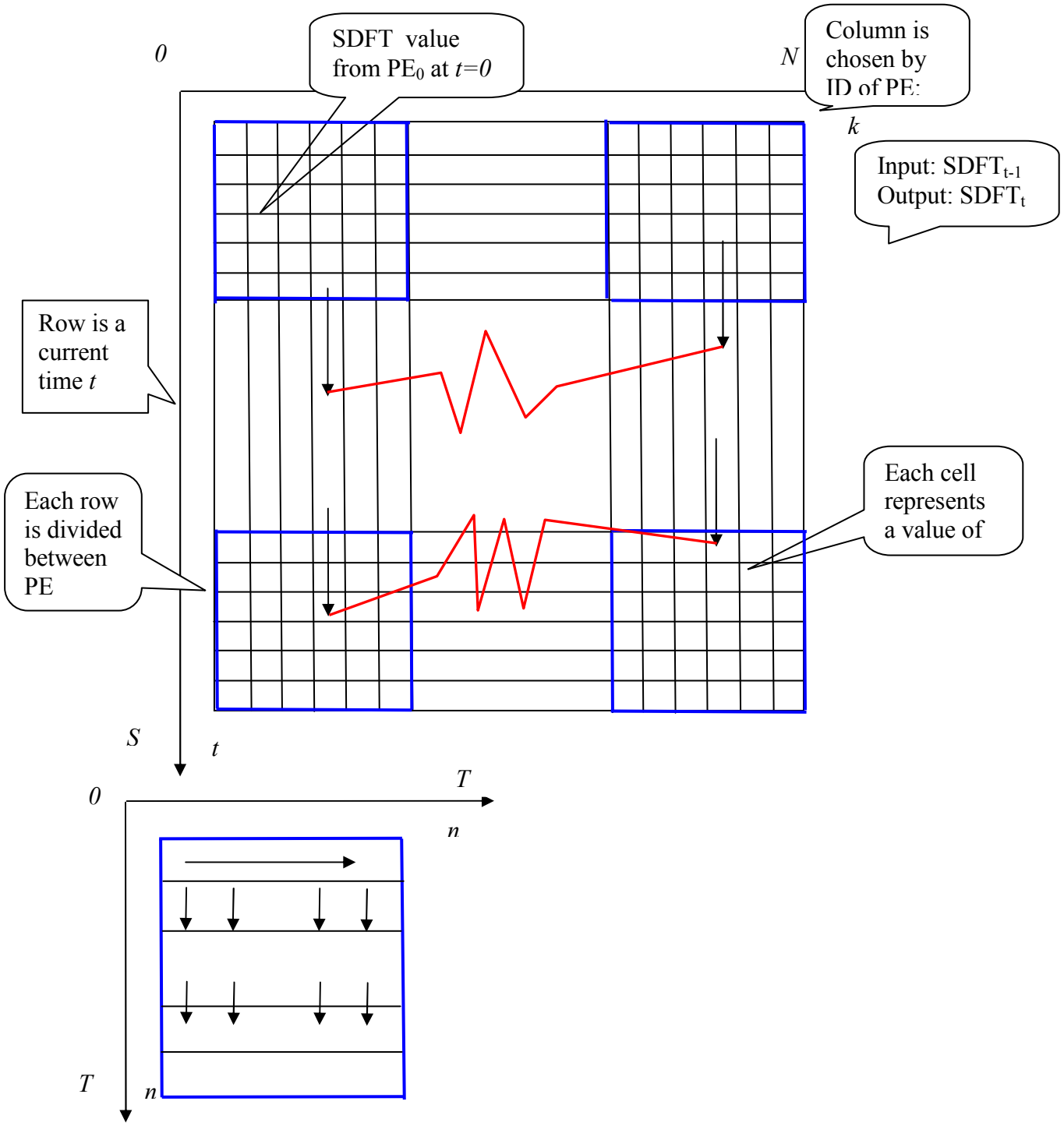

Figure 11. An asynchronous order of storing and copying chunks of information in the Mono memory and within the chunk in the Poly memory.

Figure 11 represents the communications between layers inside the chunk. These communications are done within a one communication round between poly SRAM and mono memory.

Chart 5 presents the results of the last SDFT algorithm, which was developed. It shows the best result, which was achieved on ClearSpeed CSX 600 processor. The performance is better than with chunk data copy of only a vector. After tests, it also showed that the best performance was achieved with copy of 4 complex elements between mono memory and processors, tests were done on window size, with variation from 64 elements to 4096 and with variation of elements in the chunk from 1 element to 64.

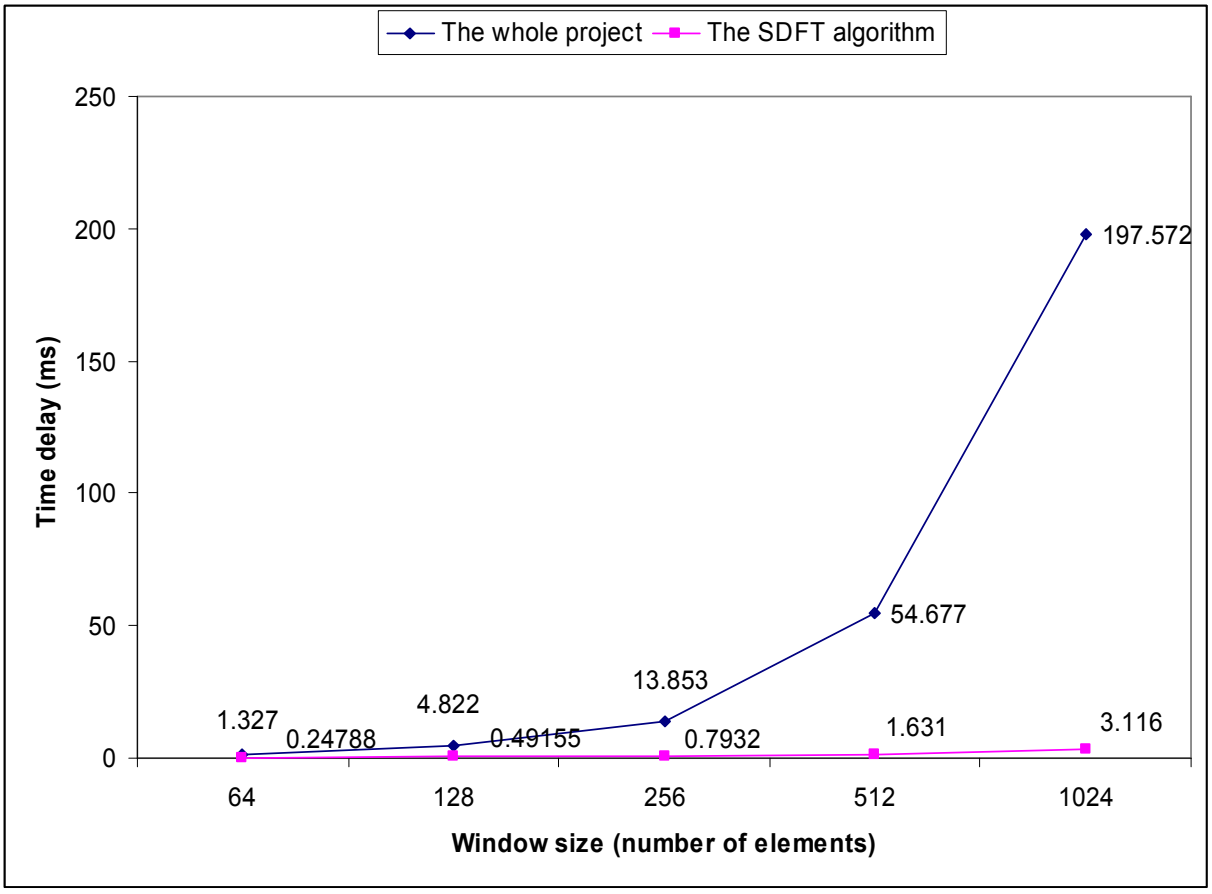

Chart 5. Dependency of time delay on window size.

Chart 5 shows the changes of time delay according to the different window sizes. Performance of the SDFT algorithm is less than 1 ms with window size 64, 128 and 256. The time delay increased slightly from 1.63 ms with 512 elements in the window to just above 3 ms in the next test. The tradition here is that by doubling the window size the time delay is increased by two times as well. On the whole, the development of the SDFT algorithm reveals the best performance out of the vector computer. Also tests were done with a window size of 2048 and 4096 elements in the window. These data did not feature in the analysis since for excellent quality of sound 1024 elements are enough, but in the "Appendix" chapter the results from all the tests are presented.

3.8 Analysis of the asynchronous chunk data copy of the SDFT values in the IDFT algorithm

After analyzing the SDFT algorithm of chunk data copy of previous SDFT and input. The whole project is included the SDFT and the IDFT algorithm. Tests show that the IDFT algorithm is in 5 - 65 times slower than the SDFT. The main reason for this is that in the IDFT it is necessary to transfer a much larger amount of data from the mono memory to the poly memory and back. An initial algorithm of the IDFT was done in the way of not copying input elements, but doing calculations on the mono memory and these calculations are done by PEs, so each parallel processor straight calls for the results from mono memory, so the results are stored on poly memory. And after all calculations are done PEs find the particular address in the mono memory and copy the results in synchrony. So an algorithm was developed of asynchronous data copy of the frequency domain signal from mono memory to poly using double buffering. The resulting implementation issued quite slowly.

The analysis of the asynchronous double buffered IDFT algorithm shows that it is necessary to improve it. An improvement was made to this algorithm by creating chunks of complex elements, sending these chunks to each PE, executing calculations on them, and sending back the results. After the development of such an algorithm and the implementation of it, the algorithm was tested on different window sizes and different numbers of elements in the chunk.

Window size varied with 64 to 4096 elements; chunks varied with 1 to 128 complex elements. It shows much better time delay than the previous implementation of the IDFT algorithm. And also the best time was achieved with 128 complex elements per chunk.

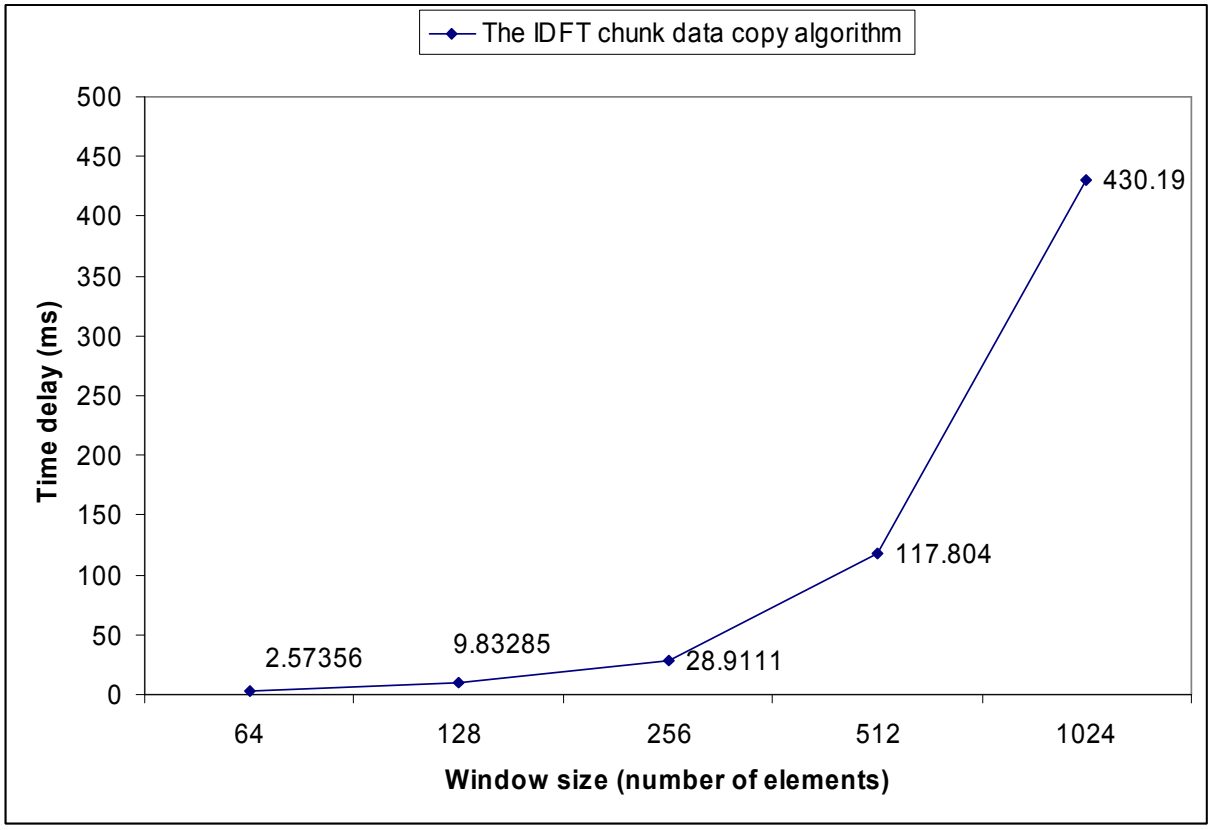

Chart 6. Dependency of time delay on the window size of the asynchronous algorithm with chunk data copy of SDFT for calculation of the IDFT.

Chart 6 outlines the results of tests conducted to determine the best performance of the IDFT algorithm with asynchronous chunk data copy. The time delay gradually increases from 64 to 256 elements in the window. Furthermore, the tendency changes on 512 elements in the window and time steadily grows to 117 ms per one element. The time delay deteriorated on testing with window size equal to 1024 and rose dramatically to 430 ms. Lastly, we showed two different IDFT algorithms, and with these performance was much lower than with the SDFT algorithms; the IDFT algorithm is a performance bottleneck in this project.

All things considered, we showed the detailed description of developed and implemented algorithms. The analysis of parallel algorithms was also done with a description of each algorithm; development and implementation was characterized satisfactorily. We optimized the SDFT algorithm in order to speed it up. We also developed the IDFT algorithms in order to get the whole tool for a conversion from a time domain signal into a frequency domain representation and vice versa. In the next chapter we will describe the results and conclusions of the conducted research.

Chapter 4

Results and Evaluation

After empirical analysis of the project experiments it is necessary to compare results and find the fastest algorithm. Firstly, we undertook an evaluation of

- the parallel algorithm of asynchronous data chunk copy of the previous SDFT and input elements for the SDFT;
- the parallel algorithm of asynchronous data chunk copy of the input signal for the SDFT;
- the parallel algorithm of asynchronous data copy for the SDFT;
- the parallel algorithm of synchronous data copy for the SDFT.

The chart below shows the evaluation of time delays of one sample utilizing a variety of developed parallel algorithms.

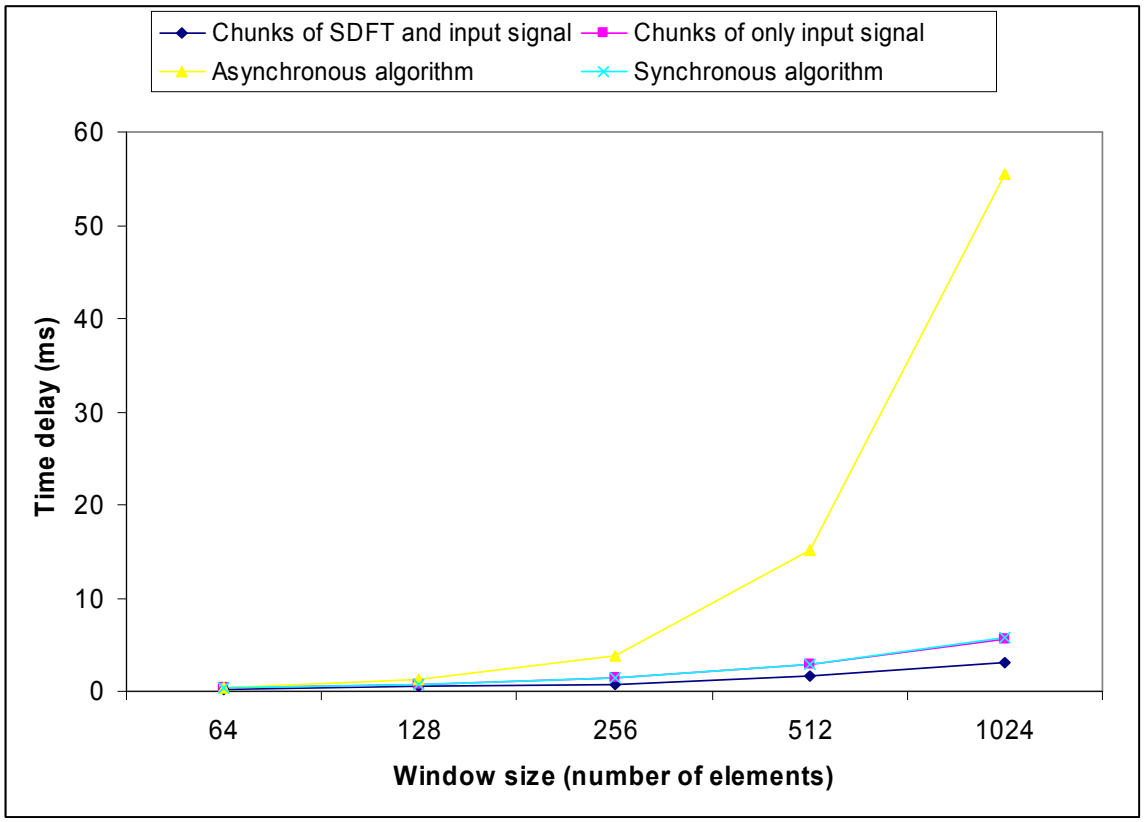

Chart 7. Evaluation of SDFT algorithms.

Chart 7 provides an overview of the performance per sample of parallel algorithms. The asynchronous version is the slowest algorithm, and with a window size of 1024 samples it executes a sample within 55 ms, while for others it takes less then 5 ms. It is not surprising, that the synchronous algorithm and the asynchronous algorithm (with chunk data copy of the input signal) give approximately the same results of 5.66 ms and 5.75 ms respectively. These algorithms transfer the same number of samples and we did not use any technique to improve performance in the asynchronous version. The algorithm of asynchronous chunk data copy of input and previous SDFT arrays showed the best performance; it needed 3.16 ms with a value of N = 1024 for the calculation of one sample. Consequently, the parallel algorithm that makes the most of the parallel architecture is the asynchronous chunk data copy of the input and the previous SDFT frequencies with a chunk of 4 samples.

Secondly, it is necessary to compare the results of IDFT algorithms. We implemented two algorithms of the IDFT:

1. the asynchronous chunk data copy of the IDFT algorithm;
2. the algorithm with IDFT calculations which were done on poly and mono execution units asynchrony.

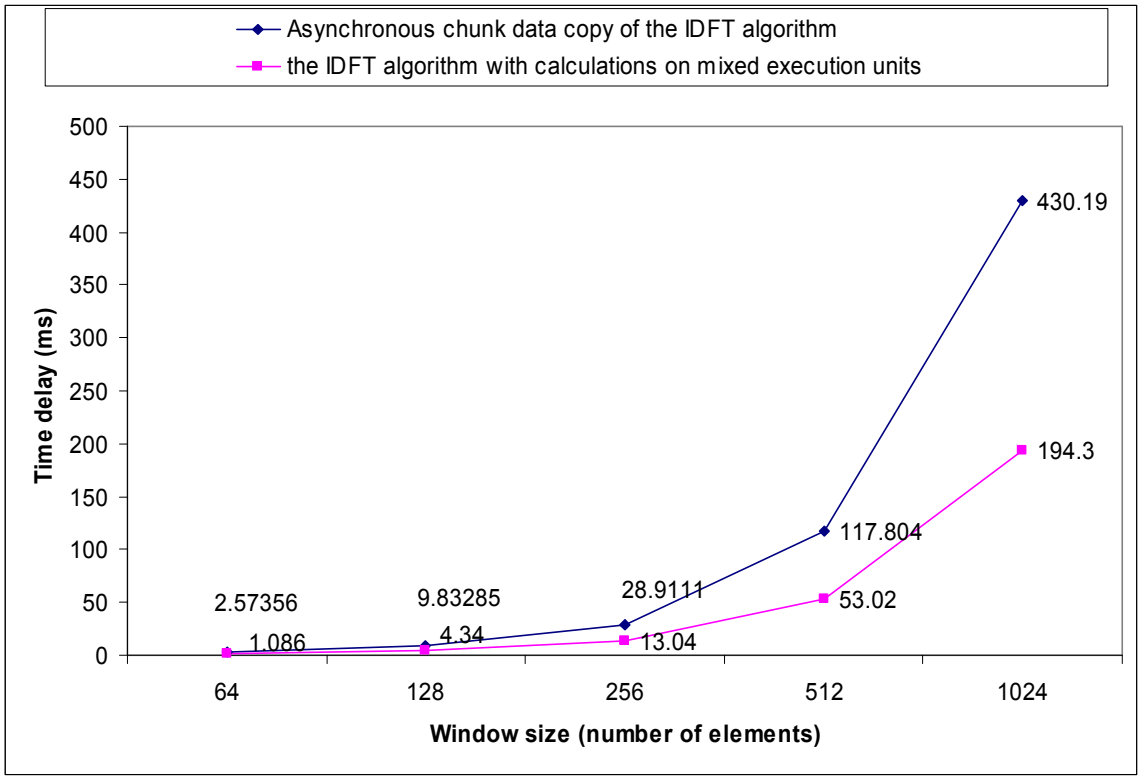

Chart 8. Evaluation of IDFT algorithms.

Chart 8 shows that the best performance achieved utilising the IDFT algorithm when calculations are mixed on the poly and mono execution units. The algorithm of the asynchronous chunk data copy is slower by a factor of 2 on each window size. Therefore, in the IDFT algorithm, where calculations are mixed on poly and mono execution units, performance is significantly higher. However, the execution time of the IDFT and the SDFT fastest algorithms are very different. Furthermore, in the IDFT algorithm it was necessary to manipulate a large amount of data between execution units, so features of communication tools was a consideration. Also, in order to achieve a real-time performance from the IDFT algorithm with mixed calculations, it is possible to use it only with a window size of 256 double precise complex samples.

To sum up, the fastest IDFT algorithm is the algorithm of asynchronous chunk data copy of input and previous IDFT arrays with a performance of 3.16 ms per sample and window size N = 1024. The best SDFT algorithm is the IDFT algorithm with calculations done on mono and poly execution units with a performance of 13.04 ms per sample and window size equal to 256 double precision elements.

The analysis of the parallel implementation of the SDFT showed a great performance. The achievement of the real time implementation was successful. The optimization of the concurrent algorithm reached a real-time performance and maximisation of the performance on the CSX600 processors was very successful.

Chapter 5

Conclusions and Future Work

Traditionally, the development of scientific parallel computing has been tightly coupled with general computing. It has been shown in the "Literature Review" chapter, that historically the supercomputer was created as an extension of the computer. This extension was created mostly on the level of hardware but it seems much more complicated to provide good quality software in order to fully exploit the potential of the supercomputer. The architecture of the parallel computer still varies with specific features of applications that are executed on the machine. Scientific concurrent applications are only used in mathematics, physics, engineering and computer sciences. Yet, parallel computing has become dominant through science. A tradition established that beyond the world of computing engineers there is a push towards HPC and from the other side, from the side of natural sciences, there is a pull, stemming from their needs for parallel processing. It is also clear that sequential algorithms and programming have become behind a lesser consideration in computer science.

In the beginning, when the decision was made to implement the SDFT algorithm on the Clearspeed CSX600 processors, it was not known whether parallelisation would be beneficial for this particular task. The conclusion depends on many complex factors such as synchronisation needs, features of the architecture and the nature of the algorithm. Too many nodes introduce communication overheads owing to scheduling, redundant PE management as well as context-switching.

This research shows how time crucial parallel computations are in digital signal processing and music technology in particular. After analysis and evaluation of the algorithms it is clear that in order to create a High Performance Audio Computer it is necessary to conduct many observations of the algorithms, which are in use in the audio processing area. It is necessary to strive for greater productivity in order to discover how efficient parallel processing can be.

We found a real time algorithm for the SDFT and we developed the IDFT as an extra. The fastest SDFT algorithm transfers chunks of 4 double précised samples per one communication round. The fastest algorithm of the IDFT was implemented as a composition of calculations on mono and poly execution units, whereas in the IDFT algorithm all computations were done within the poly execution unit. The data management was the most difficult part during the development and implementation of the algorithms, yet data integrity and validation were successfully achieved.

After analysis of the developed concurrent algorithms it became clear that the SDFT algorithm runs faster than the IDFT algorithm. The main difference in computations of these algorithms is that the IDFT requires a large amount of data, consequently, communications between processor elements and mono memory became the most expensive part in terms of time. Also, each PE has only 6 Kbytes of SRAM, which was not enough for high performance implementation of the IDFT algorithm with the large window size. The nature of the project may require a use of a heterogeneous multi-core computer.

The next task is to create algorithms for the heterogeneous parallel computer in order to achieve better performance on the IDFT algorithm. The real-time performance on the SDFT

algorithm was achieved and run within 3 ms with the window size set to 1024. It was suggested that in order to achieve approximate performance for the IDFT algorithm it would be necessary to employ high bandwidth 20 Kbytes SRAM on each PE, as well as to integrate a faster PCI host interface. This is due to the fact that communications costs between mono and shared memories are quite expensive.

The HiPAC has a high reputation within computer music society. The research, which has been carried out, is only a first step, with yielding results of HiPAC and the research elucidated on a challenging area of audio processing, which limits have not permitted to exploit parallel processing in extensively free. We believe that this research will reflect an area of audio performance computing, as well as digital signal processing in general, due to the developed algorithms widely in use.

To the best of the knowledge of the researchers of the current project, there is not yet any real-time parallel tool for professional audio processing, which we have made here. The issue is that researchers have not seriously looked at the problem of parallel computing in audio processing. Yet to achieve parallel virtue, Parallel Research Centres should recruit more parallel computer scientists. Lastly, bridging the gap between computer music composers and the hardware must be done urgently since otherwise the field of computer music risks becoming completely left behind.

Bibliography

1. Kant, I., *Critique of pure reason. 1781.* Modern Classical Philosophers, Cambridge, MA: Houghton Mifflin, 1908: p. 370-456.
2. Einstein, A., *The foundation of the general theory of relativity.* Annalen Phys, 1916. **49**(769-822): p. 31.
3. ffitch, J., R. Dobson, and R. Bradford, *HIGH-PERFORMANCE AUDIO COMPUTING—A POSITION PAPER.* International Computer Music Conference, Belfast 2008.
4. Abe, T., T. Kobayashi, and S. Imai, *The IF spectrogram: a new spectral representation.* Proc. ASVA, 1997. **97**: p. 423–430.
5. Lazzarini, V., J. Timoney, and T. Lysaght. *Streaming Frequency-Domain DAFX in Csound 5*. 2006.
6. Lazzarini, V., *Developing Spectral Processing Applications.* Proc. Of the 2nd Linux Audio Developer's Conference (Karlsruhe: Zentrum fuer Kunst- und Medientechnologie, 2004)
7. Mersenne, M., *Harmonie Universelle (1636).* Fac-similé F. Lesure éd., Paris, CNRS.
8. Koyré, A., *Metaphysics and measurement*. 1992: Gordon and Breach Science Publishers : New York.
9. Strutt, J., *The theory of sound*. 1877. London: MACMILLAN AND CO. Vol 1 and Vol 2.
10. Sabine, W., *Collected papers on acoustics*. 1922: Cambridge : Harvard University Press.
11. Johnson, H. Earle. 1979. *Symphony Hall, Boston*. New York: Da Capo Press.
12. Fildes, J. *'Oldest' computer music unveiled*. 2008; [cited 10/2010; http://news.bbc.co.uk/1/hi/technology/7458479.stm].
13. *Christopher Strachey, biography*. Wikipedia [cited 10/2010; http://en.wikipedia.org/wiki/Christopher_Strachey].

14. *Max Mathews, biography*. Wikipedia [cited 10/2010; http://en.wikipedia.org/wiki/Max_Mathews].
15. Hochheiser, S. *John Pierce: Biography*. 2008; [cited 10/2010; http://www.ieeeghn.org/wiki/index.php/John_Pierce].
16. *John R. Pierce, biography*. Wikipedia [cited 10/2010; http://en.wikipedia.org/wiki/John_R._Pierce].
17. *Risset, Jean-Claude, biography*. The living composers project [cited 10/2010; http://www.composers21.com/compdocs/rissetjc.htm].
18. *Center for Computer Research in Music and Acoustics*, Stanford University. [cited 10/2010; https://ccrma.stanford.edu/].
19. *SYNTH SECRETS*. Sound on Sound 2000; [cited 10/2010; http://www.soundonsound.com/sos/apr00/articles/synthsecrets.htm].
20. *Yamaha DX-7*. Vintage synth explorer; [cited 10/2010; http://www.vintagesynth.com/yamaha/dx7.php:]
21. *IRCAM, the Institute for Research and Coordination Acoustic/Music*. [cited 10/2010; http://www.ircam.fr/ircam.html?&L=1].
22. Walker, J. *The analytical Engine, the first computer*. Fourmilab Switzerland [cited 10/2010; http://www.fourmilab.ch/babbage/].
23. Babbage, C. *The Analytical Engine*. 1888 [cited 10/2010; http://www.fourmilab.ch/babbage/hpb.html].
24. Fuegi, J. and J. Francis, *Lovelace & Babbage and the Creation of the 1843'Notes'*. IEEE Annals of the History of Computing, 2003: p. 16-26.
25. (2008) *Introductory Programming Manual The ClearSpeed Software Development Kit*. in *Clear Speed Corp.* 2007 [cited 10/2009 http://support.clearspeed.com/documentation/hardware/csx600/].
26. Summers, B., *ClearSpeed CSX620 Overview*, in *Clear Speed Corp.* 2007. [cited 10/2009 http://support.clearspeed.com/documentation/hardware/csx600/].
27. Bradford, R. and R. Dobson, *John ffitch. Sliding is Smoother than Jumping.* SuviSoft Oy Ltd, Tampere, Finland, editor, in *International Computer Music Conference.* 2005: p. 287–290.

28. ffitch, J., et al., *Sliding DFT for fun and musical profit.* in *International Linux Audio Conference. 2008.*
29. Menabrea, L. and A. Lovelace, *Sketch of the Analytical Engine Invented by Charles Babbage, Esq*. 1843: Taylor and Francis.
30. *Parallel computing, history*. Wikipedia [cited 10/2010; http://en.wikipedia.org/wiki/Parallel_computing#cite_ref-PH753_42-0].
31. Cruz, F.d. *The IBM 704*. [cited 10/2010; http://www.columbia.edu/acis/history/704.html].
32. *Background: Bell Labs Text-to-Speech Synthesis: Then and Now Bell Labs and "Talking Machines"*. [cited 10/2010; http://www.bell-labs.com/news/1997/march/5/2.html].
33. Gill, S., *Parallel programming.* The Computer Journal, 1958. **1**(1): p. 2.
34. Russo, E., *Methods for Sinusoidal Analysis and Resynthesis of Musical Signals*, in *Universita degli Studi di Milano*. 2009: Milano. p. 111.
35. Jacobsen, E. and R. Lyons, *The sliding DFT.* Signal Processing Magazine, IEEE, 2005. **20**(2): p. 74-80.
36. Oppenheim, A. and R. Schafer, *Discrete-time signal processing.* Prentice Hall Signal Processing, 2009: p. 1120.
37. Goertzel, G., *An algorithm for the evaluation of finite trigonometric series.* American mathematical monthly, 1958. **65**(1): p. 34-35.
38. Moore, G.E. *Gordon E. Moore, Chairman Emeritus of the board* Intel Corp. [cited 10/2010; http://www.intel.com/pressroom/kits/bios/moore.htm].
39. Flynn, M., *Very high-speed computing systems.* Proceedings of the IEEE, 2005. **54**(12): p. 1901-1909.
40. Asanovic, K., et al., *The landscape of parallel computing research: A view from Berkeley.* EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, 2006: p. 2006-183.
41. Catanzaro, B., et al. *SEJITS: Getting productivity and performance with selective embedded JIT specialization*. EECS Department, University of California, Berkeley 2009: Citeseer.
42. Garcia, G., *Optimal filter partition for efficient convolution with short input/output delay.* Preprints- audio engineering society, 2001.

43. Battenberg, E., A. Freed, and D. Wessel, *ADVANCES IN THE PARALLELIZATION OF MUSIC AND AUDIO APPLICATIONS.* in *International Computer Music Conference.* 2010.
44. Y. Orlarey, D.F., S. Letz *Fauts, signal processing language*. 2009; [cited 09/2010; http://faust.grame.fr/].
45. Y. Orlarey, S.L., D. Fober. *MULTICORE TECHNOLOGIES IN JACK AND FAUST*. in *International Computer Music Conference*. 2008.
46. Bradford, R. and R. Dobson. *The Sliding Phase Vocoder*. in *International Computer Music Conference*. 2007.
47. Hill, M. and M. Marty, *Amdahl's law in the multicore era.* Computer, 2008. **41**(7): p. 33-38.
48. ffitch, J., Richard Dobson, and R. Bradford, *The Imperative for High-Performance Audio Computing* in *Linux Audio Conference.* 2009.
49. Orlarey, D., et al., *Reinventing Audio and Music Computation for Many-Core Processors.* in *International Computer Music Conference*. 2008.
50. Van Roy, P. *The Challenges and Opportunities of Multiple Processors: Why Multi-Core Processors are Easy and Internet is Hard*. in *International Computer Music Conference*. 2008.
51. Puckette, M. *Thoughts on parallel computing for music*. in *International Computer Music Conference*. 2008.
52. Wang, G., *A COMMENT ON MANY-CORE COMPUTING AND REAL-TIME AUDIO SOFTWARE SYSTEMS (2008).* 2008.

Appendix A

Tests of variations of the SDFT algorithm

Number of DFT input elements	Number of SDFT input elements	Algorithm wall clock time (s)	Time of calculation an element (ms)	Initialisation time (s)	Time of SDFT loop Calculation (s)	Time of IDFT loop Calculation (s)
64	1000	0m1.972				
64	10000	0m19.852				
64	50000	1m39.249	1.98	0m0.003	0m1.701	1m37.560
128	1000	0m7.917				
128	10000	1m18.952	7.89	0m0.006	0m0.69	1m18.213
256	1000	0m31.453				
256	10000	5m14.386	31.43	0m0.017	0m1.385	5m12.559
512	1000	2m5.389	125.3	0m0.063	0m0.334	2m5.010
1024	1000	8m14.263	494.2	0m0.963	0m0.785	8m13.772
2048	100	3m18.463	1984.6	0m0.961	0m1.070	3m18.212
4096	100	13m18.461	7984.6	0m3.888	0m4.110	13m17.942

Table 1. Results of successive DFT and IDFT algorithms.

N	S	Algorithm wall clock time (s)	Time of calculation an element (ms)	Initialization time (s)	Time for SDFT calculation (s)	Time of SDFT per sample (ms)	Time of IDFT loop Calculation(s)
64	1000	0m1.525					
64	50000	1m14.402	1.48	0m0.027	0m18.789	0.375	0m55.837
128	1000	0m5.216					
128	10000	0m51.733					
128	20000	1m43.437	5.17	0m0.029	0m14.880	0.744	1m28.741
256	1000	0m14.825					
256	10000	2m27.063	14.70	0m0.034	0m14.263	1.426	2m12.722
512	1000	0m55.950					
512	10000	9m18.459	55.84	0m0.055	0m28.439	2.843	8m49.921
1024	1000	3m20.088	200.08	0m0.123	0m5.759	5.75	6m59.693
1024	100000	332m56.532					
1024	1000000	3329m22.400					
2048	1000	13m8.606	788.60	0m0.403	0m11.648	11.648	12m57.294
4096	500	25m32.399	3064.7	0m1.477	0m12.696	25.392	25m21.094

Table 2. Results of synchronous data copy of SDFT and IDFT algorithms.

Number of DFT input element	Number of SDFT input elements	Algorithm wall clock time (s)	Time of calculation an element (ms)	Initialisation time (s)	Time of SDFT loop Calculation (s)	Time for SDFT calc. per element (ms)	Time of IDFT loop calculation	Time of IDFT calculations per element (ms)
64	1000	0m1.464						
64	50000	1m11.335	1.426	0m0.027	0m17.038	0.340	0m54.331	1.086
128	10000	0m56.490						
128	20000	1m52.928	5.646	0m0.030	0m26.185	1.309	1m26.811	4.34
256	1000	0m16.943						
256	10000	2m48.885	16.88	0m0.034	0m38.329	3.832	2m10.455	13.04
512	1000	1m8.256	68.256	0m0.055	0m15.220	15.22	0m53.023	53.02
512	10000	11m21.51						
1024	1000	4m9.855	249.8	0m0.123	0m55.459	55.45	3m14.316	194.3
2048	100	1m40.930	1009.3	0m0.402	0m22.611	226.1	1m18.765	787.6
4096	100	6m33.780	3937.8	0m1.476	1m28.174	881.7	5m7.786s	3077

Table 3. Results of asynchronous data copy of the SDFT algorithm.

Number of DFT input elements	Number of SDFT input elements	Number of elements in a chunk	Algorithm wall clock time (s)	Time of calculation an element (ms)	Time of calculation an element in SDFT	Time of SDFT loop Calculation (s)
64	1000	32	0m2.643			
64	1000	64	0m3.831			
64	50000	16	1m41.079	2.021		0m48.393
64	50000	32	2m10.749	2.614		1m19.822
64	50000	64	3m10.155	3.803		2m22.612
64	50000	8	1m26.258	1.725		0m32.677
64	50000	4	1m18.842	1.576		0m24.797
64	50000	2	1m15.149	1.502		0m20.880
64	50000	1	1m13.313	1.466	0.37858	0m18.929
128	20000	1	1m41.729	5.086	0.743	0m14.860
128	20000	2	1m43.204	5.16		0m16.421
128	20000	4	1m46.158	5.306		0m19.553
128	20000	8	1m52.092	5.604		0m25.856
128	20000	16	2m3.982	6.199		0m38.446
128	20000	32	2m27.738	7.386		1m3.608
128	20000	64	3m15.269	9.763		1m53.847
256	10000	1	2m24.781	14.478	1.4102	0m14.102
256	10000	2	2m25.910	14.591		0m15.219
256	10000	4	2m28.159	14.815		0m17.470
256	10000	8	2m32.685	15.215		0m22.000
256	10000	16	2m41.740	16.174		0m31.054
256	10000	32	2m59.847	17.984		0m49.164
256	10000	64	3m36.024	21.602		1m25.345
512	10000	32	10m28.346			
512	10000	64	11m40.707			
512	1000	1	0m55.917	55.917	2.856	0m2.856
512	1000	2	0m56.143	56.143		0m3.079
512	1000	4	0m56.59	56.593		0m3.530
512	1000	8	0m57.499	57.499		0m4.437
512	1000	16	0m59.309	59.309		0m6.248
512	1000	32	1m2.931	62.931		0m6.248
512	1000	64	1m10.167	60.167		0m17.107
1024	1000	1	3m20.157	200.157	5.669	0m5.669
1024	1000	2	3m20.569	200.569		0m6.081
1024	1000	4	3m21.396	201.369		0m6.908
1024	1000	8	3m23.059	203.059		0m8.569
1024	1000	16	3m26.381	206.381		0m11.890
1024	1000	32	3m33.022	213.022		0m18.532
1024	1000	64	3m46.288	226.288		0m31.798
2048	100	1	1m20.006	800.06	15.10	0m1.510
2048	100	2	1m20.094	800.094		0m1.593
2048	100	4	1m20.267	800.267		0m1.758
2048	100	8	1m20.614	800.614		0m2.090
2048	100	16	1m21.306	801.306		0m2.754
2048	100	32	1m22.690	802.690		0m4.083
2048	100	64	1m25.453	805.453		0m6.736
4096	100	1	5m10.315	3103.15	36.687	0m3.687
4096	100	2	5m10.474	3104.74		0m3.847

4096	100	4	5m10.792	3107.92		0m4.171
4096	100	8	5m11.431	3114.31		0m4.820
4096	100	16	5m12.711	3127.11		0m6.118
4096	100	32	5m15.268	3152.68		0m8.715
4096	100	64	5m20.378	3203.78		0m13.901

Table 4. Results of asynchronous chunk data copy of input in the SDFT algorithm.

Number of DFT input elements	Number of SDFT input elements	Number of elements in a chunk	Algorithm wall clock time (s)	Time of calculation an element (ms)	Time of SDFT loop Calculation (s)	Time of SDFT calculations per element (s)
64	50000	1	1m13.333	1.466	0m18.978	
64	50000	2	1m6.764	1.335	0m12.614	
64	50000	4	1m6.399	1.327	0m12.394	0.24788
64	50000	8	1m11.817	1.436	0m18.250	
64	50000	16	1m27.131	1.742	0m31.715	
64	50000	32	1m57.968	1.959	1m1.620	
64	50000	64	3m0.360	3.607	2m2.159	
128	20000	1	1m41.714	5.085	0m14.873	
128	20000	2	1m36.591	4.829	0m9.93	
128	20000	4	1m36.448	4.822	0m9.831	0.49155
128	20000	8	1m40.748	5.037	0m14.542	
128	20000	16	1m52.630	5.631	0m25.344	
128	20000	32	2m16.889	6.844	0m49.261	
128	20000	64	3m6.076	9.303	1m37.783	
256	10000	1	2m24.786	14.478	0m14.103	
256	10000	2	2m19.329	13.932	0m8.855	
256	10000	4	2m18.539	13.853	0m7.932	0.7932
256	10000	8	2m21.502	14.150	0m10.893	
256	10000	16	2m30.623	15.062	0m20.017	
256	10000	32	2m49.285	16.928	0m38.679	
256	10000	64	3m27.317	20.731	1m16.708	
512	1000	1	0m55.914	55.914	0m2.855	
512	1000	2	0m54.788	54.788	0m1.812	
512	1000	4	0m54.677	54.677	0m1.631	1.631
512	1000	8	0m55.270	55.270	0m2.223	
512	1000	16	0m57.127	57.127	0m4.081	
512	1000	32	1m1.001	61.001	0m7.954	
512	1000	64	1m8.728	68.728	0m15.680	
1024	10000	16	32m48.545	196.854		
1024	1000	1	3m20.151	200.151	0m5.668	
1024	1000	2	3m17.743	197.743	0m3.551	
1024	1000	4	3m17.572	197.572	0m3.116	3.116
1024	1000	8	3m18.610	198.610	0m4.152	
1024	1000	16	3m21.989	201.989	0m7.532	
1024	1000	32	3m29.078	209.078	0m14.620	
1024	1000	64	3m43.237	223.237	0m28.778	
2048	100	1	1m20.003	800.03	0m1.510	
2048	100	2	1m19.465	794.65	0m1.089	
2048	100	4	1m19.492	794.92	0m1.002	10.002
2048	100	8	1m19.745	797.45	0m1.242	
2048	100	16	1m20.455	800.45	0m2.050	
2048	100	32	1m22.353	802.35	0m4.028	
2048	100	64	1m25.753	805.75	0m7.568	
4096	100	1	5m10.304	3103.04	0m3.691	
4096	100	2	5m9.152	3091.52	0m2.843	
4096	100	4	5m9.246	3092.46	0m2.663	26.630
4096	100	8	5m9.696	3096.96	0m3.126	
4096	100	16	5m11.207	3112.07	0m4.703	
4096	100	32	5m14.967	3149.67	0m8.568	
4096	100	64	5m21.700	3217.00	0m15.486	

4096	50	1	2m37.408	3148.16	0m2.587	
4096	50	2	2m36.830	3136.60	0m2.163	
4096	50	4	2m36.902	3138.04	0m2.096	
4096	50	8	2m37.167	3143.34	0m2.36	
4096	50	16	2m38.083	3161.66	0m3.324	
4096	50	32	2m39.739	3194.78	0m5.026	
4096	50	64	2m43.106	3262.12	0m8.485	

Table 5. Results of asynchronous chunk data copy of input and previous values of the SDFT in the SDFT algorithm.

Number of DFT input elements	Number of SDFT input elements	Number of elements in IDFT chunk	Algorithm wall clock time	Time of SDFT calculations per element
64	50000	1	2m23.585s	2.8717ms
64	50000	2	2m18.303s	2.76606ms
64	50000	4	2m11.025s	2.6205ms
64	50000	8	2m13.363s	2.66726ms
64	50000	16	2m12.458s	2.64916ms
64	50000	32	2m8.158s	2.56316ms
64	50000	64	2m8.678s	2.57356ms
128	20000	1	3m41.220s	11.061ms
128	20000	2	3m36.190s	10.8095ms
128	20000	4	3m32.551s	10.62755ms
128	20000	8	3m21.620s	10.081ms
128	20000	16	3m29.398s	10.4699ms
128	20000	32	3m28.629s	10.43145ms
128	20000	64	3m19.424	9.9712ms
128	20000	128	3m16.657s	9.83285ms
256	10000	1	5m12.949s	31.2949ms
256	10000	2	5m7.702s	30.7702ms
256	10000	4	5m3.338s	30.3338ms
256	10000	8	5m0.943s	30.0943ms
256	10000	16	4m50.805s	29.0805ms
256	10000	32	4m58.692s	29.8692ms
256	10000	64	4m58.153s	29.8153ms
256	10000	128	4m49.111s	28.9111ms
512	1000	1	1m58.931s	118.931ms
512	1000	2	2m1.148s	121.148ms
512	1000	4	1m59.833s	119.833ms
512	1000	8	1m59.081s	119.081ms
512	1000	16	1m58.616s	118.616ms
512	1000	32	1m54.857s	114.857ms
512	1000	64	1m58.027s	118.027ms
512	1000	128	1m57.804s	117.804ms
1024	1000	1	7m11.081s	431.081ms
1024	500	2	3m33.831s	427.662ms
1024	500	4	3m38.475s	436.950ms
1024	500	8	3m37.260s	434.520ms
1024	500	16	3m36.477s	432.954ms
1024	500	32	3m35.898s	431.796ms
1024	500	64	3m29.282s	418.564ms
1024	500	128	3m35.095s	430.190ms
2048	100	1	2m58.565s	1785.81ms
2048	100	2	2m57.319s	1773.19ms
2048	100	4	2m55.837s	1758.37ms
2048	100	8	2m49.732s	1697.32ms
2048	100	16	2m49.230s	1692.30ms
2048	100	32	2m48.864s	1688.64ms
2048	100	64	2m48.581s	1685.81ms
2048	100	128	2m53.231s	1732.31ms
4096	50	1	5m52.095s	7041.90ms
4096	50	2	5m49.929s	6998.58ms

4096	50	4	5m47.159s	6943.18ms
4096	50	8	5m45.475s	6909.50ms
4096	50	16	5m34.236s	6684.72ms
4096	50	32	5m33.539s	6670.78ms
4096	50	64	5m32.978s	6659.56ms
4096	50	128	5m32.565s	6651.30ms

Table 6. Results of asynchronous chunk data copy of SDFT values in the IDFT algorithm.