PARALLELIZING THE ALSA MODULAR AUDIO SYNTHESIZER

Ede Cameron

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

September 2015

CONCORDIA UNIVERSITY

Department of Computer Science and Software Engineering

This is to certify that the thesis prepared

By: **Ede Cameron**

Entitled: **Parallelizing The ALSA Modular Audio Synthesizer**

and submitted in partial fulfillment of the requirement for the degree of

Master of Computer Science

complies with the regulations of the University and meets with the accepted standards with respect to originality and quality.

Signed by the final examining committee:

____________________Dr. W. Shang__________________Chair

____________________Dr. S. P. Mudur________________Examiner

____________________Dr. H. Harutyunyan______________Examiner

_____________________Dr. D. Goswami________________Supervisor

Approved by ____
Chair of Department or Graduate Program Director

____
Dean of Faculty

Date ____

ABSTRACT

Parallelizing the ALSA Modular Synthesizer

Ede Cameron

Digital audio synthesizers are frameworks for generating digital audio, and are the backbone for creating synthesized digital music. Some of the existing audio synthesizer engines are quite popular due to the following reasons: economy, flexibility and convenience for the user in synthesizing music/audio, compatibility with commodity hardware and software platforms. One such audio synthesis engine is the ALSA (Advanced Linux Sound Architecture) Modular Synthesizer, which is an emulation of an analogue modular synthesizer. Until recently, audio programming software has been inherently sequential. There have been some attempts to parallelize a few of these engines with mixed results. The goal of parallelization is not only to obtain speedup but also to increase throughput so that more complex synthesizers can be built to enhance quality and/or complexity of the sound generated. By design, audio synthesizers have soft real-time requirements. This can mean that many of the techniques that are normally used to parallelize a program can in certain situations be too expensive to offer any real performance gain. As a consequence, a naïve parallelization technique for an audio synthesizer can in fact be too expensive due to added overheads, and hence is of no benefit. This paper discusses our methodologies and experiences on parallelizing the ALSA Modular Synthesizer on a multicore environment, and elaborates the experimental results highlighting the advantages of parallelization.

ACKNOWLEDGEMENTS

First and foremost, I wish to express my profound gratitude to my supervisor Dr. Dhrubajyoti Goswami for his continuous commitment, encouragement, and support. I feel privileged for having the opportunity to work with them and share passion for research. I would like also to extend my appreciation to my committee members and to all professors who provided me with their valuable knowledge during my studies at Concordia. I would like to pass my special thanks and gratitude to all my family members and friends:

- My Parents without whose support I would be at a loss in many ways. Words and thanks cannot express my gratitude.
- My daughter Morgane for the understanding and unyielding support, whose "teenagerish" distractions allow me the time to relax and enjoy her company.
- My friends close and far for just been friends.

Thanks to every one who in some way or another played some part in my endeavors.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

KEYWORDS AND ABBREVIATIONS

ALSA - Advanced Linux Sound Architecture - An application layer that allows programmers access to audio devices, it also includes a MIDI sequencer. MIDI has been integrated into most professional sound cards

MIDI - Musical Instrument Digital Interface - a protocol that allows communication between digital instruments. It can be used to sequence multiple synthesizers at the same time and as an interface between physical musical instruments, like a keyboard, and a software synthesizer.

Jack Audio Server - An audio server that allows for the connection of multiple software instruments to sound card devices, also includes MIDI connections, is an application layer that allows better flexibility to musicians than ALSA. The jack server may use ALSA to connect to sound card devices.

Modular Synthesizer - A synthesizer that is built from different modules. Each module is designed to perform a specific synthesis task. Generally these modules are connected together using patch cords.

AMS - ALSA Modular Synthesizer- A Linux based software synthesizer that is designed to emulate an analogue modular synthesizer.

Module - A single processing device, or function that performs a dedicated audio task. Software modules can be simple functions like generating the inverse of an input signal, or complex, like the modules that emulate analogue filters.

Patch - When working with modular synthesis a patch describes a collection of modules used to create an instrument. The reasoning for this is that modules in the original analogue synthesizers were connected together with patch cords (audio cables) to allow for voltage information to be passed between modules.

Soft Real-time - Describes a type of real time application where a certain amount of error is acceptable for the system, meaning that although undesirable, small errors are not considered catastrophic. For example if some buffers are missed when rendering sound, the system will continue running, even though glitches occur in the audio stream.

Voice - Describes a single note of an instrument.

Monophonic - *one voice* - An instrument that will only play one note at a time, mono meaning one and phonic meaning sound (voice). A trumpet is monophonic. A monophonic synthesizer will only play one note at a time, if a second note is played before the first note finishes the second note played will cut the first note short.

Polyphonic – *many-voices* - An instrument that can play more than one note at a time. A piano is an example of this. A polyphonic synthesizer can play more than one note simultaneously if a second note is played before the first has finished the two notes will be played together. A piano is an example of a polyphonic instrument.

Latency - Describes the time a sound takes to be generated by a computer's audio system. An example would be the time it takes for a sound to be rendered to audio, from an initial strike of a keyboard combined with the time it takes for the synthesizer to generate the sound from specific functions to the data been sent to the audio card. Low latencies are required for real-time audio.

Cycle - A cycle of the AMS is the size in bytes of the audio sample size that is passed between modules and then sent to the audio output to be handled by the sound card. The cycle size defines the size of the audio buffer and represents one dimension of the buffer (see buffer). The cycle size remains constant for the lifetime of the application. Each module will fill the audio buffer specific to the cycle size. The cycle size is what determines the latency of the audio system, the larger the cycle size, the higher the latency.

Buffer - A multidimensional array, e.g. a three dimensional array where one dimension is voice (or note), second dimension is a parameter of a voice (e.g. amplitude, frequency), and the third dimension is cycle size, which is the size of the “audio snapshot” which defines the latency of the system (see cycle). Modules request the output data of the modules connected to its inputs, these requests are handled and added to the modules buffer, which is then sent to the requesting modules as required.

Port (in port, out port) - A port is a possible connection from module to module, an in port accepts another module’s out port, and vice versa. The signal or control flow goes from out port to in port. It should be clear from this that two in ports or two out ports can’t be connected together.

CHAPTER 1 INTRODUCTION

1.1 GENERAL BACKGROUND

Audio Programs have for the most part relied heavily on the assumed speed of the single CPU and many software synthesizers are written to perform sequentially. Generally the sequential application is adequate most of the time but there are sometimes requirements that stress a sequential program that could perhaps be solved using parallel programming techniques. Audio applications have soft real-time requirements. This means that there is a time limit placed on the execution of a program and due to this a limit on how much can be done in the allocated time. Many of the synchronization and parallelization techniques that are normally used to parallelize a program can in certain situations, due to the added overhead in real-time audio programs become too expensive to offer any real performance gains.

In general all audio programs are designed differently, but all rely on the use of an audio buffer, in essence an array of bytes that contains data that is ultimately sent to the sound card to be rendered into sound. The amount of time it takes to generate the data for this buffer is essential to the success of an audio program as far as fulfilling the necessary real-time requirements. So the less computational time it takes to fill this buffer the more chance there is of fulfilling the requirements. The cost many of parallelization techniques has created the assumption that audio programming would not benefit from parallelization and that the use of general synchronization techniques, such as the use of barriers and other synchronization primitives, add too much overhead and that for the most part sequential programming is sufficient [1]. A list of design patterns or guidelines for implementing real time computer music systems can be found in [2].

Needless to say there have been attempts with varying amounts of success to parallelize audio programs. There is a wealth of open source programs that offer programmers insight into the general methods of audio application design and also allows programmers to develop multicore versions of these applications. Again it should be clear that there is no general method to build an audio application but the parallelization techniques used by different programmers can offer insight into possible parallelization techniques and also reveal the difficulties of parallelization of audio applications through programmers failed attempts [15] [16].

Audio programs have generally been written to execute sequentially. There are, however, some aspects of audio programs that could clearly be executed in parallel. An example would be rendering two different sounds, which are then combined at the output, e.g. a piano and a voice. Finding the independent aspects of an audio engine can be difficult since each engine is unique and although there are some similarities and some universal design principles, like the output buffer, the implementation methods are always unique. For example some audio applications have no notion of polyphony [8] while others handle polyphony by duplicating the whole execution path of a single voice [10], and others like the AMS have voice defined in each module of the call graph. There is however a notion of a graph that flows from one processing node in the graph to another, from input to output, in all audio applications.

The ALSA Modular Synthesizer (AMS) is an audio application designed to emulate a modular synthesizer from the 1970's [3]. It is a good example of an audio application that is generic enough to be quickly learned by the average user, but can also be a complex design tool for audio synthesis due to its modular characteristics. The modular nature of the AMS also allows more complex synthesizers to be built and tested for performance gains. The more complex a patch is the more strain is placed on the system. The strain on the system from complex patches is the primary motivation for a parallel engine, which may offer the user a more flexible tool for audio synthesis as the limitations of a sequential version are exposed. Although speedup is a concern of parallelization, the final goal of parallelization is also to gain an increase in throughput. The increase in throughput offers better overall stability when running complex patches. Stability is a reflection of greater throughput because if a synthesizer is able to process more data in a set time them this means that as more modules are added to patches there is less chance of failure in the system. This allows the user to create more complex patches without the concern of failure to meet the requirements of the system, as the application is capable of handing more data.

The AMS engine was designed to run sequentially but its design has specific qualities that are inherently parallel. The goal of this research is to exploit the parallel aspects of the AMS and develop the best methods to design a parallel engine, which meets the requirements for soft real-time audio applications.

1.2 RESEARCH PROBLEM

Based on the above our research problem can be stated as follows:

What are the parallelization techniques using readily available libraries that will result in performance improvements as compared to their serial code versions, taking into account the specific characteristics of audio applications that make parallel program development on heterogeneous computing environments a challenge?

1.3 RESEARCH OBJECTIVES

The main objective of this research is to explore and develop parallelization techniques for audio applications. Although the audio application used in this research is the AMS the desire is to develop techniques that offer, to some degree, insight into general methodologies to parallelize audio applications.

There are two goals motivating the parallelization of the AMS. One is to test the AMS for runtime performance gains. The other is to utilize and develop a portable, and easily adaptable parallelization language for other audio applications.

Runtime performance gains are reflected specifically in relationship to speedup and throughput. The ability to execute audio applications faster means lower latencies can be achieved, at some point though the physical characteristics of a computer and audio hardware may limit speed up. Although speed up is a desirable reason to design a parallel engine the other goal of increased throughput, achieving more computations in the same amount of time, which is the result of running more complex patches, is another possible performance gain from parallelization. The increase in throughput also means that the AMS will be more stable when running complex patches. When the term stability is mentioned it is in fact in relation to throughput. The greater the throughput the more stable a patch will be.

Using portable cross platform parallelization techniques is also another objective, although the AMS is specifically designed to run on the Linux platform, the C++ Thread Library [4] is used to parallelize the engine in order to expose the possible use of these techniques for other audio applications. The new C++ standard thread library is chosen as our development library of choice as compared to other choices like Linux Pthread [5], TBB [6] or OpenMP [7] due to the following reasons: portability (e.g. as compared to Linux Pthread), more flexibility in lower level thread control and efficiency, and easy availability on most commodity platforms. As an example: we opted for a Thread Pool based solution for real-time performance reasons (section 3), since it does not cause extra overheads of thread creation and termination. As an alternative, TBB offers a fork-join based solution in its "parallel for" construct, which is found to be not so suitable for real-time solutions[14]. OpenMP also uses the fork-join paradigm and offers a much higher level of abstraction to the programmer, which is not suitable for our requirement of fine-grained thread control.

Implementing the parallel engine in C++ in the newer 2011 Thread library offers Programmers the ability to use simply designed and readily accessible libraries. The ease of use of these C++ Libraries however doesn't diminish the amount of flexibility offered to the programmer because there is enough diversity within the Library to allow the programmer different techniques and optimization methods in order to obtain better results when compared to using more generic application interfaces, such as OpenMP.

1.4 THESIS STRUCTURE

The rest of this thesis is organized as follows. In Chapter 2, several audio applications are introduced that have been parallelized with varying degrees of success, In Chapter 3, the basic design of the AMS engine is discussed, with specific attention to the basic of audio application design and how this is implemented in relation to the AMS. Chapter 4 discusses the parallelization techniques for the AMS and some implementation decisions. These techniques are further elaborated in Chapter 5 with focus on the different implementation details developed from the techniques discussed in Chapter 4. Chapter 6 discusses the experiments and their results. Future work and the conclusions of the experiments are discussed in Chapter 7.

CHAPTER 2 RELATED WORKS

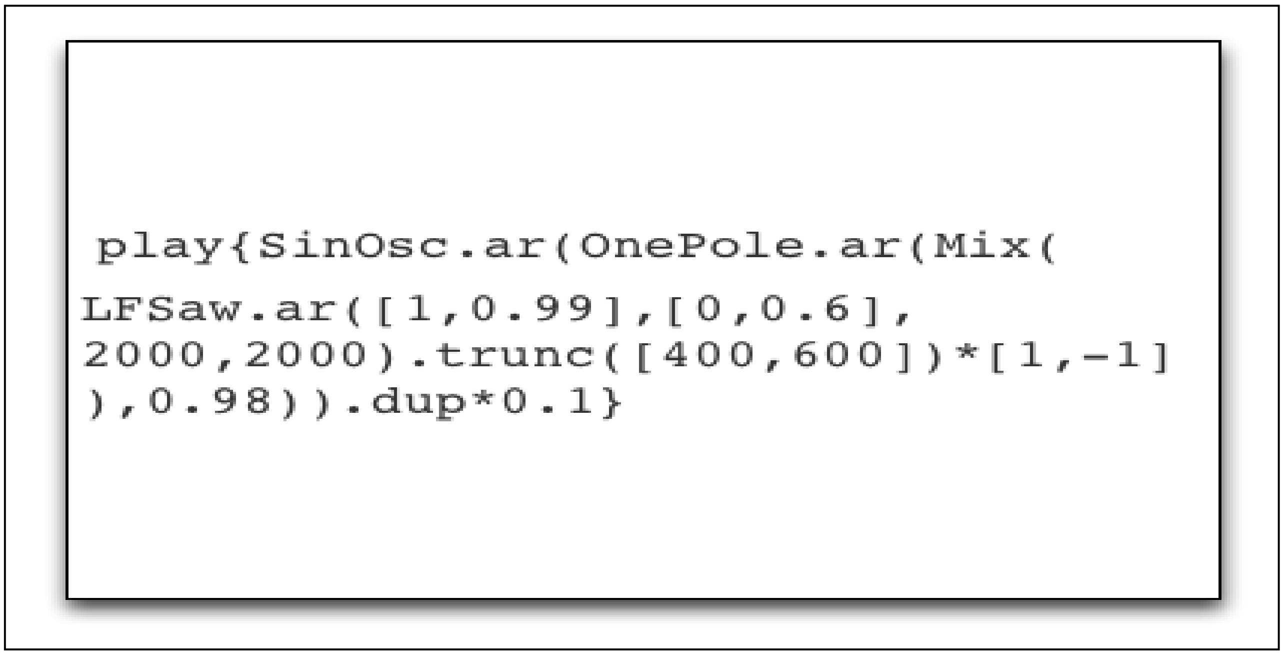
Audio Synthesis Software covers a broad range of tools to generate digitally synthesized sounds, These range from low level text based programming languages to software synthesizers that emulate the traditional hardware electronic synthesizer. These emulations are known as software synthesizers or soft-synths. Apart from the simple soft-synth that emulates a single instrument most programming languages and environments are modular in nature, allowing sound designers to create unique instruments using different modules. This flexibility, that puts less restriction on what can be achieved using modular style programming, is what makes many of the programming environments attractive to users but in turn can over tax the system when too much demand is put on the system's resources.

Programs like the AMS are designed to emulate the modular synthesizers of the 1970's [3] and can be considered high level as far as programmability is concerned. There are also programs like SuperCollider [8] and Faust [9] that are text based programming languages that offer a lower level view of digital synthesis. Pure Data [10, 11] another programming environment sits between the more high-level software synthesizers and the text based programming languages.

In general the control flow of these modular software synthesizers is from input to output where each module or object performs some type of digital manipulation of its input signal and then routes these changes to its output. These programs, however, are all designed differently. Each application has a unique interface and is programmed accordingly and there is no standard method to the design an audio engine. Faust, SuperCollider and Pure Data have been adapted to run as multi-core applications and we can assume that these techniques have also been adopted for use in commercial software.

2.1 SUPERNOVA– A PARALLEL SUPERCOLLIDER ENGINE

SuperCollider [8] is a text based audio programming language whose implementation follows the client server paradigm. The server is written in C++ and can be considered the audio engine. It is also referred to as the *scsynth*. The client application is written in a derivative of Small Talk and is the controller of the server. The client defines and invokes audio events that are in turn generated on the server and routed to the audio output. The basic sound generating functions are known as Unit Generators (UGens), and the events that control these UGens can be described simplistically as patterns defined by the client side language [8].



```
play{SinOsc.ar(OnePole.ar(Mix(
LFSaw.ar([1,0.99],[0,0.6],
2000,2000).trunc([400,600])*[1,-1]
),0.98)).dup*0.1}
```

Figure 1: SuperCollider Code - modulating the pitch of a single sine-wave oscillator. [12]

The basic modular structure of SuperCollider's Audio Engine and UGen paths are based on the concepts of Nodes in a Graph and is described below.

“There are 2 subtypes of nodes, Graph and Group. Graph is so named because it executes an optimized graph of UGens. It can be likened to a voice in a synthesizer or an “instrument”... The Graph type implements the SuperCollider concept of a Synth Group is simply a container for a linked list of Node instances, and since is itself a type of Node, arbitrary trees may be constructed containing any combination of Group and Graph instances.”[13]

SuperCollider’s engine is structured like a tree where the nodes represent a collection of synths, also known as a Group or a Graph. These Graphs or Groups could be seen as the modules of a synthesizer each representing a specific sound generating or transforming function. The tree defines the order of execution of the nodes, starting at the root of the tree. This tree like data structure is very similar to the basic design of a modular synth where each node is a module and the tree is defined by the connections between the modules.

Supernova is the parallel implementation of SuperCollider [14, 15], which extends the concepts of Graphs and Groups by introducing Parallel Groups and Satellite Nodes. Parallel Groups are UGens that can be run in parallel, and Satellite Nodes are Nodes in the Graph that have only one previous dependent and therefore may be run independently of other elements of the Graph. The dependencies and parallel groups are resolved by topologically sorting the nodes contained within the Audio Graph.

The performance gains through the Supernova extension of SuperCollider are at times nominal and at other times great depending on the use case. For certain tasks Supernova performs better than SuperCollider and for some use cases achieves linear speedup, but the overhead of node manipulation and generation of the DSP queue are hot spots in the Supernova server [15].

2.2 FAUST

Faust (Functional Audio Stream) [9] is another text based programming language. Faust as the name suggests follows the paradigm of functional composition. Written specifically for audio synthesis, Faust is modular in design where each module represents an audio function or process. These functions are combined using composition operators to generate more complex synthesizers. In simple terms, each block of code takes an input and transforms the input through some functional compositions into an output. Faust uses an intermediate compiler, simply called the Faust Compiler, which compiles these functions into optimized C++ code and it is during the compilation process that the user can explicitly instruct the compiler to search for ideal sections of the code that could be parallelized.

Faust implements two parallelization techniques the first been vector processing and the other parallelization through OpenMP [7]. The vector scheme attempts to simplify the auto-vectorization of the C++ compiler [16], and the parallel scheme analyzes the dependencies of these vectorized loops and adds OpenMP parallel paradigms to these loops.

2.2.1 Vector Code Generation

The vectorized code generation utilizes the auto-vectorization feature of most C++ compilers. However, the developers of Faust found that the automated vectorization feature of the C++ compiler didn't vectorize the code to optimal performance when the code contained complex instructions within a loop. The Faust Compiler, therefore, performs an intermediate step of loop simplification by splitting more complex loops into smaller loops. This simplifies the task of auto-vectorization by C++ compiler. The Faust compiler analyzes the already compiled scalar code to find loops that are good candidates for vectorization. By reducing the scope of each loop, the generated C++ Code is then more easily vectorized using the auto-vectorization option available in most C++ compilers.

2.2.2 Parallel Code Generation

Parallel code generation in Faust is built upon the vectorized code generation, and uses OpenMP to generate code that can be executed in parallel using multiple threads and the fork-join paradigm. The Faust compiler generates a graph of the code blocks for each function. This graph is a directed acyclic graph where each node represents a function and an edge represents control flow. The graph is topologically sorted to determine the nodes that can execute in parallel. The parallel elements of the graph are executed using the OpenMP directive "#pragma omp section" which divides the parallel sections among a defined number of threads, generally the number of threads being the number of CPU cores. OpenMP by default creates a barrier at the end of each parallel statement and this barrier synchronization waits for all executing threads to complete before continuing with the next function in the control flow.

2.2.3 Performance

Performance of the Faust engine was measured for scalar, vectorized and parallel implementations using GCC and ICC (Intel C++ compiler). One performance measure was memory latencies where the input was simply copied to the output and the throughput is measured. Both scalar and vectorized versions have similar performance metrics. No real gain is achieved using the vectorized code, but the parallel versions perform disastrously. In other benchmark tests, where more computational stress was put on the engine, Faust showed better performance results with speedup of over 2 for OpenMP compiled code. It was observed that sequential code was more efficient for simple computations, and the engine could only benefit from parallelization when heavy computational demands were imposed on FAUST. These tests show definite improvement for both parallel and vectorized algorithms, but markedly when using the ICC [16].

2.3 PURE DATA

Pure Data [10, 11] is a GUI based music programming environment based on the data flow model, where objects are connected together using virtual cables to and from inputs and outputs of one module to the other. Pure Data modules represent a lower level of synthesis than traditional modular synthesizers but the modular nature of its design is still apparent due to the fact that objects inputs and outputs are connected through the use of virtual cables, much like in a modular synthesizer. These modules or objects represent sine waves, *biquad* filters, mathematical functions and comparators to name a few. Since Pure Data is based on the data flow model it could be parallelized using the same methods as Faust or SuperCollider. The flow of control from on module to the next could be topologically sorted to expose implicit parallelism. In Figure 2, Handclap 2, there is a clear section of the patch that could be executed in parallel. Reading the graph from top to bottom there are 4 filters, a lowpass filter (lop~) and three bandpass filters (bp~) that are clearly independent in the flow of control. The developers of Pure Data, however, chose the pipeline design instead [1].

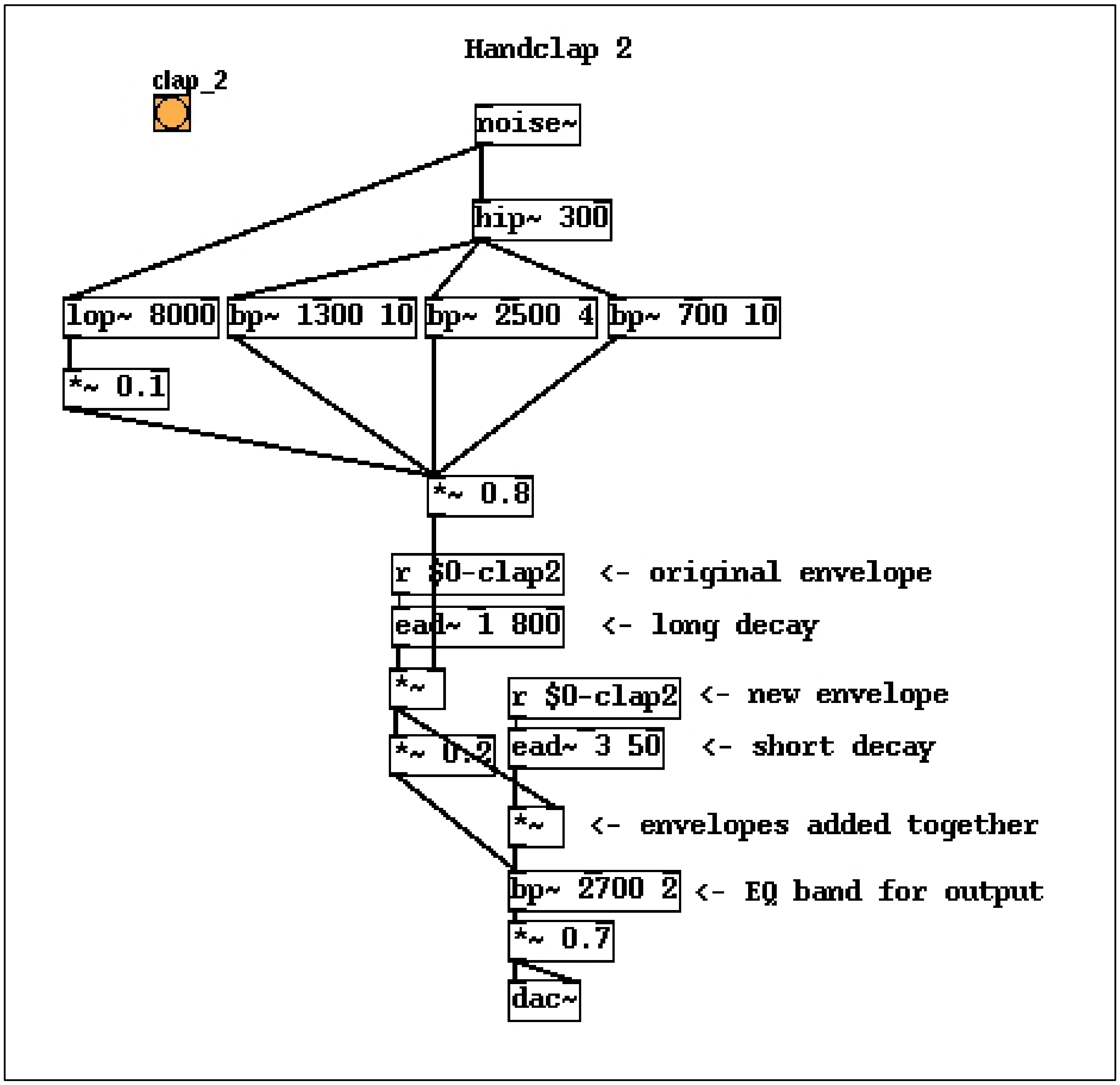


Figure 2: A Pure Data patch showing how objects are connected using virtual connections much like the cables of a modular synthesizer.

In order to parallelize Pure Data, the developers introduced the ~pd object. The ~pd object creates a new instance of Pure Data within the main GUI window which is already running an instance of Pure Data. The new instance created by the ~pd object is an exact replica of the Pure Data engine. The system scheduler sees and treats each ~pd object as an independent process, each with its own user-defined modular graph. These objects are connected as a pipeline, similar in concept to a graphic's rendering pipeline. It was found that this way of parallelizing Pure Data is not scalable since each new instance of the ~pd object adds another 10 cycle clicks to the output, making the pipeline unusable after creating only a few ~pd objects in the pipeline [15].

The previous discussion shows that, much like other programming paradigms, audio programs are designed differently and each requires a different technique of parallelization. Some of these techniques succeeded and others failed to deliver. The next Chapter discusses another high-level audio synthesis engine, which is different from the previous approaches in its modular design and implementation, and to the best of our knowledge no attempt was made to parallelize it in the past.

CHAPTER 3 THE ALSA MODULAR SYNTHESIZER

The ALSA Modular Synthesizer (AMS) is an Open Source Project, initially developed by Matthias Nagorni [3]. It has undergone several updates but remains consistent to its original release. It runs on most Linux Platforms, ALSA been short for Advanced Linux Sound Architecture and was specifically designed to emulate the Modular Synthesizers of the 1970's.
The key characteristic of the AMS is that sounds are generated through the interconnections of different virtual modules. These modules each represent a specific function in order to generate or manipulate an audio signal, examples being filters and oscillators, both key modules in a 70's style synthesizer.

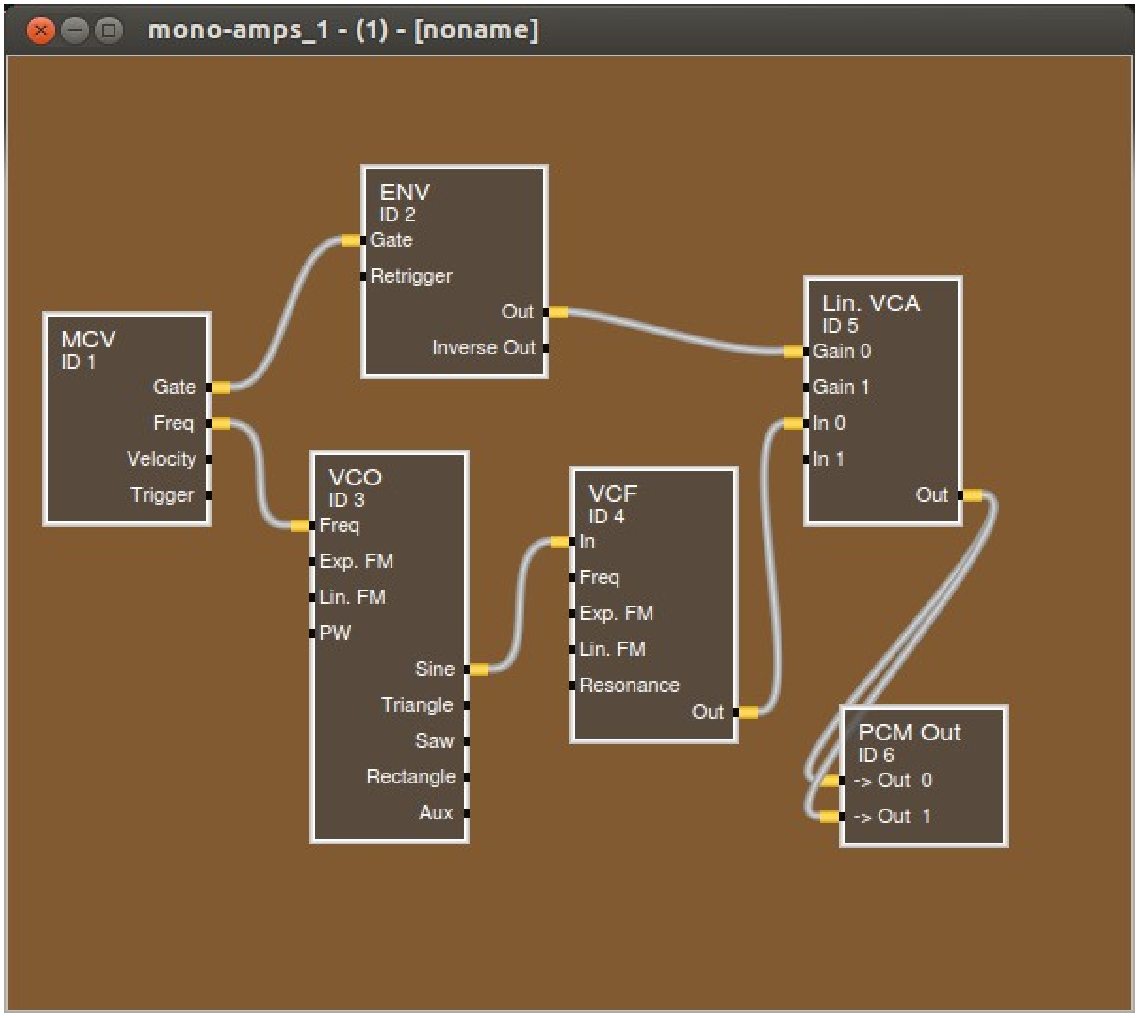
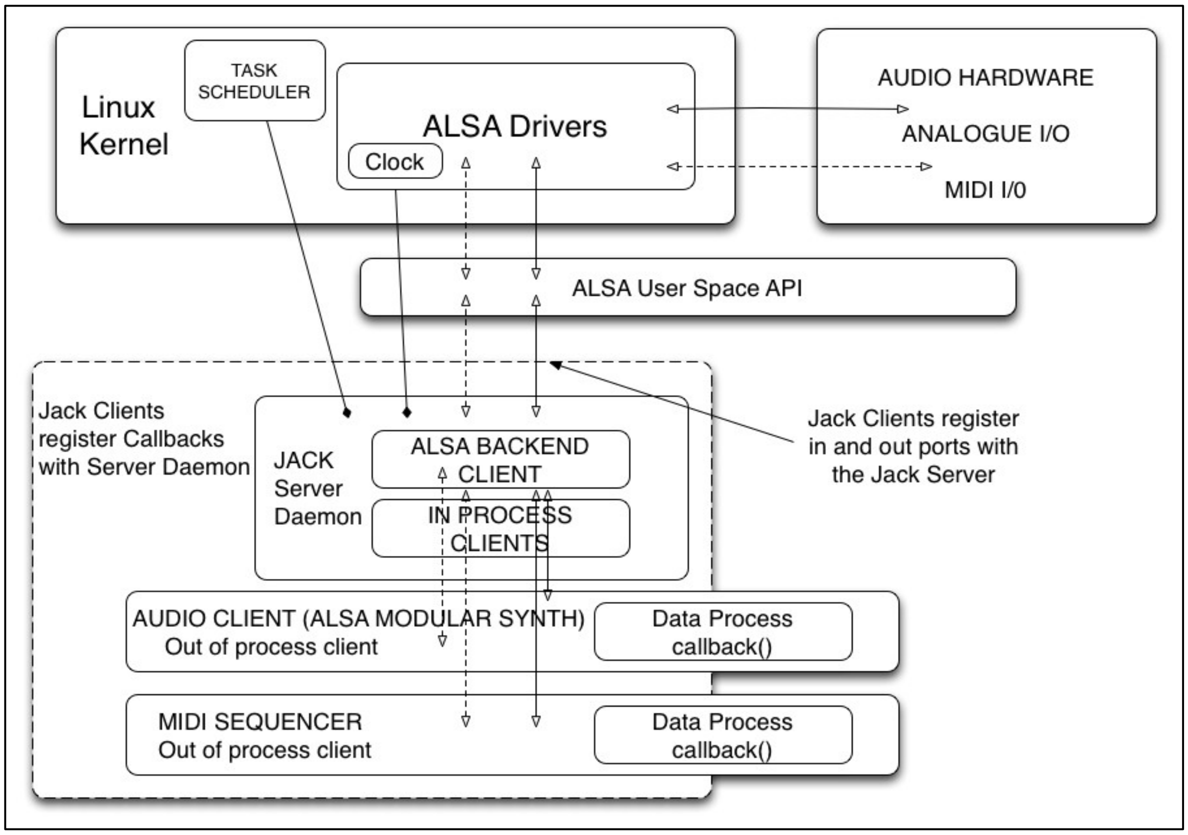


Figure 3: A simple patch made in AMS consisting of the modules MCV, VCO, ENV, VCF, VCA and PCM Out

Modular synthesizers are built through the interconnections of modules. Modules are connected together by the use of patch cords. In the AMS these are virtual patch cords that, in simple terms, are connected from one modules output to another modules input. The final signal path from one module to another to create a more complex synthesizer is known as a “patch” (Figure 3). The more modules and connections used the more complex a sound. A simple patch may sound thin when compared to a sound that uses numerous modules in more subtle ways. Complex patches also allow the designer to generate more than just one sound within a patch.

The AMS can be one of several instruments being used simultaneously to perform a song and therefore it can be run as a plugin along with other instruments. In order for Linux to run different instruments concurrently, these instruments need to be synchronized in some manner so that they play in synch with one another. Linux solution to this is through the use of the Jack Audio Connection Kit (abbreviated as *Jack*) [18]. Jack is a client server based application that runs between the ALSA drivers and the (software) audio instruments. ALSA drivers are part of the Linux Kernel and communicate directly with the audio hardware. Jack sits in between the audio instruments and ALSA. It has the responsibility of synchronizing all the instruments connected to it and sending the synchronized audio to ALSA (Figure 4).



Linux Kernel
TASK SCHEDULER
ALSA Drivers
Clock
AUDIO HARDWARE
ANALOGUE I/O
MIDI I/0
ALSA User Space API
Jack Clients register Callbacks with Server Daemon
JACK Server Daemon
ALSA BACKEND CLIENT
IN PROCESS CLIENTS
Jack Clients register in and out ports with the Jack Server
AUDIO CLIENT (ALSA MODULAR SYNTH) Out of process client
Data Process callback()
MIDI SEQUENCER Out of process client
Data Process callback()

Figure 4: *Basic overview of Linux audio architecture, simplified from Jack diagram. The Jack Audio website [18].*

The AMS in itself is not a complete audio synthesis tool; it requires other external tools to synthesize audio. In essence, the AMS has limited tools to sequence sounds to create a complete song. This means that the AMS requires some external control logic to create complex songs. These external tools communicate with the AMS using the MIDI protocol. MIDI (Musical Instrument Digital Interface) is a protocol that sends control messages to digital instruments [17]. It is important to stress that MIDI alone doesn't generate sounds, but simply sends control messages. The basic MIDI messages sent to an instrument are, note on/ off messages, note number, and velocity (the amplitude of the note). The piano keyboard is the most common way to transmit control data to the AMS.

The use of MIDI isn't however that important to understanding the way that the AMS generates sound. MIDI is processed through a specific module called MCV (MIDI to Control Voltage) but once the data is received from an external MIDI device it is converted and handled for the specific needs of the synthesizers engine, in fact the MIDI keyboard could easily be replaced by a computer keyboard or computer mouse as long as the data is received and interpreted by the AMS to represent note on/ off messages. For example if when the mouse button is pressed a note on message is sent, then it can be interpreted by the AMS as an open gate or value of 1, which when multiplied by another amplitude signal will send the amplitude of the signal to the systems output. One multiplied by an amplitude value is just the amplitude value itself. When the mouse button is released a note off message is sent. The value of note off been 0 which when multiplied by the amplitude of the signal is 0, silence. So the note off value silences the signal by sending a value of zero to system output. It is important to understand that the signal path of the AMS is always being calculated, whether the final value is zero or not. In short one is “make sound” and zero is “make no sound”.

The virtual modules of the AMS are each designed for a specific task, and for the most part have a hardware analogue counterpart. These modules are named to mimic those counterparts like VCO, Voltage Controlled Oscillator or VCA, Voltage Controlled Amplifier. It only takes two modules to make a sound, an output module and a module that generates sound, such as an oscillator module. When these two modules are connected a constant frequency is outputted to the system and sounds a lot like the test tone emitted from a television when a station isn't broadcasting. It is important to understand that the AMS synth engine is always running, and that even if silence is sent to the system output the whole execution path of the AMS is executed with zero values, silence, finally being calculated for the output.

In order to generate sound there is only one module that must be present and that is the output module, “PCM Out”. Of course on its own it doesn't generate sound but this module must be present because it sends whatever input data it receives to the system output. If as mentioned above the VCO module is connected to the input of the “PCM Out” then a continuous tone is sent to the system's output. Obviously to generate more complex sounds it requires more modules.

As a difference from the other synthesizers discussed in the previous chapter, the AMS modules are executed as a call graph. Referring to Figure 3, the input module is MCV and the final output module is PCM Out. However, the entry function of PCM Out is the first function to be called as a (Jack) callback function, which in turn calls the entry function of VCA in the call graph; and so on until the last function call is made to the entry function of MCV. The results of the function calls are propagated in the reverse order of calling, e.g. the first output from MCV will be input to ENV and then to the VCA, the VCA will then call the VCF then the VCO and MCV, which will then return the data to the VCO and so on until the last input is to PCM Out, which produces the final result of the callback. As discussed in the following chapters, one of our techniques of parallelization is to execute this call graph in parallel, at each callback invocation of PCM Out's entry function.

An important characteristic of audio programs is that calculations of audio data are not done one value at a time but in blocks of data, or buffers. Input to each module in the AMS is a multidimensional buffer, e.g. a three dimensional array where one dimension is voice (or note), second dimension is a parameter of a voice (e.g. amplitude, frequency, etc.), and the third dimension is cycle size, which is the size of the "audio snapshot" (Figure 5). These buffers can be as small as 32 bytes and can exceed 1024 bytes, in the AMS the buffer size is defined by the cycle size but the overall size of each module's buffer is defined by the number of output parameters the module has, the number of voices being run by the AMS and the cycle size combined.

The decision of cycle size has two effects. First the smaller the cycle the lower the latency, the less time it takes to calculate the complete audio path. Lower latencies are better as far as performance. High latency means that there can be an audible delay between when an instrument is played and when the sound is heard, which is obviously undesirable. Secondly, however, if the cycle is too small there is more chance that the audio program will not have time to calculate a complete traversal of the audio stream, and incomplete data will be sent to the audio output. This can create audible distortion in the final sound (glitches) and may, in extreme cases, crash the program due to repeated callback without any output. The decision of cycle size can obviously have serious consequences on performance, but as long as the latency is below 20 milliseconds [19], any cycle size is fine.

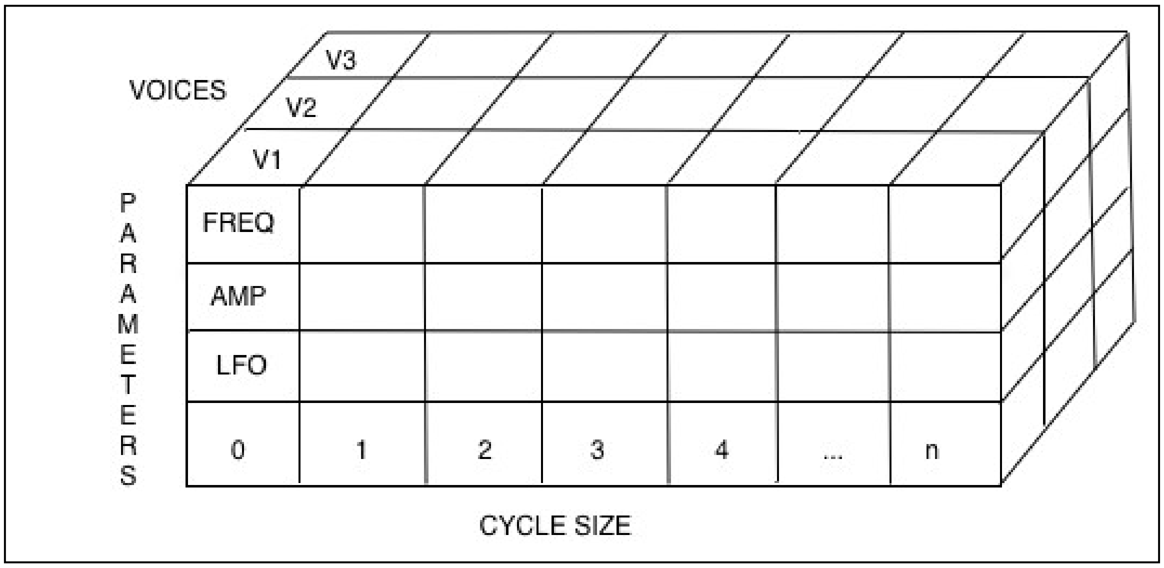


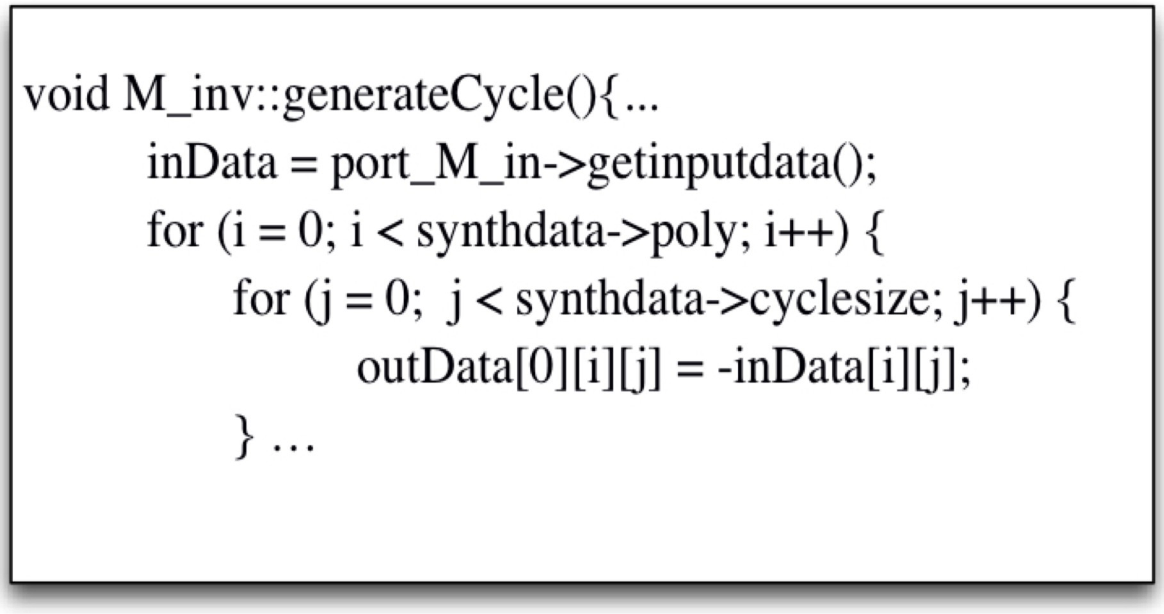
Figure 5: A 3-D buffer in AMS

An important characteristic of the AMS is that it is designed to be polyphonic, i.e., being able to play many notes (voices). Although most analogue synthesizers in the 1970's were monophonic, i.e., playing only a single note, almost all modern synthesizers and soft synths are polyphonic. A monophonic synthesizer only plays one note at a time, hence mono. A polyphonic synthesizer can play more than one note at a time, poly. In a monophonic sound, when one note is playing and another note is played simultaneously the first note is cut short and is no longer present in the sound. On the contrary, polyphony is when two or more notes can be sounded in unison. An example of a polyphonic sound is when a pianist strikes several keys of the piano at the same time. Each key being struck represents a unique voice and the multiple voices are sonically combined to make a single sound.

In the AMS, one of the dimensions of the multidimensional buffer represents the different voices and another dimension represents the different parameters of each voice, i.e., frequency, amplitude, etc. The third dimension represents a (virtual) time, called the cycle size (Figure 5). The number of voices that can be processed simultaneously as well as the cycle size are defined by the user. Obviously, the larger the number of voices and the lower the cycle size, the greater the computational strain on the system.

Each module of the AMS inherits the virtual function *generateCycle*, and it is this function that calculates the buffers of data sent to the output of each module to be read by the next module in the call graph. Since the calculations need to be from the input of a module to the output of a module intuitively the order of execution should be from the first module of the synthesizer to the final module's output. On initial inspection, however, the initial call is made to the final module in the call graph, “PCM out”. “PCM out” then calls *getInputData* on the module connected to its inputs, referred to in the AMS as *inports*. The data however from this module is not returned, but the call to *getInputData* calls *generateCycle* on the connected module. The function calls propogate in the reverse sense until the final module is called. The final module is infact the first module in the call graph and it this module that will calculate the requested data and return it to the module that requested it. So in this sense the order of execution is from the first module in the call graph but these call are propagated in the reverse order.

The function *getInputData,* which finally returns the requested data from connected modules, is key to understanding the call graph, as this function recalls *generateCycle* on the module that is connected to the initial modules input. If the module is connected to a third module then *generateCycle* is called on this module. The cycle of each module is not calculated on any modules until *generateCycle* is called on the final module of the synthesizer, this module will have no connected modules at its input and therefore is the final module in the graph. This module will calculate its internal data and then return this data to the module it is connected to. The final module to process its data in will be the module that initially called *getInputData*



```
void M_inv::generateCycle(){...
        inData = port_M_in->getinputdata();
        for (i = 0; i < synthdata->poly; i++) {
                for (j = 0;  j < synthdata->cyclesize; j++) {
                        outData[0][i][j] = -inData[i][j];
                } …
```

Figure 6: The simplest of modules implemented in the AMS, INV, which takes the data from one input connection and calculates the inverse value and sends it to a single output.

The output data of each module is calculated using an inner and outer loop. The outer loop loops through each voice, and the inner loop calculates the buffer, or cycle for each voice. The fact that the inner loop in the array represents the cycle of a single voice means that these voices can be calculated independently and are only combined at the output to create a single sound. As an illustration, the code sample in Figure 6 shows the *generateCycle* function from the simplest of AMS modules, INV, which takes its data from single *inport*, calculates inverse, and then sends the result to the next module connected to its single output port.

In addition to call graph parallelization already mentioned, the other parallelization technique explored in this research is the outer loop parallelization [20] in each of the *generateCycle* routines. These techniques have to comply with the soft real-time requirement of audio and are elaborated in the following chapters.

CHAPTER 4 PARALLELIZING THE AMS

Considering the design of the AMS there are two possible ways to parallelize the engine. Since the voices of the AMS are independent, represented by one dimension of the 3D buffer discussed in the previous chapter, the first method would be to simply execute the independent voices of each module in parallel. The other option would be to execute the modules that are independent in the call graph. Even in a simple patch there is the possibility that modules can be executed independently. There are some limitations to the second approach because in most patches there will be more dependent sections than independent, whereas each polyphonic voice is independent until combined at the output. There are however dependencies between modules, because the data of one module must be calculated in full before the next connected module's data can be processed. This is discussed in more detail later in this chapter.

4.1 Outer Loop Parallelization

Referring to the code sample in Figure 6, parallel execution of the individual voices initially appears straightforward. The independent voices in the outer loop could simply be executed by individual threads, as there are no dependencies between voices. The problem however is that the overhead of creating and destroying numerous threads for numerous modules would become too expensive to warrant parallelization [2]. If a synthesizer had 8 voices and 12 modules it would mean there would be 96 threads created and destroyed per callback. The designers of Faust encountered this problem when using the OpenMP directive "#pragma omp section", which uses the fork and join paradigm [27], found that it gave limited results [16]. One can assume that the C++ fork and join would offer similar performance results. A simple solution for this is to implement a thread pool. The advantage of using a thread pool is that the number of threads created can be kept to an allowable limit, and threads can be reused for the lifetime of the application.

Of course there are overheads involved when using a thread pool e.g., tasks that are to be executed by the thread pool must be placed into some type of a queue, which must be designed for efficient concurrent access without much of synchronization overhead. The overheads of using a thread pool should, however, be much less than the overhead of dynamic thread creation. In our approach, the Boost lock-free queue [21, 22] was used when ever possible to allocate tasks to the thread pool, but when using smart pointers a lock-based queue was used.

The next issue arises in defining the concurrent tasks for the thread pool. Obviously each task executes an iteration of the outer loop, representing each independent voice. In a C++ based implementation, each such task is an object (more precisely, pointer to an object) deposited to the lock-free queue. Each such concurrent object should simply be inherited from the parent module (object) whose *generateCycle* function is currently being executed. Complexity only arises when defining the tasks that need to be sent to the queue. Each module must implement the *generateCycle* function, which for the most part separates each voice in the outer loop. Having each thread execute this function seems to be a simple matter of sending an inherited object to the thread pool to execute this task. So a unique object of each module type with access to its specific *generateCycle* function could be sent to the thread pool.

Complexity arises due to the following design issue of AMS: the design of AMS incorporates both GUI and the DSP calculations within the same object; hence creating a new module object, even a derived one, would create a GUI representation of the new object as well. This is not appropriate. C++, however, offers a solution to the previous problem through the implementation of a *friend class* for each module. The *friend class* of a module is a lightweight class that can access the private data and functions of the module that are necessary to perform the required audio calculations for each voice. The new *friend class* therefore becomes a lightweight object that only uses the data of the module that is necessary to calculate each individual voice. The *friend class* can be seen as a class, which although not a derived class still has access to the private data of the other class.

The use of the *friend class* as lightweight object solves one problem but the cost of object creation during each callback is still costly. Initial tests showed that creating objects each time the outer loop is called slowed execution time down considerably, much like creating threads for each callback. The *friend class* objects are therefore created when the module is instantiated, and reused during the lifetime of the program, much like the threads in the thread pool. Each module contains a vector of objects that represent the individual voices of the module. These are in fact pointers to objects that are sent to the thread pool that the threads can then pick up to calculate the data of the individual voices of each module.

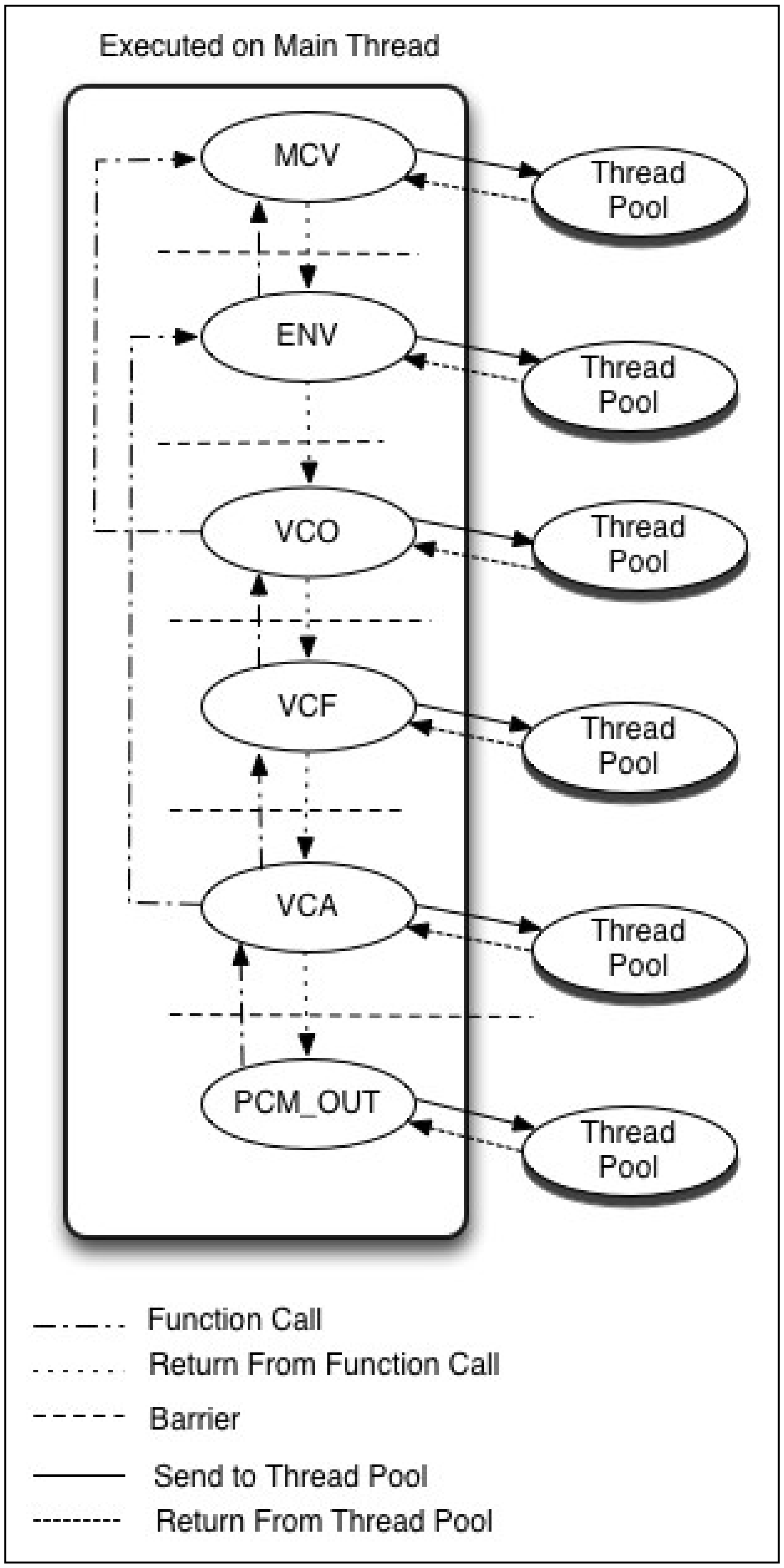


Figure 7: Execution of a single callback

Next comes the issue of synchronization. Referring to the call graph in Figure 7 and based on the discussion in the previous section, the callback invokes *generateCycle* of the output module PCM Out, which is subsequently propagated to the *generateCycle* of the first module, MCV, which has no further input modules connected to it. So MCV does its calculations and returns the results to the modules that initially called *getInputData*; and this way computations proceed in the reverse direction of the order that the calls were made; the final computations are done in PCM Out which produces the final result buffer which is sent to Jack as the return value of the callback.

The *generateCycle* methods of all modules are executed by the main thread, while the voices in the outer loop inside each module are calculated by the thread pool, which can independently write to the output buffer of an output port. The main thread must synchronize with the thread pool (e.g., barrier synchronization) so that all the threads have finished their computations of a specific module to allow computation to proceed to the preceding modules (Figure 7), i.e., their *generateCycle(s)*. Referring to the figure, once MCV finishes calculations then only the prior modules, VCO and ENV, can perform their computations because of their inherent data dependencies through the connected ports; and so on.

Our baseline solution involves "no synchronization" between the main thread and the thread pool: we call it the *naïve implementation*. The problem with the naive implementation is that there is a clear dependency between modules because the data from one module must be calculated before the next module can calculate its own internal data. This dependency is from the output data of the first module to the input data of the next module. Since *generateCycle* is a void function the main thread doesn't wait before continuing to send objects to the thread pool regardless of whether or not the previous module has completed all the necessary calculations. The data race occurs when a thread reads a modules input data before the output data of the previous module is complete.

Obviously, the naïve implementation will work correctly without any critical race condition if and only if the thread pool contains a single thread because the single thread will execute each object in the order they were placed in the thread pool. The data race occurs when there is more than one thread accessing the data because the threads in the thread pool will not execute the voices in same order that they are placed in the thread pool. Since the naïve implementation works with only one thread in the thread pool, several alternative solutions were proposed and tried to resolve the data race with multiple threads in the thread pool, while satisfying the soft real-time requirement. Some of these solutions are discussed in the following;

1) The first solution involves using an *atomic counter and signal* mechanism provided by the 2011 version of C++ Standard Library [23]. Each thread, in the thread pool, which executes a voice, decrements the atomic counter. The thread which makes the counter zero (i.e., the last thread executing a voice) signals the waiting main thread, so that the main thread can proceed to the preceding module(s).
2) The second solution involves using *standard futures* [23], as provided by the C++ Standard Library [4]. The return value of each method, calculating a voice inside a module, is changed to a Boolean and is assigned to a future variable. It should be noted that each of these methods (in friend objects) is executed by a thread in the thread pool. The main thread checks each of the Boolean return values inside a module and blocks if needed before proceeding to a preceding module. Two different granularities were evaluated: (1) *fine-grained* where each method calculates one voice and there is a future variable associated with each method; and (2) *coarse-grained* where each method calculates more than one voices and is associated with a single future. Obviously the number of futures and hence the synchronization overhead goes down in the coarse grained version, however at the cost of increased computational granularity per thread.

The experimental evaluations from both of the previous techniques are discussed in Chapter 6. Several solutions are tried to resolve the data race issue, and these are discussed in Chapter 5.

4.2 Call Graph Parallelization

The second parallelization, the Call Graph Parallelization, involves identifying and executing concurrent paths of the function call graph in parallel. There are, even in a simple patch, some modules that can be executed in parallel. In the case of Figure 3, The Simple Synth, the two concurrent paths are (1) PCM Out – VCA – ENV – MCV, and (2) VCF –VCO – MCV. If we trace the call graph two or more modules have independent execution paths if the initial call to the modules comes from the same module but from the calling module there is a branch to different modules. Each concurrent path in the call graph can execute on a separate thread and parallel execution time is governed by the *critical path length* of the call graph (Figure 8). Two important issues arise in the parallel execution of the concurrent paths and are discussed in the following.

Firstly, in the Sequential version of the AMS the module's *generateCycle* function calls *getInputData* on connected modules. The module then waits for the return value from *getInputData* before continuing execution. For the parallel version to work synchronization is required at the branching point of two concurrent paths, the call to *getInputData*. This creates a different situation when compered to the outer loop parallelization, which had no return values, and required the addition of a synchronization barrier in order to avoid data races. The callback routine relies on a return value before it can continue correctly. This return value can then be used as a barrier to synchronize the call graph. For example, referring to the two concurrent paths in Figure 1, the paths branch at the module VCA. Synchronization is needed inside VCA to collect the results of the two concurrent paths in the call graph, before (the body of) *generateCycle* of the VCA can be computed (refer to Figure 9).

In practice, in our implementation, this synchronization is achieved by using *standard futures* which part of the C++ Standard Library. The following are the sequence of actions: thread 1 executes path 1; thread 2 executes path 2 where thread 2’s execution return value is assigned to a future; thread 1, upon completing MCV and ENV, blocks on the future (inside VCA) if thread 2's execution result is not yet available, otherwise it proceeds to the computation of *generateCycle* of VCA and subsequently proceeds to calculate PCM Out.

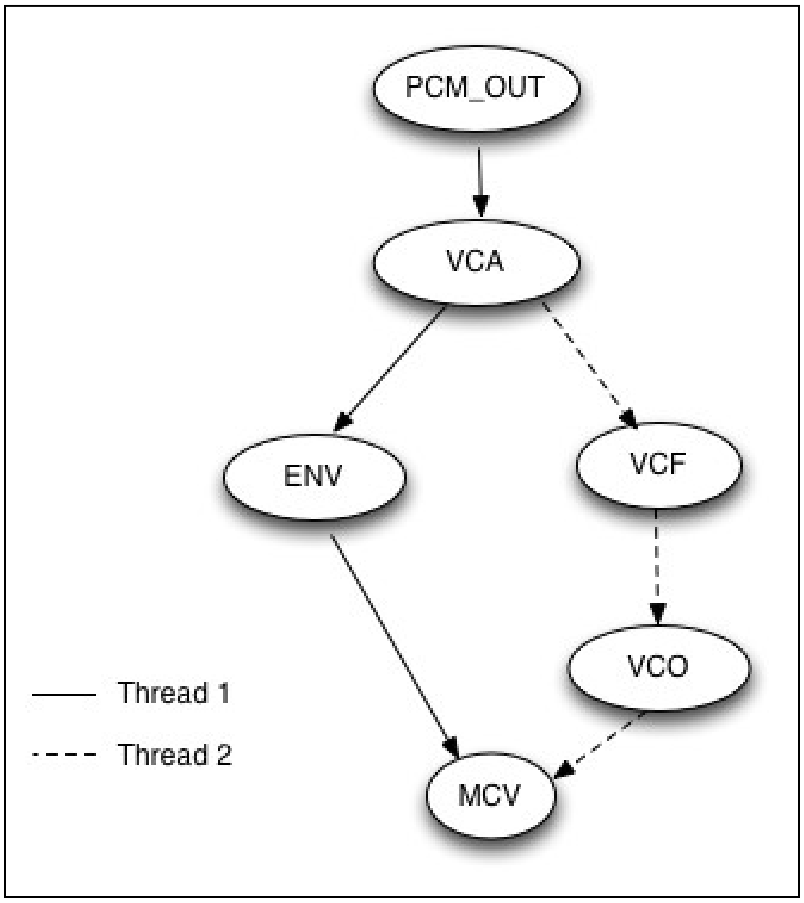


Figure 8: Concurrent Paths in the Simple Synthesizer

Secondly, synchronization is required at the meeting point of two concurrent paths. For example, referring to the two concurrent paths of Figure 6, the paths meet at the module MCV. However, *generateCycle* of MCV must be executed only once by one of the paths and not both. To resolve this, each thread reaching MCV first locks the module and then checks for a Boolean flag. If the flag is not set it means that *generateCycle* has not yet been executed, the thread then executes *generateCycle* and sets the flag to true. The subsequent thread(s) reaching MCV would find the flag set and can get the result right away (Figure 9). The division of work for the call graph parallelization is then between the main thread and the threads in the thread pool. In Figure 9, Path 1 would be the main thread and the second path would be executed as a standard future, which would be retrieved from the thread pool by any available thread.

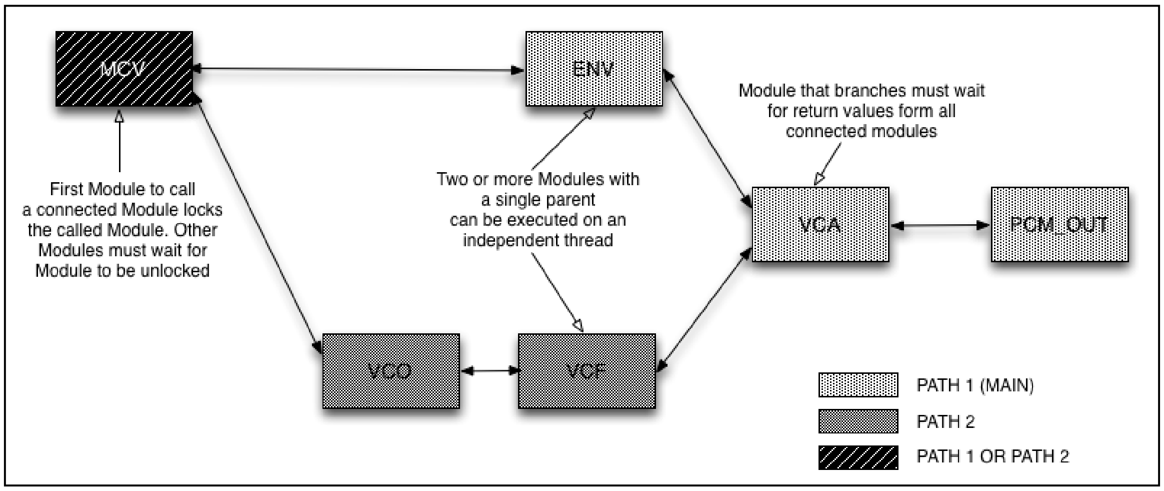

MCV
ENV
VCA
PCM_OUT
VCO
VCF
First Module to call a connected Module locks the called Module. Other Modules must wait for Module to be unlocked
Two or more Modules with a single parent can be executed on an independent thread
Module that branches must wait for return values form all connected modules
PATH 1 (MAIN)
PATH 2
PATH 1 OR PATH 2

Figure 9: Synchronization points between the concurrent threads

Synchronization overhead is counterproductive to the soft real time requirements in audio synthesis and hence two different versions were evaluated to find a compromise: (1) *fine-grained* where each concurrent path is sent to the thread pool and the return value of each branching path is assigned to a *future*; (2) *coarse-grained* where the number of futures is limited in order to reduce the overhead of searching for and creating standard futures. This is done order to optimize the call graph parallelization. Futures are still sent to a thread pool but each future may represent more than one independent path. The experimental evaluations with both these versions are discussed in Chapter 6.

4.3 Combining The Two Techniques

The next obvious way of parallelizing AMS is combining both the outer loop parallelization and call graph parallelization techniques discussed in the previous two subsections. The results from combining these techniques are discussed in Chapter 6.

CHAPTER 5 PARALLELIZATION TECHNIQUES

There are various ways to implement the parallelization of the outer loop and the call graph. There is a certain logic of how each implementation lead to the next proposed solution to parallelize the AMS. This Chapter discusses in detail these solutions and the specific implementation details used.

The C++ Standard Library, introduced in 2011, offers the programmer numerous libraries specifically designed in response to an ever-growing need to write parallel programs for multi-core environments. Some of these libraries have been used extensively to parallelize the AMS, e.g., the Thread Library, the Standard Future Library, and other libraries as required, an example being *bind*, which is needed to wrap functions to be used as futures.

Most of the solutions use a thread pool, consisting of a vector of threads and a FIFO queue. The use of a thread pool allows threads to be reused, meaning thread creation can be kept to a minimum. Although there is the added overhead of using a thread pool, the assumption is that the use of a thread pool is less expensive than creating numerous threads during the callback routine (Chapter 4).

The implementation of the thread pool means there must be a concurrent queue where tasks are placed that the threads can then retrieve and execute. The Boost lock-free queue is used when ever possible. The reason for using the Boost lock-free queue is because lock-free data structures are considered the ideal data structures to use for audio software [2] because they don't use traditional synchronization techniques, like locks, which add extra overheads. Conversely, locks are considered poor choices for real-time requirements. The Boost lock-free queue is a FIFO Queue that is thread-safe but doesn't use locks to ensure concurrent behavior. Instead it uses the atomic compare and exchange algorithm.

Unfortunately, by design, the Boost lock-free queue cannot be used when combined with futures and shared pointers. The compromise due to this incompatibility was to use a traditional lock-based FIFO Queue wherever futures were used to parallelize the AMS. As mentioned above, there is an assumption that the use of locks should be avoided as much as possible when designing audio programs. Historically locks may have had poor performance metrics, but through optimization and the modernization of the Standard C++ Library lock-based data structures may perform just as well as lock free algorithms. Hence the assumptions about lock based data structures that may have been justified 10 years ago may not be true now [24]. For our use case the performance of both data structures are comparable.

When testing the lock-free and lock-based queues performance does suffer slightly using a lock-based queue. Performance gains when using the Boost lock-free queue are slightly better, when a limited number of tasks are sent to the thread pool. Performance results are much closer between the lock-free and the lock-based queues when a large number of elements are sent to the thread pool.

Neither queue, however, is optimal because threads are constantly polling the queue to access data even if the queue is empty. Each thread in the thread pool because of this polling will use 100% of a CPU's resources. The solution to stop the threads from polling the queue is to have the queue notify the threads when there is data added to the thread pool. This means that the threads maybe sleeping until data is added to the pool, adding an extra overhead when waking sleeping threads. The addition of signals that notify waiting threads when there is data available in the queue resolves the polling issue and does lower the CPU usage per thread, but degrades performance considerably due to the additional time it takes for threads to wake before executing data. This degradation in performance made this type of queue unusable.

There were also several experiments using a Single-Producer/Single-Consumer (SPSC) lock-free queue, which can be used in conjunction with standard futures and shared pointers. As the name suggests the SPSC queue requires that there be only one producer and one consumer exchanging data. For the AMS this means that the main thread acts as the producer and only one thread running in the thread pool, acting as the consumer. This design had limited practical use because most of the tests involved using more than a single consumer, i.e., more than one thread in the thread pool. The SPSC queue did however perform similarly to the Boost lock-free queue.

The test, therefore, were performed if possible using the Boost lock-free queue. Specifically if raw pointers could be used to represent data been sent to the thread pool. This means that the Boost lock-free queue was used for the outer loop parallelization if futures were not been sent to the thread pool. All tests using the C++ Standard Futures use a lock-based queue so that shared pointers could be used.

5.1 Outer Loop Parallelization

The initial parallelization method, the *naive method*, uses no barriers or synchronization techniques to control the execution of the individual voices. As mentioned in the previous Chapter, objects are sent to the thread pool and executed by individual threads regardless of any data race that may occur. This method works when there is only one thread running in the thread pool. Since the objects are sent in FIFO order to the queue, the single thread will execute each object in the order they are inserted avoiding any data race that may occur. When two or more threads are executing in the thread pool then a data race occurs, specifically causing a read-before-write inconsistency (Chapter 4).

Interestingly when the AMS was tested running more than 32 voices there occurs a type of load balancing and the program executes correctly, even when running more than one thread in the thread pool, while still using no synchronization primitives. The reason for this is perhaps due to the timing as a result of load balancing where the main thread executes close to the same speed as the threads in the thread pool. So the read before write error doesn’t occur. The correctness of the program however isn't resolved and there is no guarantee that increasing the number of voices actually solves the data race; it works merely by chance.

Another problem that arose when testing the naive method is that it was not possible to record the execution time of the AMS because once the objects are sent to the thread pool they are no longer synchronized with the main thread. Timing each thread’s execution time was not possible without adding some type of a barrier that would synchronize the executing threads in order to measure the callback routine’s overall execution time. When a barrier is added to the main thread, causing it to wait for the execution of all the modules in the call graph, the times recorded are less impressive than without the barrier. This solution however offers a more accurate measure of the execution times and still offers speed up when compared to the sequential version. This barrier doesn't resolve the data race between the threads in the thread pool, as it merely stalls the main thread from completing the call back routine until all the threads in the thread pool have finished.

The solutions to avoid the data race between modules discussed in Chapter 4 are: 1) To have the main thread wait for each module's voices to be executed by the threads in the thread pool before continuing to the next module. This is achieved by using a barrier, specifically an atomic counter and signal. 2) Bind the voices of each module to a standard future before sending them to the thread pool. Both of these solutions use a barrier to force the main thread to wait for all voices of a module to be executed before continuing. These solutions resolve the data race between read and write errors occurring between connected modules but both add extra overheads.

The first solution requires using a barrier that forces the main thread to wait for all the voices of a module to be executed before adding the next module's voices to the thread pool. The C++ Condition Variable is used to implement the barrier and an atomic counter is used by the threads in the thread pool. As each thread in the thread pool executes a voice of a module it decrements the atomic counter. The counter is initially set to the number of voices being used by the AMS. When the counter reaches zero, the executing thread signals the main thread, notifying it that it may continue execution. The main thread on receiving the signal then starts adding the next modules voices to the thread pool. This resolves the data race but adds a barrier that slows down the overall execution time, when compared to the naive version, but as will be seen in the experimental findings, offers a noted increase in throughput.

The second solution, which sends the voices of a module to the thread pool as standard futures, creates a similar situation to thread and object creation overheads in the initial investigations with parallelizing the outer loop (Chapter 4). A solution to avoid the overheads of creating numerous standard futures involved reusing the same future by placing bound functions in a vector that were then instantiated at the same time as the module. These functions were then sent as futures to the thread pool at run time. This minimizes the number of futures created during each execution of the function callback routine. Unfortunately, due to the way standard futures are created, this didn't fully resolve the problem because, although the bound function could initially be created, the futures still needed to be instantiated for each call back.

The initial solution, where each voice of a module was mapped to a single future (fine-grained), performed worse than the sequential version as there were too many futures generated during execution causing the AMS to crash even when running the simplest of patches.

A work around was proposed that would reduce the number of futures instantiated by making each future represent more than one voice, in effect creating a coarse-grained solution. This would reduce the number of futures needed for each outer loop by sending a block of voices to the thread pool bound to single future. The data race would still be avoided since the main thread would have to wait for all the futures to return before executing the next module's code. The expected outcome is that there should be a performance gain by reducing the number of futures. This however is not the case and in fact this solution is no better than the original fine-grained solution, perhaps because, although there are less futures used in the solution, there is more work allocated to each individual thread in the thread pool.

5.2 Parallelization of the Call Graph

Parallelizing the call graph requires a different approach compared to the outer loop parallelization. The method to parallelize the call graph is not as simple as the outer loop parallelization because the independent paths between modules must be found. Unlike the simple task of executing the outer loop of the *generateCycle* functions, which are static regardless of the patch or module, each path in the call graph is unique to a patch and may change as the patch is modified. The call graph parallelization must then be adaptable to the changes in a patch, whereas the voices of each module are always independent regardless of the design of the patch. For the call graph parallelization: 1) the call graph needs to be traced on the fly or 2) the order of execution and the independent paths must be represented in some static form so that the paths can be executed by individual threads. The second choice would mean that every time a patch is changed, theses changes would need to be updated to represent the new paths in the call graph. The design of the AMS is such that patches can be modified while the program is running. Patch cords can be plugged and unplugged while the synthesizer is still generating audio. This type of real time editing means that any static form representing the parallel execution of the call graph would need to be updated quickly in order to not disrupt the flow of editing a patch. Updated changes would also need to be handled in the first solution but, as will be shown, updating a patch *on the fly* is trivial (Figure 10). The overhead and added complexity of the second choice made the *on the fly* version a lot more attractive.

The important functions that are executed during the call graph routine are *getInputData* and *generateCycle* of each module. Chapter 4 describes these routines in more detail but importantly, *generateCycle* calls *getInputData* on each of the modules *inports*, which in turn calls *generateCycle* on any connected module. If there is an independent path then the function call to *getInputData,* is bound to a standard future and sent to the thread pool. The return value from the future is the data from the function *getInputData,* which is returned to the initial module that created the standard future before the output data of the module is calculated (Figure 10).

The 2011 C++ Standard Library introduces new synchronization primitives that offers the programmer simpler solutions for multi-threaded applications. The Standard Future Library offers a specific solution to synchronize the independent threads used in the call graph parallelization described above. Futures are part of the C++ 2011 Thread Library, which not only offers asynchronous access to data but also offers a barrier, so that program execution can be halted when the future is required. It is therefore possible to execute the independent paths of the call graph using standard futures. These futures can be sent to the thread pool and executed by different threads as required by the program. Program execution can continue until the data is needed by the calling thread, at which point it may block if the data is not ready.

```
for(i, i<portlist.size, i++){
  if port i is connected do
     for(j, j<portlist.size-i, j++){
          if  i doesn't equal j and j is connected
              and i's and j's connected module are not the same.

     do {
          add i to thread pool
          break;
        }
     }//return to j

     inputData i = getInputData //sequential
     bool checked i = true
}//return to i

If futures list isn't empty{
for(i, i<futuresList.size, i++){
    if port i is still connected// handle graph change
    inputData = get future from i
}}

for all other ports InputData = get inputData from Main;
```

Figure 10: Pseudo code for the Call Graph Parallelization

Initially any independent path that was found was sent to the thread pool as a future. This fine grained solution performs well when there are a only a few modules used in a patch, meaning there are only a few futures being created for each callback. As the size of a patch is increased then the program suffers from slow execution times and ultimately starts to fail when more demands are put on the synthesizers engine. When referring to the pseudo code in Figure 10, it can be seen that there are several conditions that need to be met and several checks, then rechecks of connections before the future can be sent to the thread pool and then returned to the main thread. The rechecks of connections must be done due to the real-time nature of the AMS patch manipulation, which allows for connections to be changed *on the fly* as the program is executing. Any change in the graph must be accounted for. If a path in the call graph, that was running as a standard future is disconnected, this change must not affect the execution of the AMS. If there is a call for a future that has been disconnected, the AMS will crash following a system error. The changes in the graph, however, are easily handled by rechecking the connections of a module before calling the future, if the module's connection has changed then the return value of the future is simply not requested.

The overhead of the call graph parallelization is expensive, as is reflected in the execution times during the initial tests, if all the modules in a patch are to be searched for independent paths. The reason for this are two fold: 1) as the number of modules is increased in a patch there is an increase in the number of calls to the function to find independent paths, adding extra overheads, and 2) each time an independent path is found then the number of futures created is increased adding a second overhead to the system.

The initial solution to the added overhead is to increase the number of threads in the thread pool. Dividing the work amongst a number of threads decreases the amount of futures handled by each thread. This leaves only the overhead of the search for independent paths to be run by the main thread. This solution didn't offer any gain in speedup but did increase throughput, and hence performed better than the sequential version. As more threads are added to the thread pool, the better the stability of the engine and respectively the throughput. The complexity of the patches could be increased to 56 modules using 4 threads in the thread pool on the quad core system. Any attempt to run more than four threads in the thread pool failed due to the limitations of hyper-threading [26]. Although the CPU uses hyper-threading, claiming eight virtual cores, there are really only 4 cores. It was found that about 75% of the CPU resources were utilized with 4 threads of the quad core system, and this percentage increased drastically after more threads were used, causing distortion in the signal path more than likely due to context switching with more than one thread running on a single core.

Using four threads for a single soft synth, however, seems to be a poor use of resources in a commodity system, noting that the AMS, generally speaking, will not be the only instrument running on the system as a compositional tool, but would be one of several audio applications running at the same time (Figure 11) and ,as suggested in [2], limiting the number of threads executing real time audio applications may in fact improve performance. Because of these facts a different approach was found that offers a better solution to the added overheads of tracing the call graph and the number of futures generated during the call back routine. This is discussed in the following.

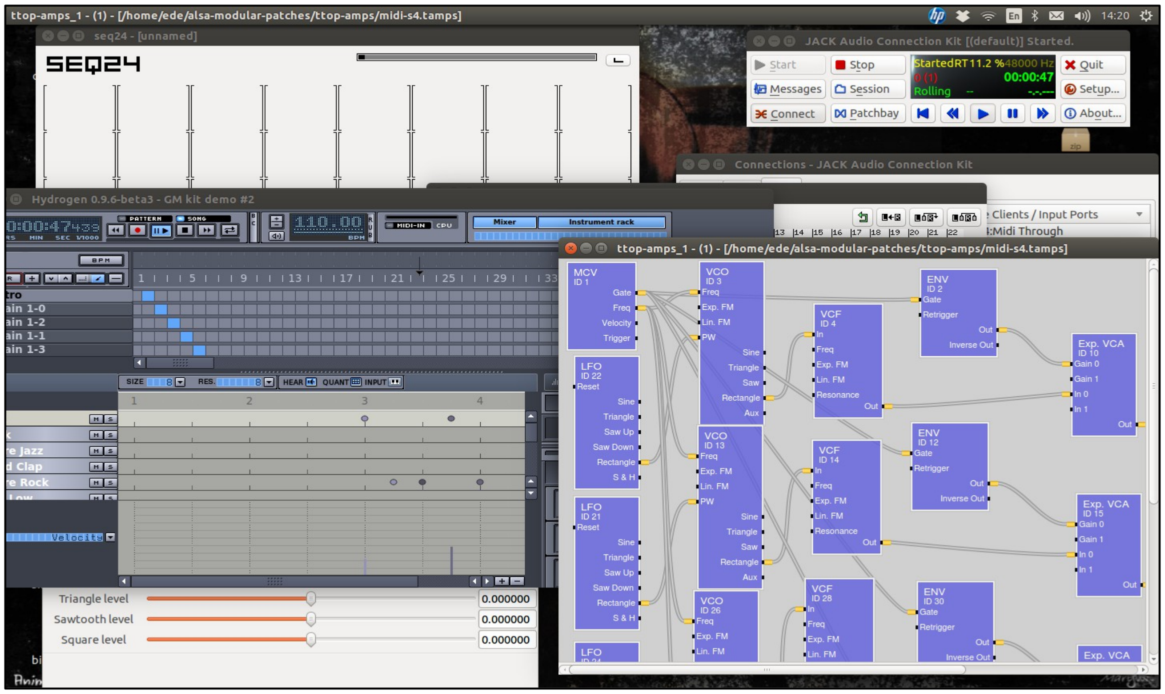

ttop-amps_1 - (1) - [/home/ede/alsa-modular-patches/ttop-amps/midi-s4.tamps]
seq24 - [unnamed]
SEQ24
JACK Audio Connection Kit [(default)] Started.
Start
Stop
Quit
Messages
Session
Setup...
Connect
Patchbay
About...
Connections - JACK Audio Connection Kit
Hydrogen 0.9.6-beta3 - GM kit demo #2
Mixer
Instrument rack
Clients / Input Ports
Midi Through
MCV
VCO
VCF
ENV
LFO
Exp. VCA
Triangle level
Sawtooth level
Square level

Figure 11: Basic Linux audio studio set up running a sequencer, a drum machine, the AMS and an effects rack.

5.2.1 Coarse Grained Parallelization of the Call Graph

When testing the AMS call graph parallelization on a single thread neither throughput nor speedup increases and the system is overwhelmed when running complex patches. This is due to the cost of finding the independent paths and number of futures generated during the callback. Experiments showed that as the complexity of the patch increases, the number of futures created should be reduced in relation to the number of modules in the patch. So when there are 46 modules in a patch there should be less futures created per module than when there are 13 modules in a patch. Table 1 shows the relationship between the number of modules, the number of possible futures, and the maximum number futures generated for a patch before the performance of the AMS is critically compromised.

Number of Modules	Number of Possible Futures	Ratio of modules to futures	Number of Generated Futures from divisor	Number of Possible Futures from Formula
7	3	1/1	3	5
13	6	1/1	6	6
23	20	1/2	12	7
35	26	1/4	9	8
45	32	1/6	8	6
56	44	1/8	7	6
66	52	1/16	5	7

Table 1: Experimental results of Parallelization of the AMS using Standard Futures.

The tests in the table were performed using one thread in the thread pool, meaning there are a total of three threads running. One thread controls the GUI. The second thread is the main thread, which initially starts the call graph parallelization and in turn handles some of the paths in the call graph (Figure 9). The third thread is the thread in the thread pool, which executes the independent paths sent to the thread pool as futures. The tests were run with number of voices set to eight, and the results are, of course, biased towards the specifics of the patches used in the tests.

As stated above the ratio of futures to modules should be decreased as the complexity of a patch increases, i.e. less futures per module for more complex synths. When deciding the number of futures that can be used to optimize a patch from the results listed in Table 1 it is clear that when the ratio of futures to modules is greater than 1/2 the AMS crashes when there are 23 modules within a patch and that as the complexity of the patch increases the ratio between the number of modules and futures decreases so that in the final patch there is a ratio of 1/16.

It was found that any ratio of modules to futures where the total of futures is less than 4 within a patch, even with less than 20 modules, makes for a very unstable synth. So a base number of futures is set at 4, allowing for an acceptable number of futures for less complex patches so that stability is not compromised. The first synth in the table has 3 possible futures, but the formula allows for 5, this gives two more possible independent paths than our test case, allowing for the possibility of extra futures if needed. The overhead of searching and then running the independent paths as futures for the more simple patches still offers better performance than either the fine-grained version or sequential version of the AMS.

As the complexity of the patches increases, however, the ratio of modules to futures needs to decrease. When increasing the divisor in blocks of 10 modules, and increasing the number of futures by 1 within each block, the AMS remains stable until the number of modules exceeds 40. So the block size for the ratio of futures to modules can be set at 1/10, until the number of modules reaches 40. After 40 the system is again over whelmed by the overheads of searching for and generating futures. Implying the number of futures per module needs to be reduced further, so a scaling factor is introduced so that after 40 modules the increase in additional futures is one future for every 20 modules. Reducing the formula we get:

$$MaxFutures = BaseFutures + \left\lfloor \frac{NumberOfModules}{BlockSize \times ScalingFactor} \right\rfloor$$

The outcome in performance gain is a lot better using this formula. Speedup improves as well as the stability of the synthesizer and accordingly the throughput. In fact, it increases to such an extent that the system no longer crashes as the complexity of the synth is increased to its maximum test case, when only running one thread in the thread pool. The paths in the call graph are then shared between the main thread, which executes some of the independent paths and the thread in the thread pool, which executes the other independent paths as futures.

The coarse-grained parallelization of the call graph satisfies the goal to use a limited number of threads, but at the same time achieves better performance than the fine-grained method, which has to use 4 threads in the thread pool to gain similar results. Limiting the number of threads used by the AMS is the desired outcome since the AMS would, more than likely, not be the only application used in the Linux audio environment to compose music, but would be used in conjunction with audio compositional tools (Figure 11).

5.2.2 Parallelization of the Call Graph Using Standard Async and Standard Detach

The C++ Standard Library, 2011, not only offers the user the flexibility of using Standard Futures and a simplified portable thread library, but also offers the user the option of allowing the system to decide how theses futures will be executed. Instead of using a thread pool explicitly the functions std::detach or std::async can be used. These functions are part of the C++ Thread Library and leave most of the details of how parallel branches will be executed under the system's control.

Implementing the call graph parallelization using either std::detach or std::async can be done without the explicit use of a user defined thread pool. This reduces the overhead of a thread pool, which may or may not be optimal, but in turn relies on the system to control the parallel execution of the program. Both std::async and the std::detach return a future so the parallelization of the call graph remains unchanged. The only difference is that the system is responsible for handling the futures and the number of threads used for the execution of the call graph, as there is now no explicit thread pool.

Initially all possible futures in the call graph were executed using the method std::detach, and like the initial experiments of the fine-grained call graph parallelization, using std::detach failed for even the most basic synths. In fact std::detach performed much worse than the sequential version, even when measured for throughput. Using the same formula that was used for the coarse-grained call graph parallelization also offered no real performance gain and any attempt to increase performance by reducing the number of possible parallel executions using std::detach failed. In fact any attempt to reduce the strain on the system when using std::detach didn't yield any positive results. The results are not that surprising since std::detach launches each independent path on an independent thread much like the OpenMP "parallel for" routine[7]. This outcome supports the supposition that thread creation during execution of the program is too expensive to warrant its use.

The other parallelization method using std::async performed much better. Again the call graph parallelization was used but this time it was the fine-grained version that performed the best. Again no user defined thread pool was used. Load balancing and the number of executing threads are left for the system to manage. Std::async runs any specified function asynchronously and may run the function "in a separate thread which may be part of a thread pool" or it may be executed by the same thread it was launched on [4].

The fine-grained std::detach method out performs the sequential version. The program does execute slower, but the gains in throughput are just as good as the best versions using a thread pool, specifically the coarse-grained call graph parallelization and the outer loop parallelization using conditions and signals. The difference between std::async and std::detach is that std::async doesn't always launch a new thread for each independent path but performs load balancing, and therefore decides how many threads are needed to execute the AMS call graph. The implementation details, as mentioned above, also suggest that std::async may execute functions using a thread pool. This load balancing and the implementation details are discussed in more detail in the experimental results. The slow execution times using std::async are more than likely because there is the added overhead from load balancing the futures, but throughput is threefold compared to the sequential version.

5.3 Combining the Outer Loop and the Call Graph Parallelization Techniques

There were several attempts made to gain an increase performance by combining the outer loop and call graph parallelization techniques described previously. Since the two techniques are independent of each other combining them would increase concurrency and therefore should improve performance.

An example of a combined algorithm would be the coarse-grained call graph parallelization, combined with the outer loop parallelization using conditions and signals. The out come of these experiments are elaborated in the next Chapter though there was no performance gain in any of the implemented combinations. This may however show that the overall performance of the parallel execution of the AMS is bound by the overhead of using thread pools and synchronization primitives and constrained by the number of CPU cores, When the number of these primitives is increased the system becomes overwhelmed regardless of the number of threads used to execute the program. This again is representative of [2], which states that the over use of synchronization primitives and using a large number of threads in a real time environments may in fact be detrimental to performance.

This Chapter focused on the different ways to parallelize the outer loop and the call graph. The problems that occurred when developing one technique lead to further developments of the same technique, attempts at optimizations and the development of new techniques. Some techniques worked and others revealed some of the pitfalls that can occur when parallelizing soft real-time audio applications. The next Chapter details how each of the parallelization techniques performed using several synthesizers that were designed to stress different aspects of the AMS engine.

CHAPTER 6 Experimental Results

All test were run using a 64 bit, Intel i7 Quad-Core Computer, implementing eight virtual cores due to hyper threading, running the Linux Kernel 2.6.35. The AMS audio and MIDI were connected to the Jack Audio Server and set to real time mode, at a sample rate of 48000 hertz, with the buffer size set to 256 bytes. This gives a latency of approximately 10.7 milliseconds, an acceptable latency for audio since the human ear cannot distinguish sounds that are less than 35ms apart [19]. This latency means the strike of a keyboard and the sound coming from the computer will appear to be simultaneous. The Jack callback is set at this latency, but the time measures used to test the experiments are not based off the Jack callback routine but the function call routine of the AMS engine, described in the proceeding chapter.

Although the Jack Audio Server was set up to run in real-time mode, and priorities were set accordingly, there was no real-time patch used to optimize the standard Linux kernel as it is generally recognized that the standard kernel is now sufficient for the requirements of audio applications [25]. There were also no more optimizations, such as running applications as root in order to further optimize execution of the code. The primary reason for this is to see the outcomes of the tests under normal circumstances using user level privilege. Using the cross platform C++ thread library also means that thread priorities are handled by the system [4] and not controlled by the programmer.

The number of threads in the experiments specifically implies the number of threads running in the thread pools. It is important to note that the sequential version of the AMS already runs two threads: one been the Main thread and another is for the GUI. This means that if the experiment says two threads then it is the count of the threads in the thread pool, which would make the overall thread count four.

There were three basic tests designed to stress different aspects of the AMS Engine. The first experiment increased the number of voices used by the Simple Synth from Figure 3. Increasing the number of voices stresses the outer loop of the routine that processes the audio cycle for each module. The more voices that are used by the synthesizer increases the number of objects representing the individual voices the thread pool must process within a given time.

The second test, the Complex Synth Test, keeps the number of voices constant but increases the number of modules. The Simple Synth with the addition of an LFO module was used for this test. Initially only one synth was used to generate the sound. Then these modules were duplicated to create more complex patches. Eventually a patch using twelve duplicated synths was tested. The throughput of this test became important to the overall results of the different parallelization techniques.

The third test used the same patches as the Complex Synth Test, but the number of voices was increased to eight. This test stressed both the outer loop and the main engine as the number of modules is increased. The final test put strain on the throughput of the engine a lot more than the initial two tests as strain is placed on the overall performance of the AMS because the test forces the AMS to process eight voices while at the same time processing numerous modules.

Each experiment was intended to bias one of the parallelization techniques developed for the AMS. The first experiment was intended to bias the Outer Loop Parallelization, the second the Call Graph Parallelization, and the third was intended to test the abilities of both methods to handle stresses to both aspects of the engine.

Although there were only three tests developed to stress different aspects of the AMS, there were numerous solutions derived from each possible parallelization, specifically to see performance gains using different parallelization techniques and to synchronize parallel executions where data races were occurring.

The time recorded was done using the Linux *gettime* algorithm. Only the time taken to execute the callback routine of the AMS is measured, no other routine is timed. For example getting midi input data, which may be essential to the execution of the synth but is handled in the same manner by the all the engines. There are several instances where the program crashes. The Jack Audio Server "zombifies" (terminates) the client thread because the Jack callback waits too long for a return from the AMS and therefore calls a timeout on the connection. This is the result of over taxing the system. The AMS doesn't have enough time to process the necessary audio data, and is unable to meet the real-time requirements of the Jack callback.

The experiments described in the following use the parallelization solutions described in Chapters 4 and 5. For a better understanding of the results below, it is important to refer to these Chapters to get a more thorough description of the methods used in connection to the experimental results.

6.1 Simple Synth Test

The AMS is an emulation of an analogue monophonic synthesizer and is designed to play only one voice at a time [3]. The user, however, can change the default monophonic behavior in order to use more than one voice. For the most part, polyphony is an expected attribute of a modern synthesizer. The number of voices used in a patch is left up to the user's discretion but, as will be shown in the following, the more voices the AMS processes the more strain is put on the engine. The synthesizer used for this test is the patch shown in Figure 3, the Simple Synth. The name representing the fact that by design it is a simple synthesizer in that it uses a limited number of modules and is, therefore, simple in design.

The sequential version of the AMS performs the Simple Synth Test relatively well and is able to process 64 voices while running the Simple Synth. The AMS only becomes unstable when the polyphony is increased to 128 voices at which point distortion becomes audible in the signal path. This implies, as mentioned earlier, that the synthesizer has failed to calculate the data required within the specified time of the Jack callback routine. In most cases though 64 voices is more than adequate for such a simple patch.

The initial parallelization method, the "naive" method, which uses no barriers or synchronization techniques, seems to give super linear speedup with close to 10 compared to the sequential version (Figure 12). Unfortunately, these metrics are not precise because the times recorded are inaccurate, as the execution of the voices in the thread pool are not measurable without introducing some kind of barrier in order to record the threads execution times. In other words, only the execution time of the main thread is recorded, not the execution of the individual threads in the thread pool. The introduction of an atomic counter and a barrier, which forces the main thread to wait until the threads in the thread pool have calculated the data in a patch gives a precise measure of the amount of time the naive method actually takes. This *timed* version gives a more reasonable result but does give an average speedup of 1.6 when using one thread in the thread pool.

The naive version unfortunately fails, whether timed or not, when more than one thread is running in the thread pool as the "naive" method doesn't address the data race occurring between modules (Chapter 4 and 5). The solution to this data race is to introduce an *atomic counter and a barrier* which forces the main thread to wait for the threads in the thread pool to calculate all the voices of one module before adding the next modules voices to the pool. This method works well but with no additional speedup (Figure 12) when compared to the sequential version, but is able to process up to 128 voices, achieving greater throughput than the sequential version, which is only able to process 64 voices.

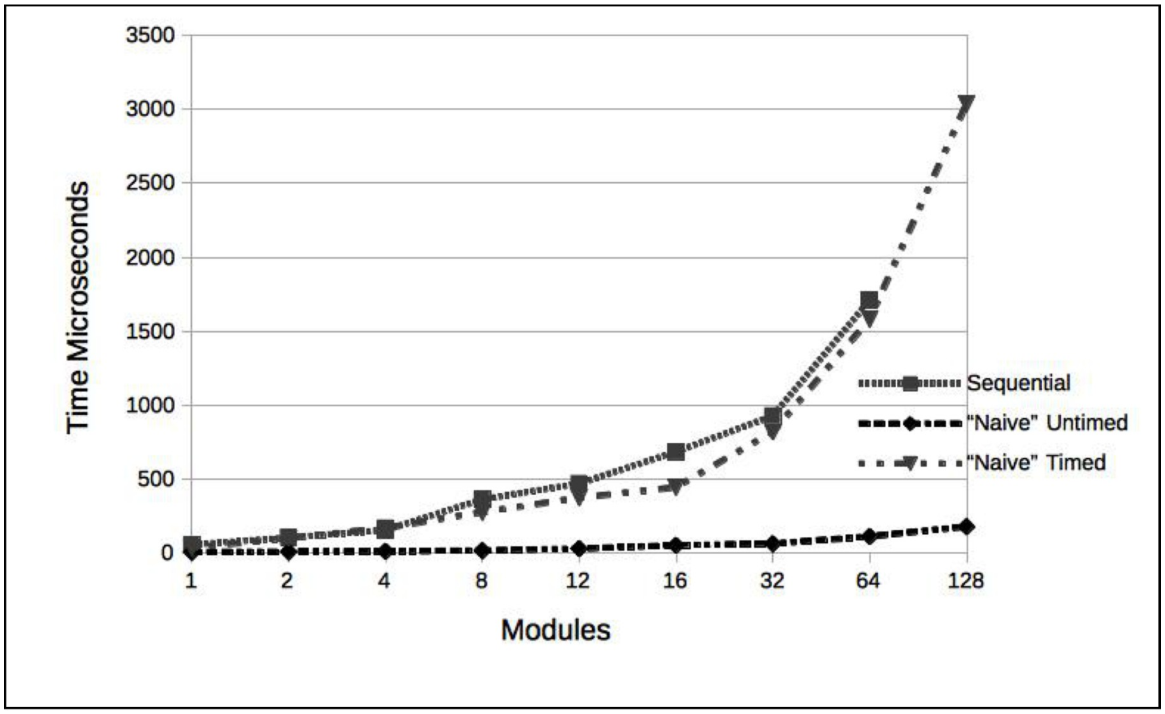


Figure 12:Simple Synth Test. Sequential Version and "Naive" Parallel Versions time and un-timed.

Another solution to the data race occurring between the main thread and the threads in the thread pools is to add a Boolean return to the method *generateCycle* for each voice, and send this method to the thread pool as a standard future. Although this solution solves the data race between modules the resulting execution times are much slower than the sequential version and also much less stable. The signal becomes distorted running only 64 voices, whereas the sequential version distorts at 128 (Figure 13). The reasoning for these results is more than likely due to the cost of generating the number of futures needed to synchronize the voices and modules. If there are six modules used in the simple synth and if each module calculates 32 voices, it means in total there are 32 times 6, 192 futures created per cycle. This fine-grained solution obviously puts a large strain on the system.

An attempt to rectify this was to send blocks of voices to the thread pool as a single, coarse-grained future. This solves the problem of overloading the system with futures by decreasing the number of generated futures, but adds work to each individual thread, which now calculates more voices. This solution offers some improvement when compared to the fine-grained solution: it is faster and is able to calculate up to 128 voices without any audible distortion. It also offers some improvement compared to the sequential version as it is able to execute up to 128 voices, but the execution times below 128 voices are almost identical to the sequential version (Figure 13).

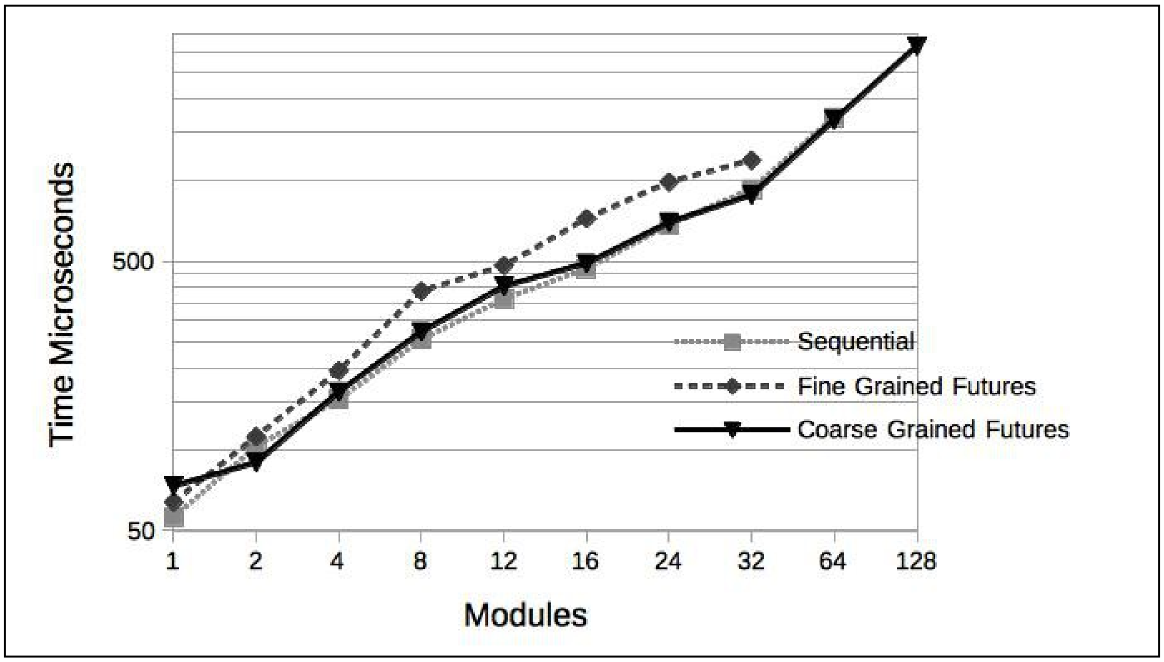


Figure 13:Simple Synth Test. Sequential and Outer Loop Parallelization using Standard Futures, Fine and Coarse Grained

.

The final solution to parallelize the outer loop using conditions and signals, which force the main thread to wait for all the voices of each module to complete before sending the next module's voices to the thread pool, shows good performance results when there are less voices used, but as the number of voices is increased, the performance becomes worse than the sequential version. Again, however, this parallel version is also able to calculate 128 voices (Figure 14).

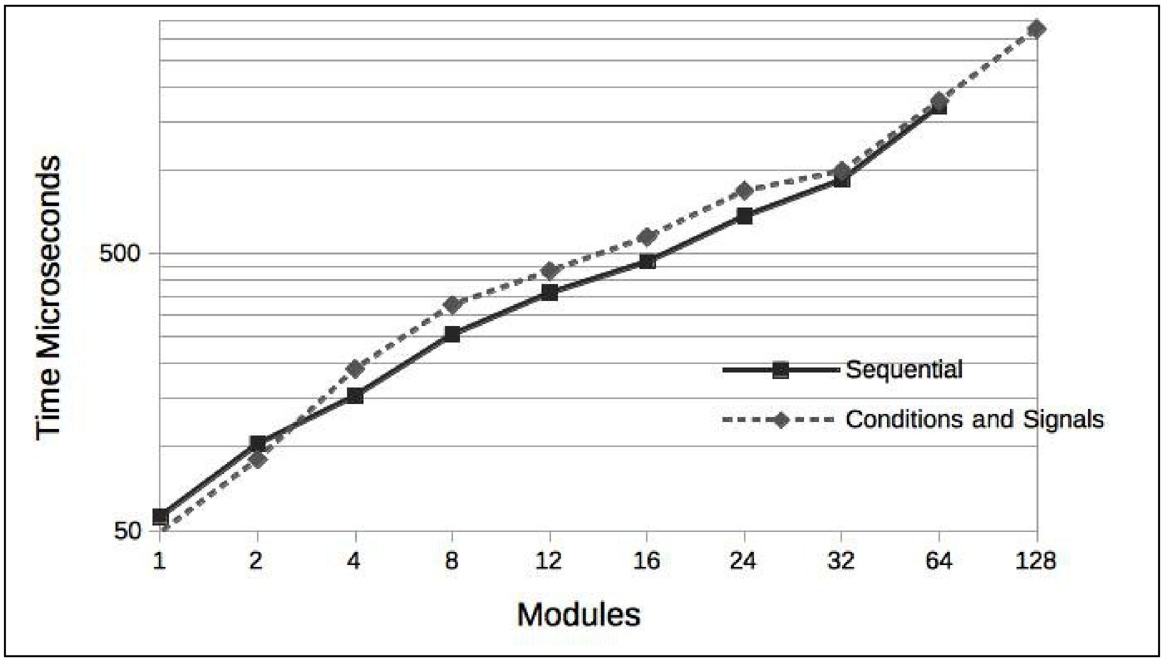


Figure 14: Simple Synth Test. Sequential and Outer Loop Parallelization using Conditions and Signals

The second solution proposed to parallelize the AMS engine is to execute the modules that can be executed independently by tracing the execution of the call graph (Chapter 4 and 5). The results of this method show a speedup of close to 1.5 for less than 8 voices. When running 2 voices, the sequential version runs at 102.5 microseconds, and the parallel version runs at 76.4 microseconds. The performance results, shown in Figure 15, show that performance worsens as the number of voices is increased past 32 voices, and then the parallel version performs similarly to the sequential version. When examining the Simple Synth, this is perhaps the expected result because there are only two independent paths in the call graph. So eventually the time to execute the voices supersedes the parallel. Although the parallel version still executes faster than the sequential version, the speedup is much less as each thread has to calculate a large number of voices. The call graph parallelization is also able to generate 128 voices without distortion. Again there is no significant change in execution time when using more than one thread in the thread pool.

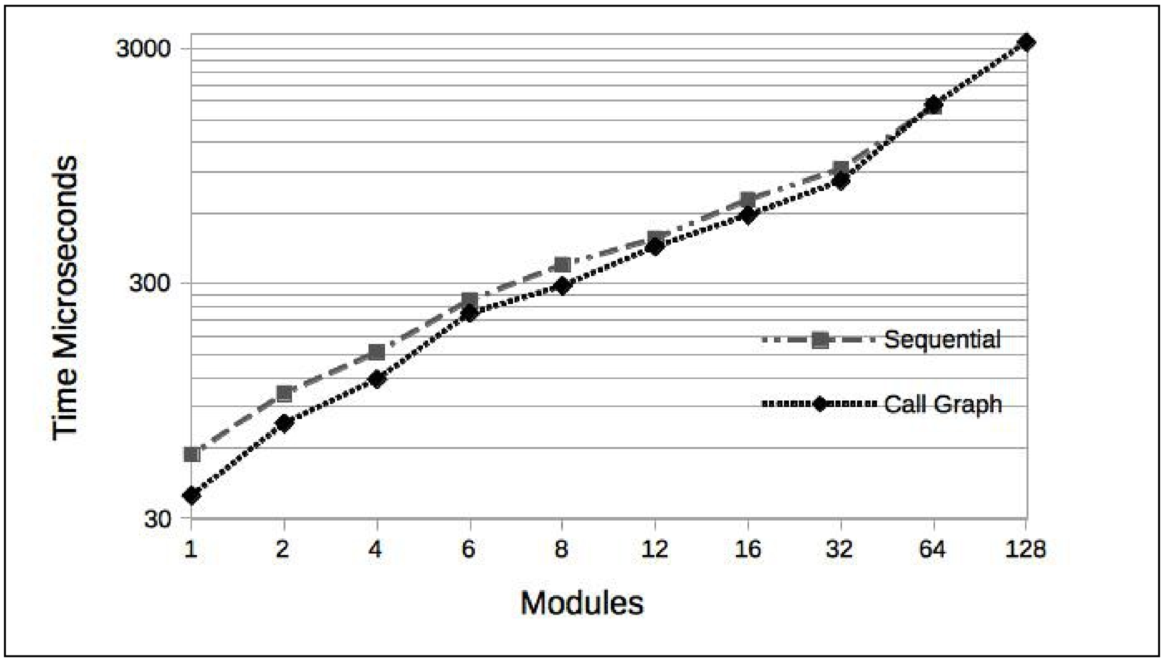


Figure 15: Simple Synth Test. Sequential and Parallelization of the Call Graph using Standard Futures.

6.2 The Complex Synth Test

The second test puts stress on a different aspect of the engine, specifically the number of modules the AMS can process in the callback time. The “Complex Synth” is initially the same synthesizer as the Simple Synth (Figure 3), but with the introduction of an LFO (Low Frequency Oscillator). The Complex Synth Test involves duplicating the modules of the previous patch, and each time testing this new more complex synth for performance. The initial patch is duplicated so there are in a sense two synths in the new patch consisting of 13 modules, then four synths consisting of 23 modules, and so on, until the maximum test case is reached, which consists of a patch consisting of 66 modules. Therefor each time the modules are duplicated creating a more complex synthesizer. Complex because the number of modules is increased and therefore the complexity of the call graph. Figure 16 is a screen shot of the Complex Synth, built using 23 modules, i.e. four duplicated synths. The number of voices is kept static, each patch playing only one voice so the strain is no longer on the processing of voices but focused on the overall performance of the engine. The sequential version performs this test well, and is able to run the maximum number of modules tested.

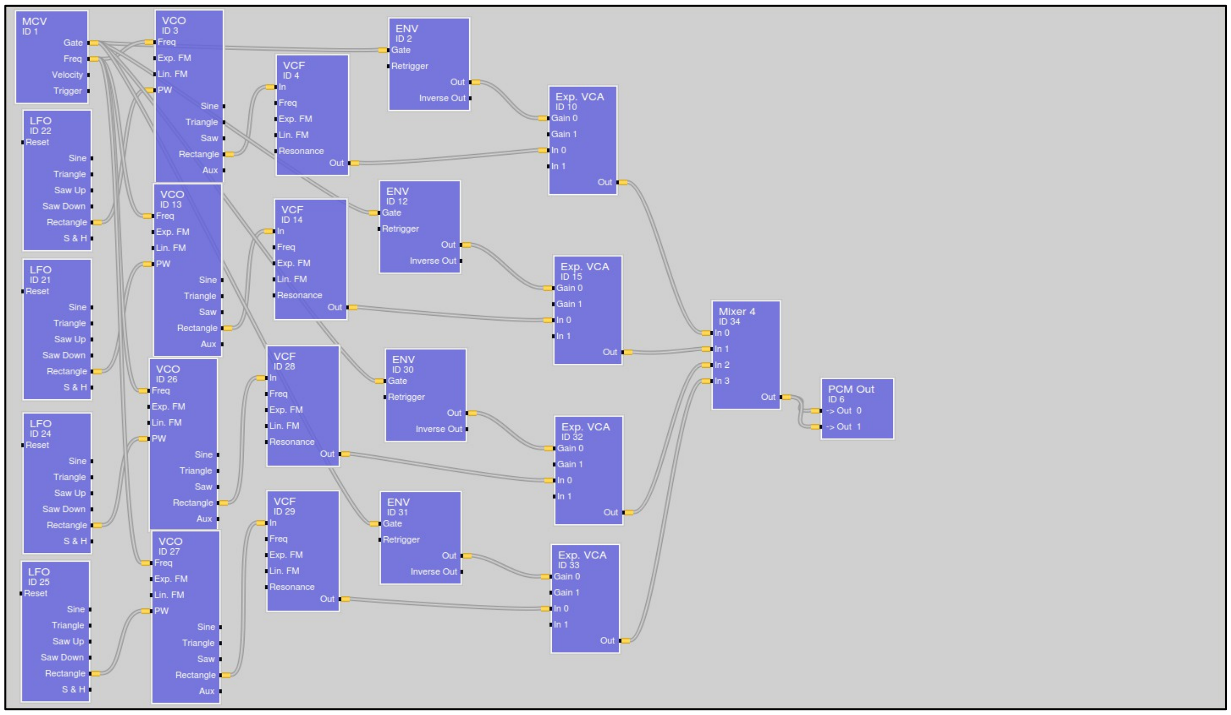


Figure 16: Complex Synth, using 23 Modules

Processing the outer loop parallelization using the timed “naïve” version runs at about the same speed as the sequential version. This, however, is not surprising as there is only one voice per module been processed in the thread pool and much of the work involves the main thread and not the threads in the thread pool, which are only executing one voice per module.

The outer loop parallelization using conditions and signals gives similar results as the sequential version for the less complex synths but performs much worse as the complexity of the patch increases. This is not an unexpected result as the overhead of synchronization is also increased as more modules are added to the patches because each module has a barrier in order to avoid the outer loop parallelization’s data race. This does give a good indication of the cost of using such synchronization primitives, which as shown in Figure 17, add an approximate overhead of about 30% for more complex patches.

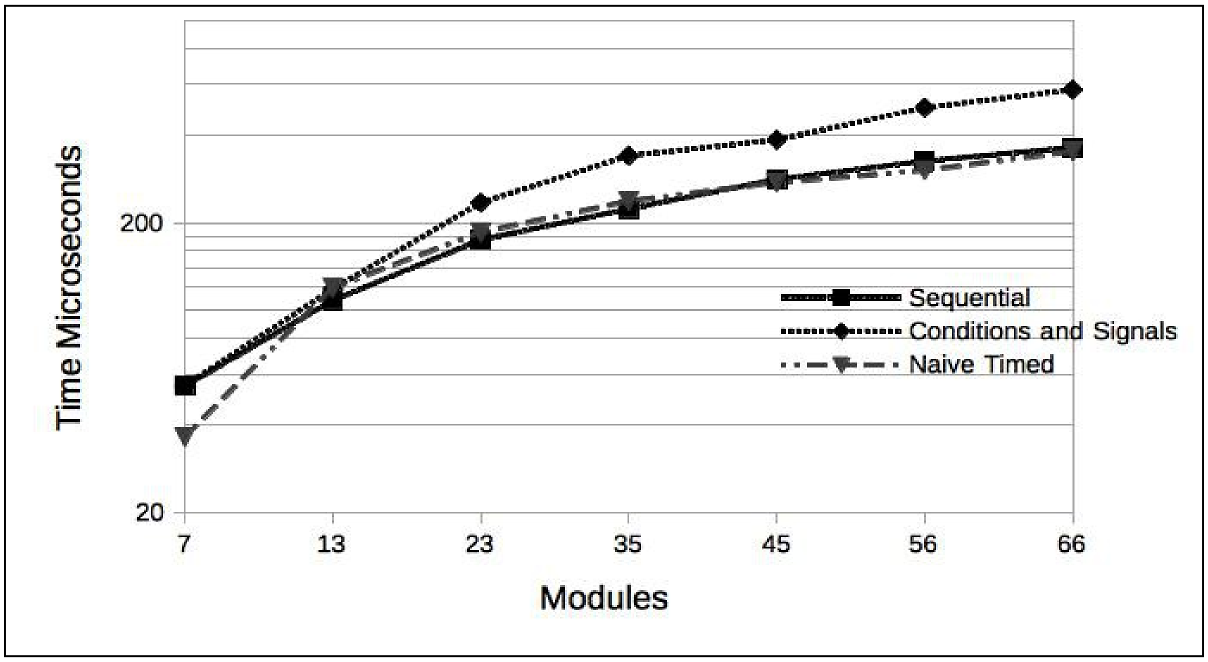


Figure 17: Complex Synth Test, One Voice. Sequential and the outer loop parallelization, Naive Timed and Conditions and Signals.

The next test, the fine-grained outer loop parallelization using standard futures, which didn’t give positive results for the Simple Synth Test, again gave poor results for the Complex Synth Test. Figure 18 shows the results for this parallelization technique. The sequential version is able to run 66 modules whereas the fine-grained outer loop parallelization fails at 45, showing just how much the creation of futures compromises performance. The coarse-grained solution was not tested as there is only one voice being used for the initial Complex Synth Test, therefore each future represents a single voice, so the granularity would remain the same for both techniques.

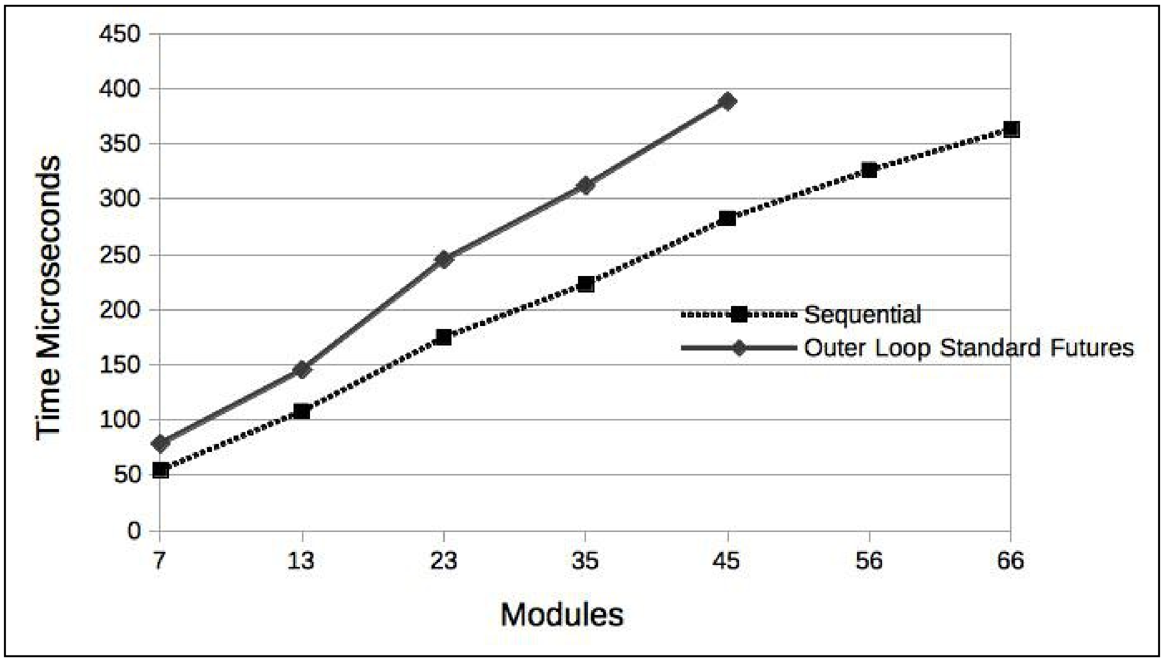


Figure 18: Complex Synth Test, One Voice. Outer Loop using Standard Futures (Fine-Grained)

The final test using standard futures to parallelize the call graph should give the best results, as the parallelization of independent paths in the call graph should take the strain off the main thread and balance the parallel execution of the graph accordingly. This is not the case and in fact returns worse results than the sequential version. The execution of call graph parallelization fails when there are more than 23 modules running in a patch when only one thread is running in the thread pool.

Running two threads in the thread pool gives better results, and performs similarly to the sequential version, but fails when tested to run more than 56 modules. The speedup running four threads is the same as for two threads, but throughput increases and the four threads are able to run the maximum number of modules tested, 64 (Figure 19). These results were disappointing but did stress the overhead introduced into the system when using standard futures to parallelize the engine.

The call graph parallelization can run more complex synths when two or more threads are running in the thread pool but still offers no improvement over the sequential version (Figure 19). As the number of modules in a patch increases the performance worsens due to: 1) the amount of time checking for independent paths in the call graph and 2) the increase in futures generated from independent execution paths found (Chapter 5).

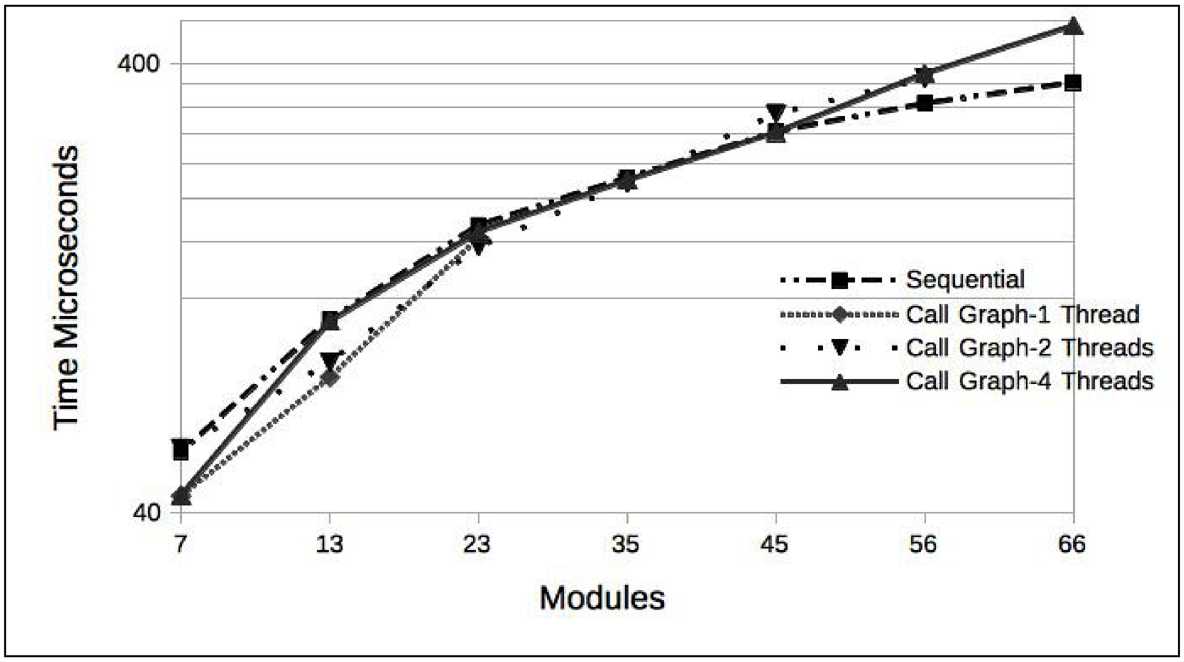


Figure 19: Complex Synth Test, One Voice. Sequential Version and Parallelization of the Call Graph using Standard Futures, Fine Grained.

The overhead introduced into the system when testing the Complex Synth using standard futures was resolved by using less futures in the call graph based on the formula described in Chapter 5. This coarse-grained parallelization reduces the overhead in two ways, there are less futures created for each callback and also less testing for independent execution paths. In Figure 20, it is clear speedup improves in comparison to the sequential version. The stability of the synthesizer, and accordingly the throughput, increases when optimally running only one thread in the thread pool. In fact, throughput increases to such an extent that the system no longer crashes when tested with the maximum number of modules. In comparison, the fine-grained version fails at 23.

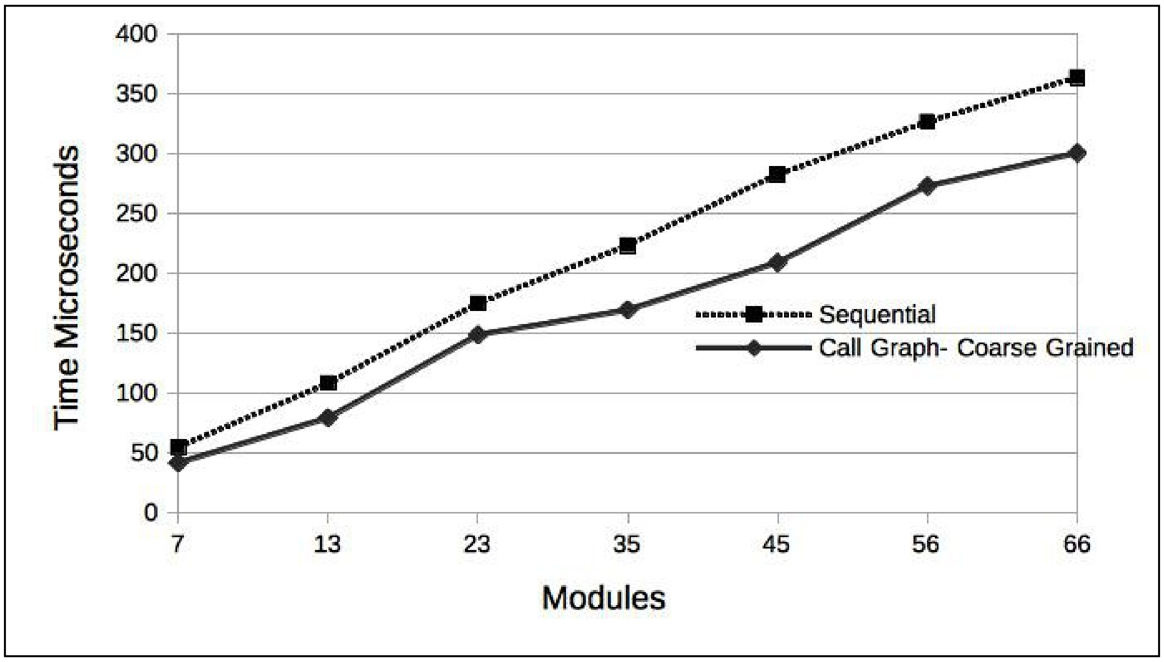


Figure 20: Complex Synth Test, One Voice. Sequential Version and Parallelization of the Call Graph- Coarse Grained

6.3 Complex Synth- Eight Voice Polyphony

The third test is designed to stress both the outer loop and the call graph parallelization of the AMS. Again, the Complex Synth is used and the complexity is increased as before through the duplication of modules. For this test, however, the number of voices is set to eight. This puts strain on both the outer loop and the call graph parallelization at the same time, because as the complexity of the synth is increased the parallel paths in the call graph increases and eight voices must be processed in the outer loop.

The sequential version performs perhaps worse than expected and after the number of modules is increased past 23, the signal becomes distorted, the demands on the engine overwhelm the system and ultimately, the application crashes. Because the sequential version crashes running only a few modules, it is difficult to assess speedup. Overall performance gains for the second Complex Synth Test are then more related to stability and throughput.

The parallelization technique using conditions and signals gives the best performance gains for the outer loop parallelization. The execution time for the less complex synths was actually slower than the sequential version, but unlike the sequential version, this parallel version was able to perform well even when the number of modules was increased over 60, increasing throughput approximately three times that of the sequential version (Figure 21).

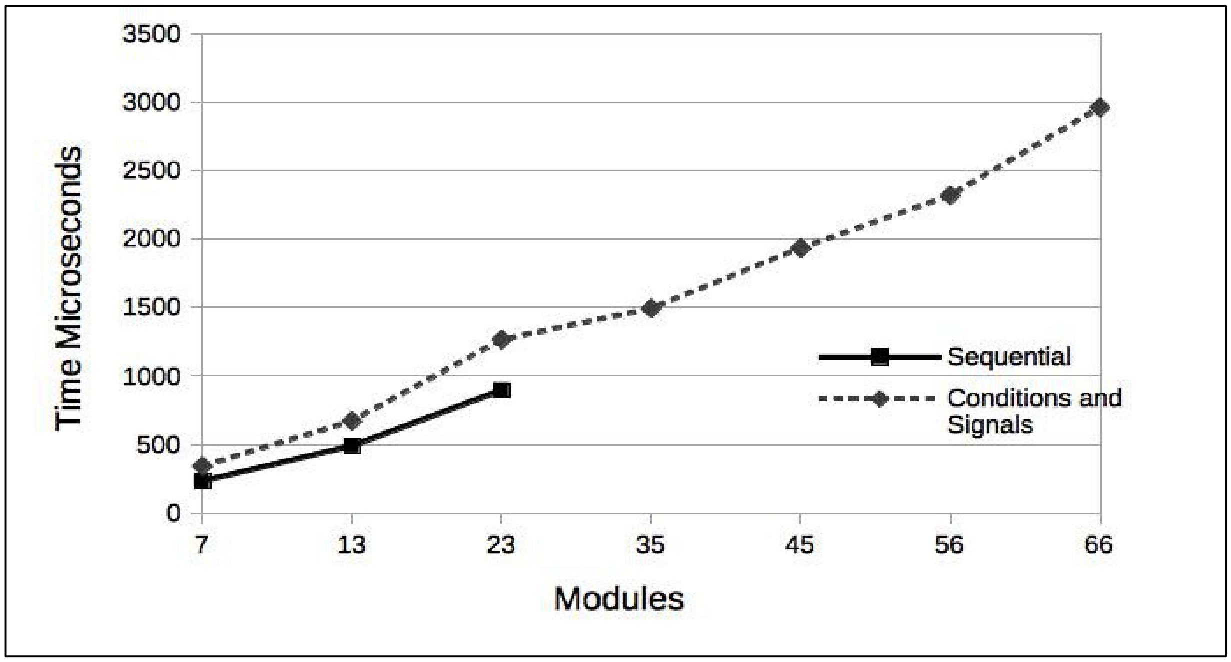


Figure 21: Complex Synth Test, Eight Voices. Sequential Version and Parallelization of the Outer Loop using Conditions and Signals. Note the sequential version fails after reaching 23 voices.

The next series of tests were to see if there could be any real gain when using standard futures to parallelize the outer loop. The fine-grained version was not tested, as it didn't show any performance gains in the first two experiments. So only the coarse-grained implementation was tested. The block size was set at 4, meaning that there are 4 voices bound to the future that is sent to the thread pool. Again this solution performed poorly. In fact, of all the attempts to parallelize the AMS using this method, whether fine-grained or coarse-grained, it gave the worse results.

The tests were also repeated for the parallelization of the call graph using standard futures, fine-grained. This technique, again, produced disastrous results for a thread count of one. The system crashed after the complexity of the patch was increased to 23 modules. These results are the same as the first Complex Synth Test, showing that the stress on the system is from the overhead of defining independent paths and creating futures and perhaps not the processing of voices.

As more threads were introduced into the thread pool, the complexity of the patches could be increased to 56 modules when using 4 threads (Figure 22). The performance gains, however, may be outweighed by the increased demand put on the system when running the AMS using four threads from the computer's resources, as explained in section 6.2. This however may become trivial if the number of processors on a CPU chip increases beyond the 8 hyper-threaded cores that are available on the test CPU.

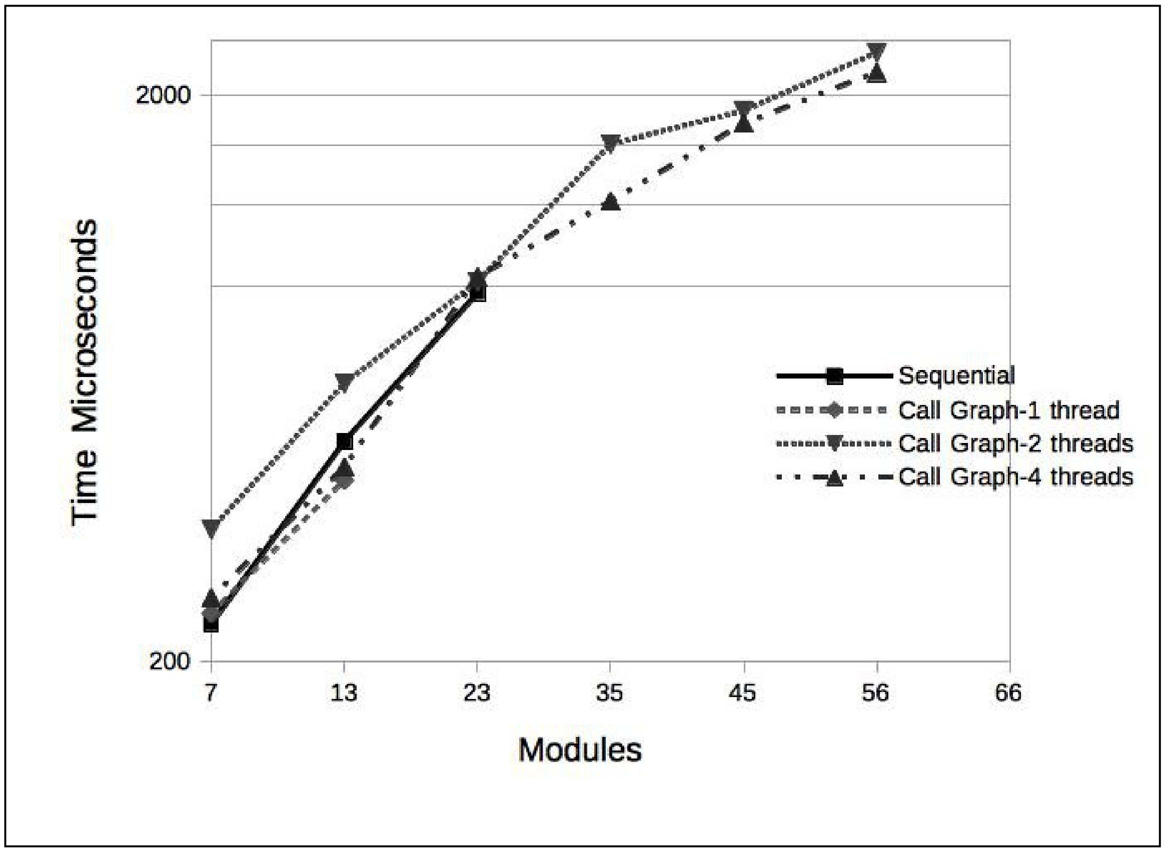


Figure 22: Complex Synth Test, Eight Voices. Sequential Version and Parallelization of the Call Graph using Standard Futures Fine Grained

Finally, tests were run for the coarse-grained call graph parallelization using futures to parallelize the independent modules. The use of the coarse-grained parallelization gives good performance results when executing the more complex synthesizers, since the system no longer crashes before reaching the maximum modules tested. There is no gain in performance when using more than one thread in the thread pool and, in fact, as the complexity of the patches increases, the results are worse, perhaps due to competing threads trying to access the queue (Figure 23).

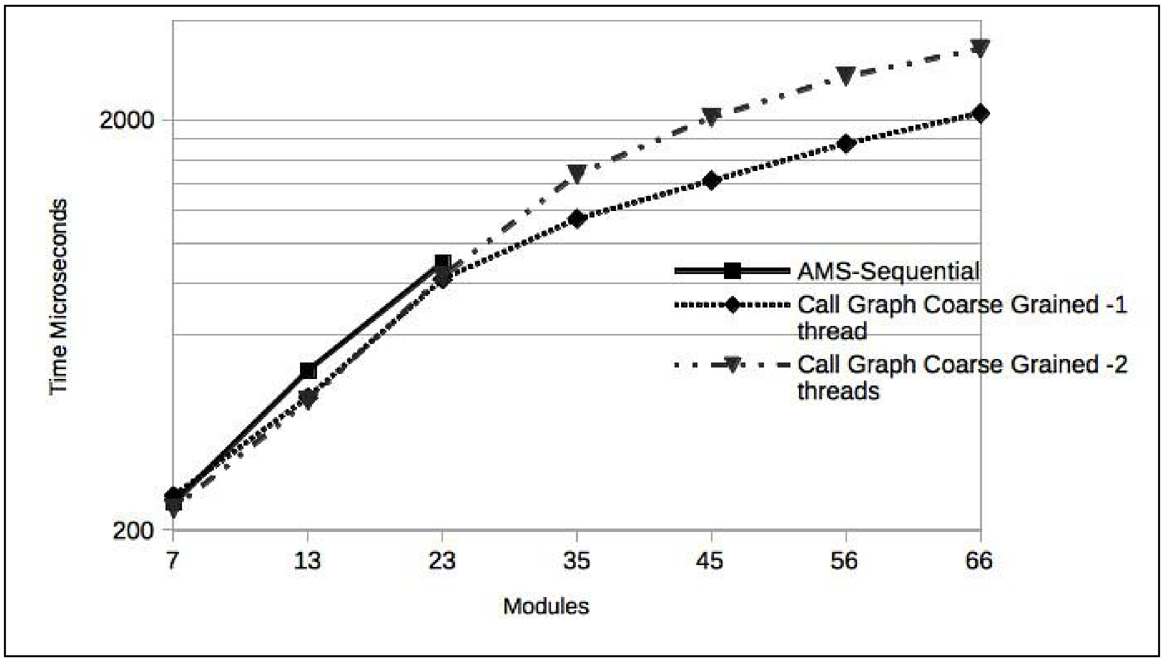


Figure 23: Complex Synth Test, Eight Voices. Sequential and Parallelization of the Call Graph, Coarse Grained

The results of the final test, using eight voices while testing the Complex Synth, revealed an important aspect of parallelizing the AMS. That is the increase in throughput and the stability of the engine when executing complex patches. Although the "naive" version performs well, it does so with no guarantee of correctness. The outer loop parallelization using conditions and signals offers no significant speedup, but gives a better performance in relation to throughput when compared to the sequential version. In fact, this parallel version can run almost three times as many modules as the sequential version. The same can be said for the throughput of parallelization of the call graph, coarse-grained, which again outperforms the sequential version three-fold.

In this respect, the important outcome of these experiments is in relation to throughput and the improved stability of several of the parallel techniques. The maximum throughput can be found by multiplying the maximum number of modules the AMS was able to run, the number of voices played and the cycle size, all divided by the latency of the system, the Jack Callback time. The reasoning for using the Jack Callback time and not the execution time of the specific experiment is that this would give misleading results. Primarily because the sequential version fails after 23 modules, which means it has a lower execution time compared to the parallel versions, but obviously its throughput is much lower. The above explanation leads to:

$$Throughput = \frac{NumberOf\ Modules\ x\ NumberOf\ Voices\ x\ CycleSize}{JackCallBack}$$

Looking at the results in Figure 25, some of the parallel techniques outperform the sequential version with regard to throughput almost three fold. The increase in throughput offers the user perhaps not a faster execution time compared to the sequential version but a more stable and scalable system for more complex synthesis routines and it is perhaps the increase in throughput that warrants the parallelization of the AMS engine. The fact that more voices and more complex synthesizers can be run when using the parallel versions that give maximum throughput suggests that, although speedup is limited, parallelization allows a for higher degree of flexibility because a greater number of voices and modules can be used in a patch.

Although there is some speedup from some of the experiments, specifically from the call graph parallelization using standard futures, coarse-grained, the results are less than expected. Throughput is increased from using some of these parallelization techniques but the overhead of using a thread pool, numerous standard futures and other synchronization techniques could be the reason for the limited performance gains. The final two tests to parallelize the AMS using the call graph implement two methods from the C++ Thread Library, std::detach and std::async.

The performance results, shown in Figure 24, of std::detach are perhaps less than expected. The program performs poorly for the Simple Synth Test. The AMS crashes when running 35 modules for the Complex Synth Test with number of voices set to one and can’t even process 13 modules when the Complex Synth is set to eight voices. Tested again using the same formula as the call graph coarse-grained parallelization, std::detach fares no better and there is no gain in performance. The final test, which limits the number of futures even less, starting with only one future created regardless of the complexity of the patch, fails the Complex Synth Test when instantiating only one and then two parallel paths and has poor results for the Simple Synth Test. As mentioned in Chapter 5, std::detach creates a new thread for each parallel path. Even when limiting the number of executed paths, these results show how expensive thread creation can be and without a re-use of threads, parallelizing the AMS gives disastrous results.

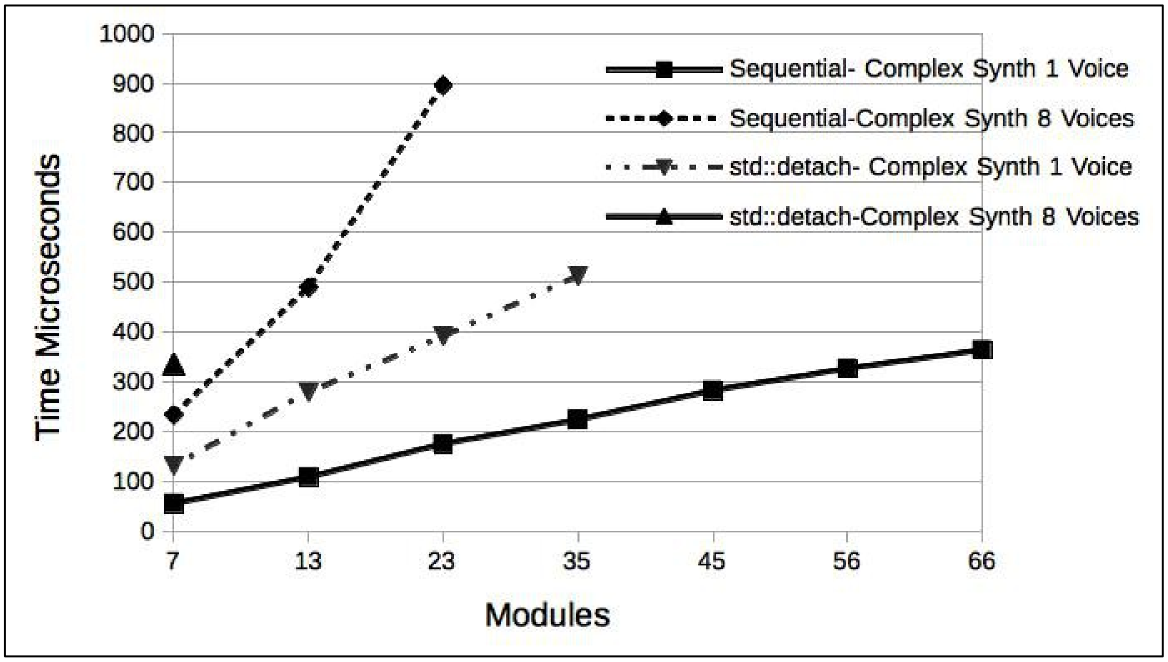


Figure 24: Complex Synth Test, One and Eight Voices. Sequential and Call Graph Parallelization using std::detach

Conversely, the implementation using std::async performs well for all tests when there is no limit put on the number of asynchronous functions generated. In fact, it is just as stable as the call graph coarse-grained parallelization. Although the std::async is slower than the sequential version (Figure 26), it does give the optimal throughput (Figure 25). Any attempt to optimize std::async by limiting the number of executed parallel paths, specifically applying the coarse-grained formula used for the call graph parallelization failed to yield better results and leaving the system to load balance the calls to std::async gave the best results.

Interestingly, the system only implements one extra thread for the std::async routine, making the number of running threads three, including the thread for the GUI. Initially, the system loads 2 extra threads, but as the program runs the system, seeing that the second thread is not needed, it suspends the thread and continues execution using only one extra thread, making a total of two threads calculating audio. The two threads are run load balanced, giving equal CPU percentages. For the most complex patch CPU usage peaks at a little over 50% for each CPU.
This is a lot like the implementations using the thread pool optimally running only one extra thread to get the best performance results.

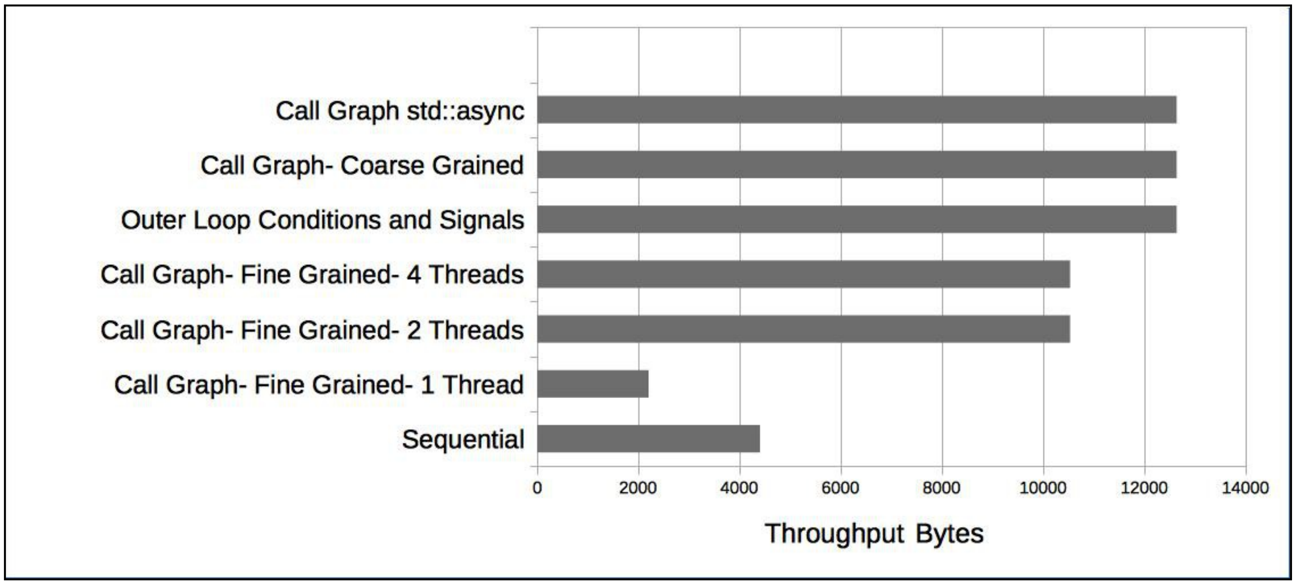


Figure 25: Maximum Throughput for Sequential and best performing Parallel Versions for the Complex Synth test running Eight Voices. The three parallel versions listed all have maximum throughput

The overall results from the experiments show that the best performance increase, for both speedup and throughput is from the parallelization of the call graph using a coarse-grained number of futures. The other technique that has good results is the outer loop parallelization using conditions and signals. Although the throughput using std::async is comparable to the other two techniques its execution time is much slower. The fact that the details of parallelization when using std::async are left up to the system means there is little possibility of further optimizations offered from this version. Conversely the std::async parallelization is much easier to implement, specifically because the implementation details are left up to the system which makes it attractive due to this simplicity.

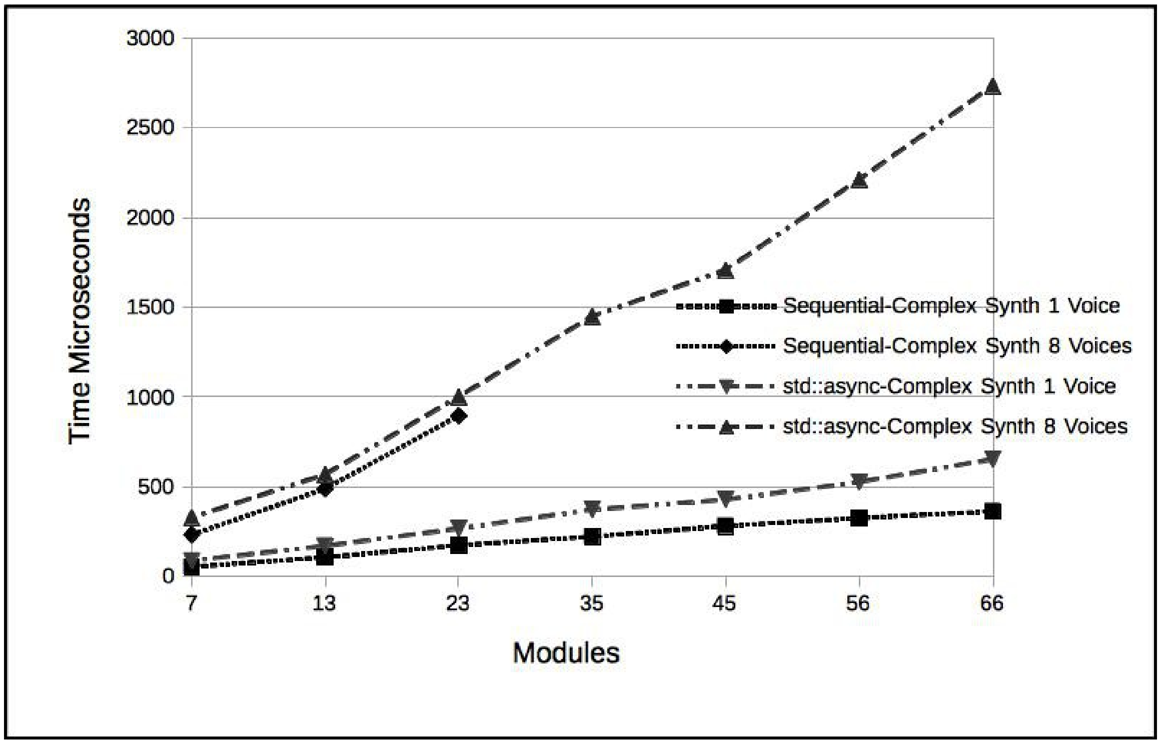


Figure 26: *Complex Synth Tests using std::async. Although the tests using the std::async library are slower than the sequential version it is clear there is an increase in throughput.*

6.4 Combing the Outer loop and Call Graph Parallelization Techniques

The reasoning for combining the outer loop and the call graph parallelization techniques is because they can be executed concurrently and, because of this concurrency, should perform well as calculations are divided between the two techniques. Unfortunately, any attempt to combine the two techniques failed. For the most part the combinations performed worse than the sequential version and none showed any gains when compared to the performance of the other parallel versions.

The initial experiments focused on the combination of the outer loop parallelization using standard futures and the call graph parallelization, fine-grained, also using standard futures. None off these techniques performed well when tested individually, but perhaps by combining them they would give better performance results. The tests were performed running numerous threads for both the outer loop and call graph but there was little performance gain by increasing the number of threads in the thread pools. There was no speedup and as the complexity of the synthesizers was increased the performance worsened. The results in Figures 27 and 28 show that the combinations don’t give better results than the sequential version for the Simple Synth Test or the Complex Synth Test.

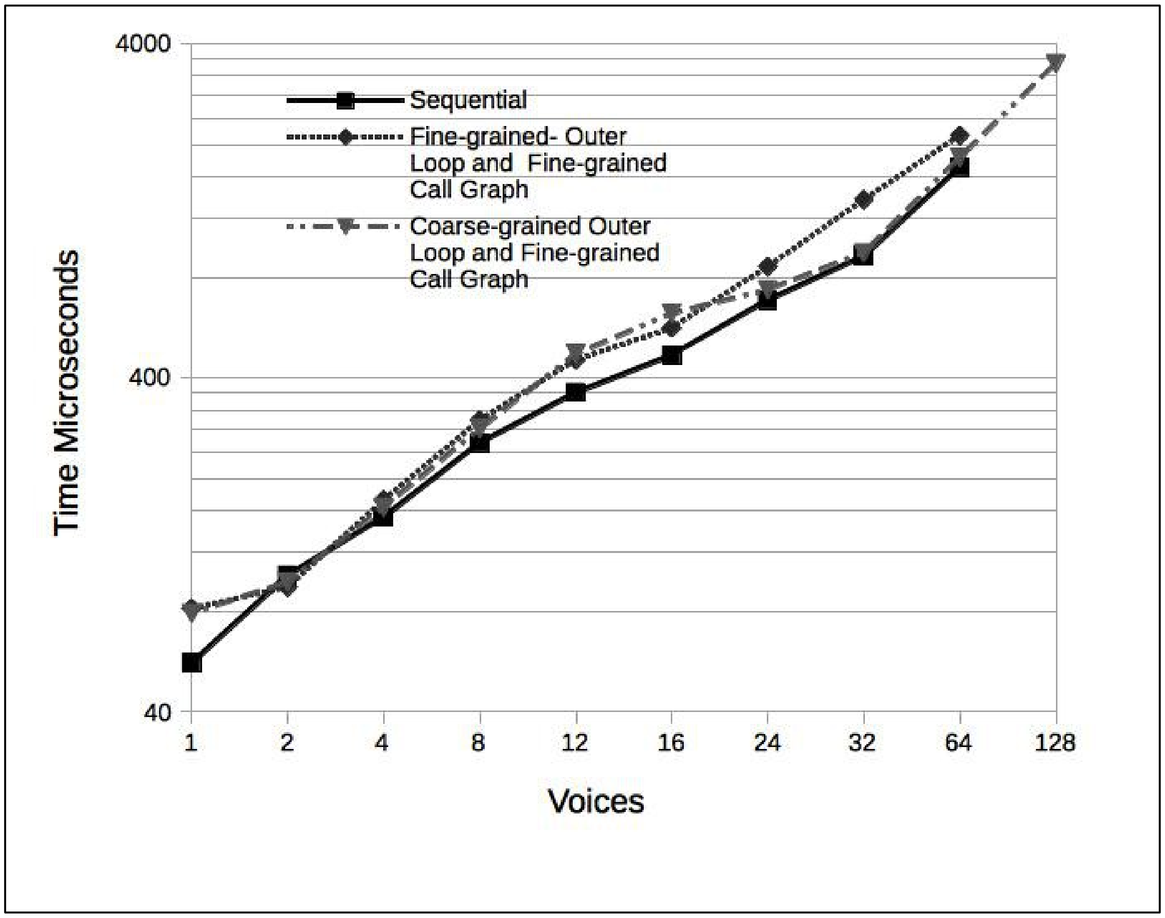


Figure 27: Simple Synth Test: Parallel Versions using a mix of the Outer Loop using Standard Futures, Fine and Coarse-grained, and the Call Graph Parallelization using Standard Futures, Fine-grained.

Although the mixed versions perform the Simple Synth Test relatively well, the same cannot be said of the performance of the Complex Synth Test where the mixed versions perform worse than the sequential version for all test cases. None of the versions could perform beyond 13 modules when running eight voices, even with more than one thread in each thread pool (Figure 28), showing just how much overhead is added to the system when parallelizing the AMS.

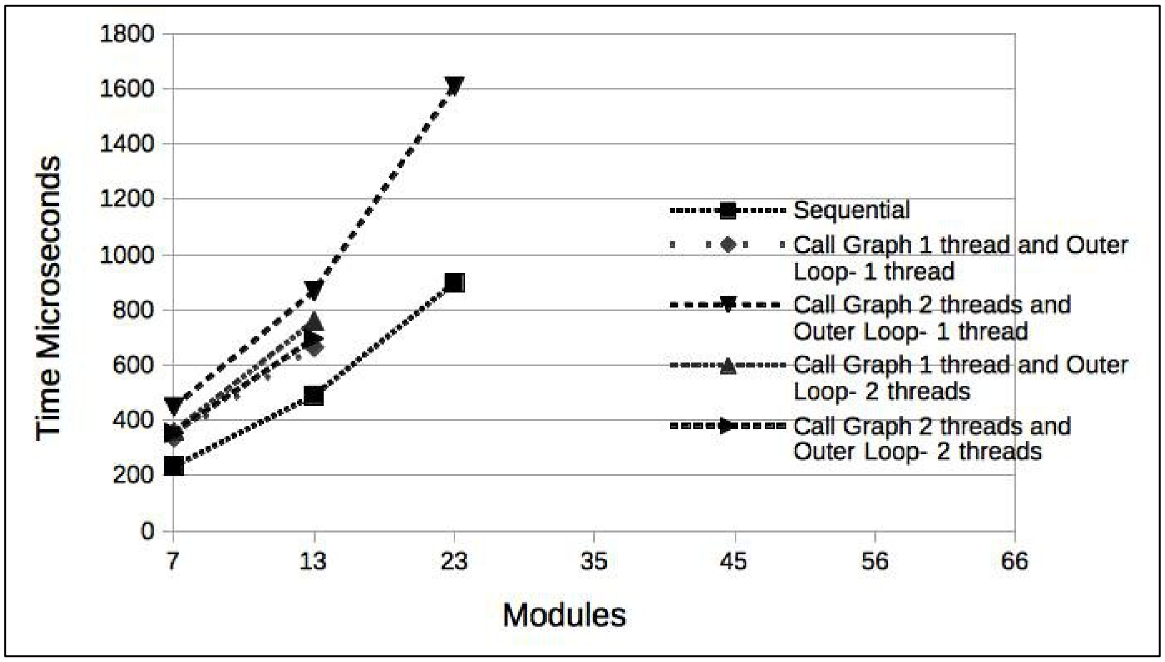


Figure 28: Complex Synth Test, Eight Voices. Parallel Versions using a mix of the Outer Loop using Standard Futures and the Call Graph using Standard Futures, Fine-grained.

The second set of experiments consisted of the outer loop parallelization implemented using conditions and signals combined with the call graph parallelization using either std::async or standard futures, coarse-grained. In the individual experiments these techniques performed well and should give good results when combined.

Again for the Simple Synth, the results are slower than the sequential version but the combination of the outer loop using conditions and signals and, the call graph coarse-grained is able to run 128 voices, increasing throughput but with no performance gain. However, it is when when testing the Complex Synth that these combinations fail, with worse results than the other mixed experiments mentioned earlier.

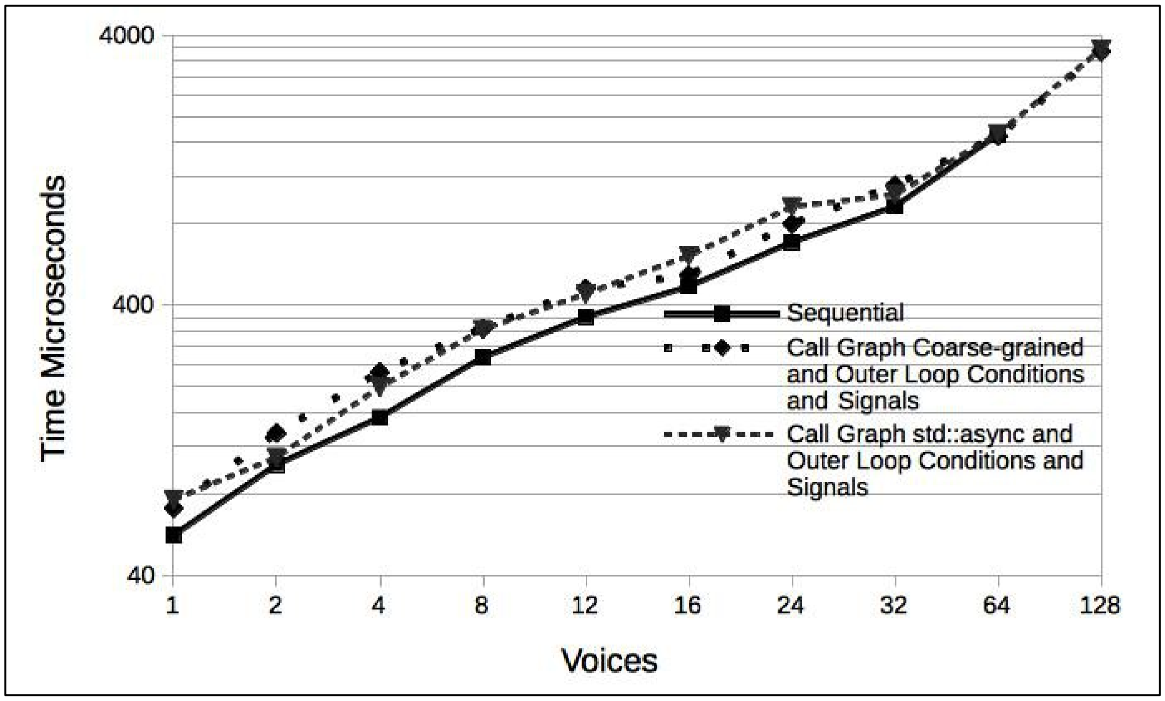


Figure 29: Simple Synth Test. Mixed Version Outer Loop, Conditions and Signals and Call Graph Standard Futures Coarse-grained and std::asysnc.

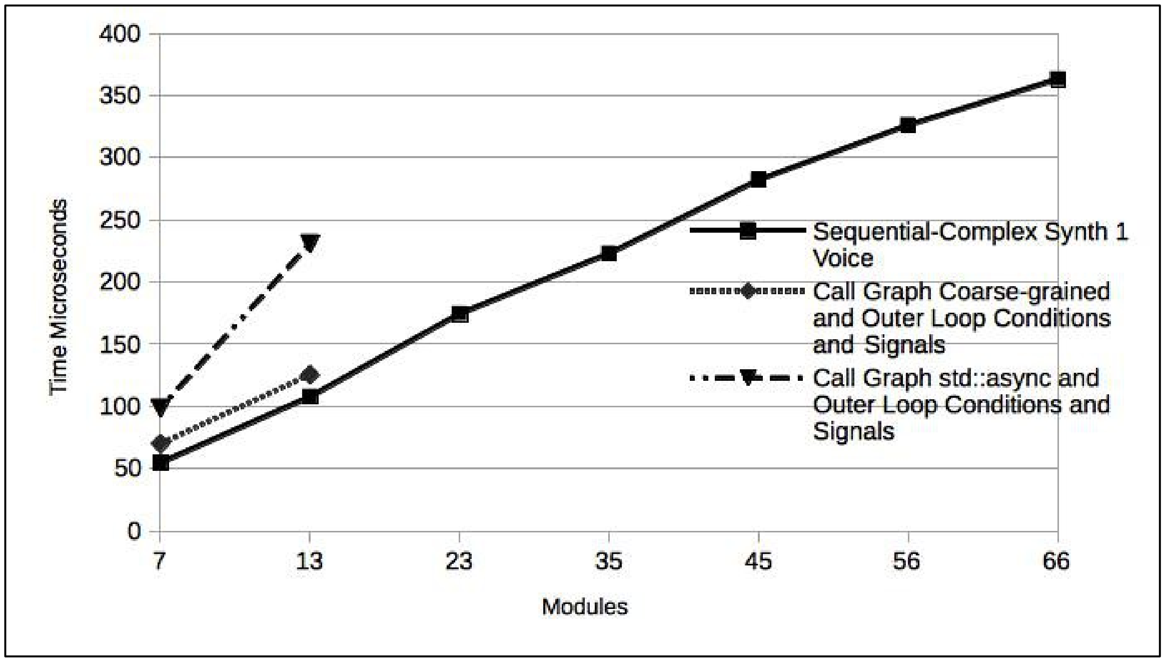


Figure 30: Complex Synth Test, One Voice: Mixed Version Outer Loop, Conditions and Signals and Call Graph, Standard Futures Coarse-grained and std::async .

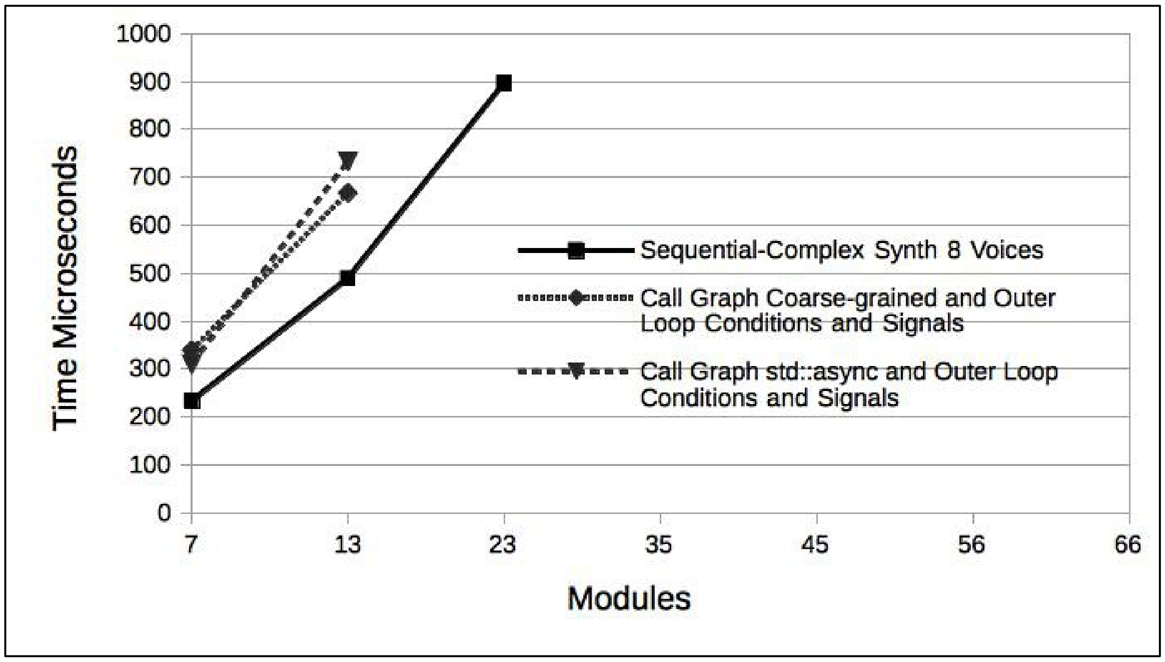


Figure 27: Complex Synth Test, Eight Voices. Mixed Version Outer Loop, Conditions and Signals and Call Graph Standard Futures Coarse-grained.

From the graphs in Figures 29, 30 and 31, it is apparent that any attempt to combine the parallel techniques results in failure specifically when testing the Complex Synths. None of the mixed versions is able to outperform the sequential version and most perform worse, both in relation to speedup and throughput, showing the overhead parallelization imposes on the system, so much so, that trying to combine any of the techniques ultimately fails due to this overhead.

6.5 Thread Pool Implementations

The performance metrics are important in the execution time of the program and the effectiveness of the thread pool is an important element of these metrics. The limitations of the Boost Lock-Free Queue, which doesn't allow the use of smart pointers, and forces the use of raw pointers, means that the use of the Boost Lock-Free Queue is limited to the outer loop parallelization and only if futures aren’t used by the parallelization technique. For the most part, a standard lock-based queue is used when implementing the call graph parallelization because of the use of standard futures, which require the use of either unique or shared pointers. The other possibility is to use a lock-free single producer single consumer (SPSC) queue, which does allow the use of smart pointers. The basic design of the lock-based queue and the SPSC Queue came from C++ Concurrency in Action [23] and both were tested using the call graph routine coarse-grained, the results of which are perhaps unexpected. Although the SPSC Queue has some performance benefits, both queues, the lock-based and the lock-free SPSC queue, perform all the tests with pretty much the same results. Figures 32, 33 and 34 show that both queues tested still perform the call graph parallelization better than the sequential version, but neither queue gives any performance benefits over the other.

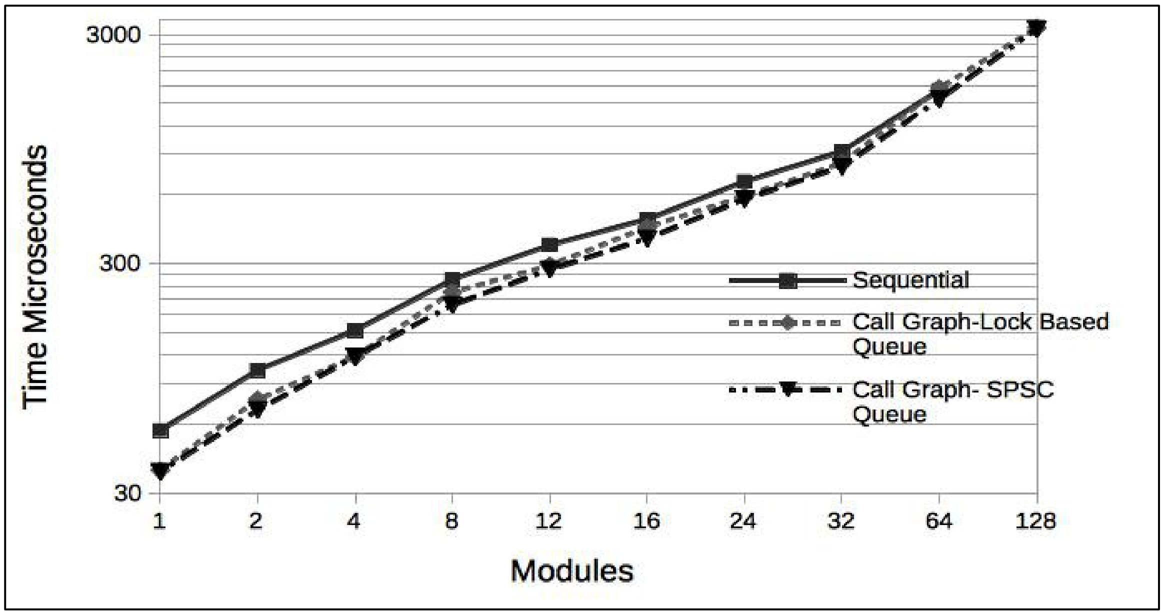


Figure 28: Simple Synth Test. Call Graph Parallelization comparing the performance of the Lock Based and SPSC Queues

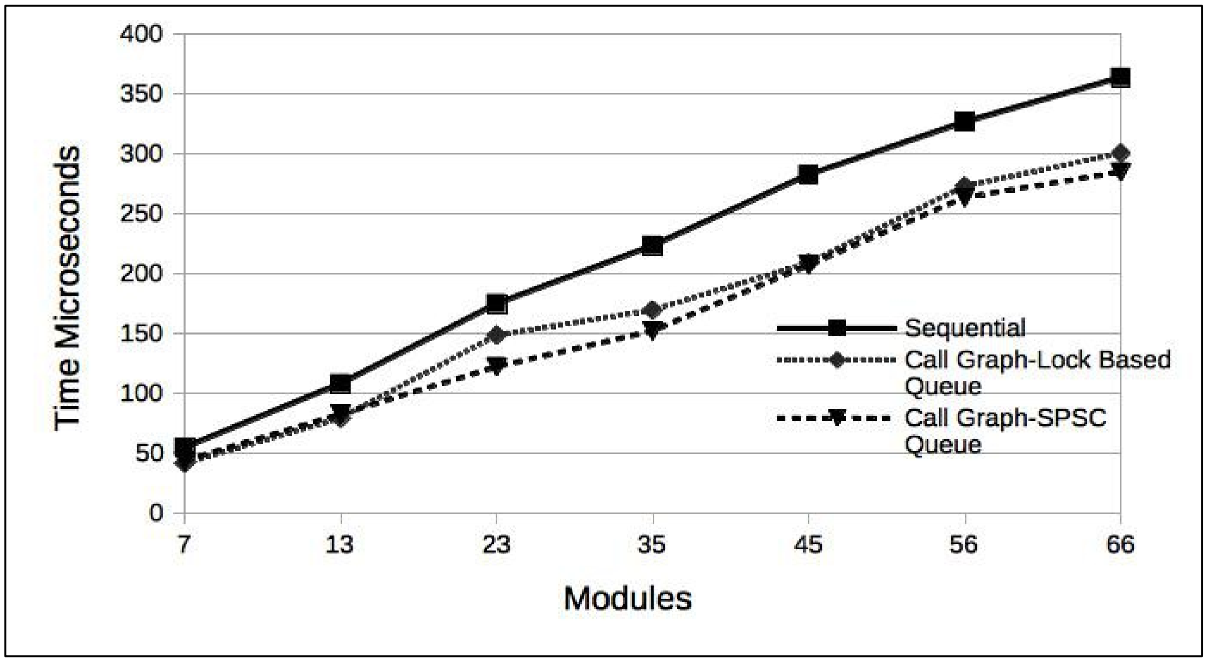


Figure 29: Complex Synth Test, One Voice. Call Graph Parallelization comparing the performance of the Lock Based and SPSC Queues

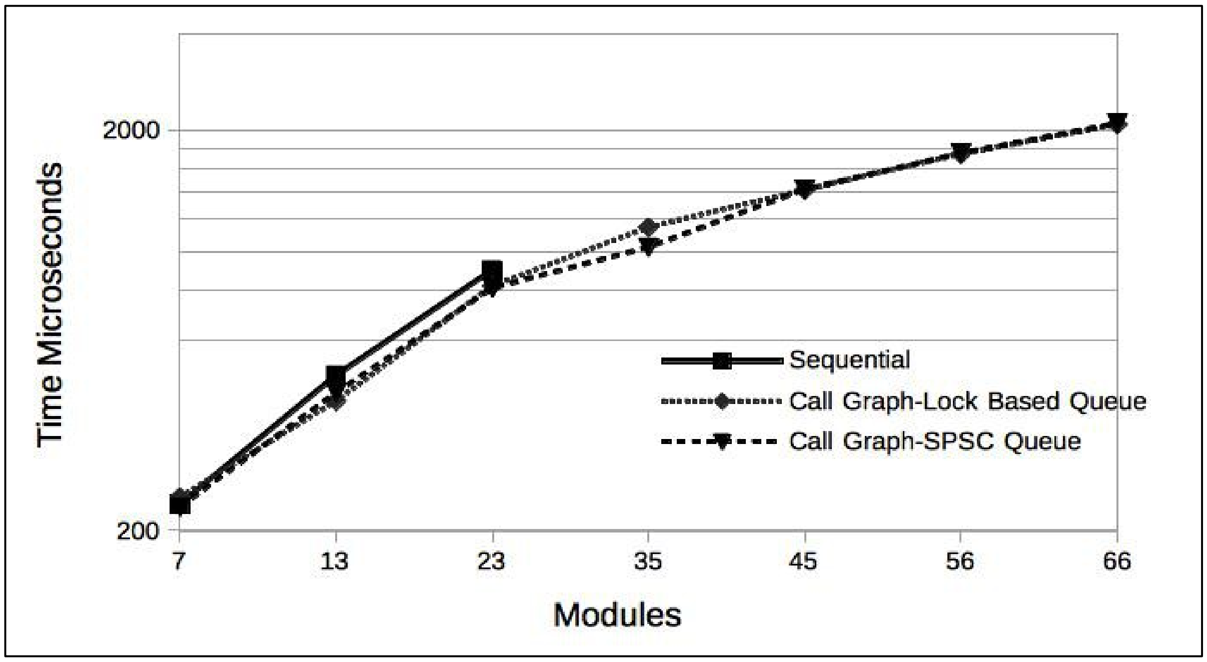


Figure 30: Complex Synth Test, Eight Voices. Call Graph Parallelization comparing the performance of the Lock Based and SPSC Queues

The outer loop parallelization was also tested using a standard lock-based queue instead of the Boost Lock-Free Queue, in order to see if the Boost Lock-Free Queue performs any better. Results from the Simple Synth Test (Figure 35) show that the lock-based queue is slower for lower voice counts. As the number of voices is increased both queues have similar performance and both offer the same throughput.

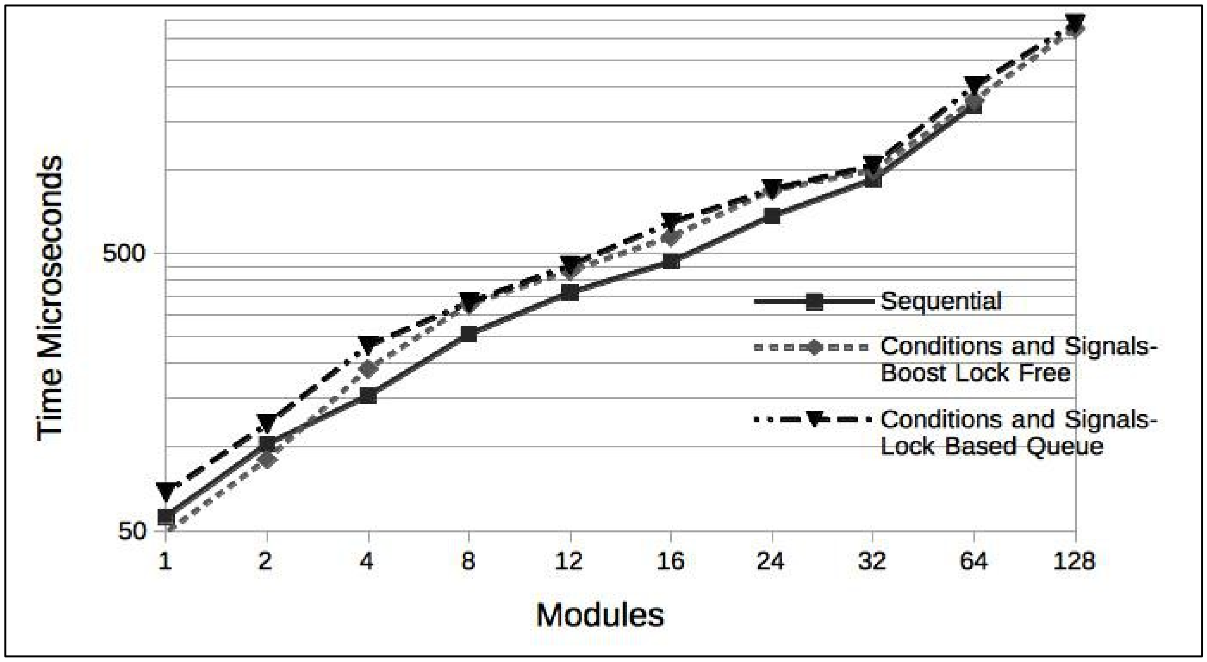


Figure 31: Simple Synth Test. Outer Loop Parallelization comparing the performance of the Lock Based and Lock Free Queues

From the results from the Complex Synth Test, using one voice, (Figure 36) it is clear that the Boost lock-free queue performs much better than the lock-based queue, but both perform just as well when running eight voices, and both offer maximum throughput (Figure 37).

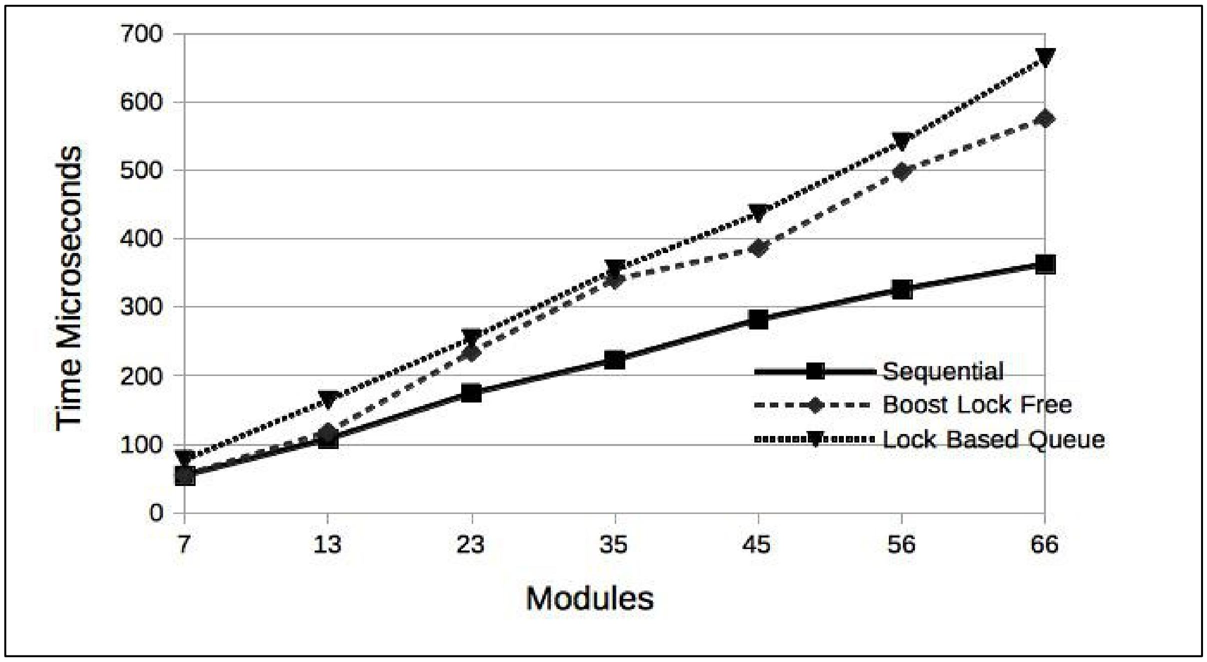


Figure 32: Complex Synth Test, One Voice. Outer Loop Parallelization comparing the performance of the Lock-based and Lock-free Queues

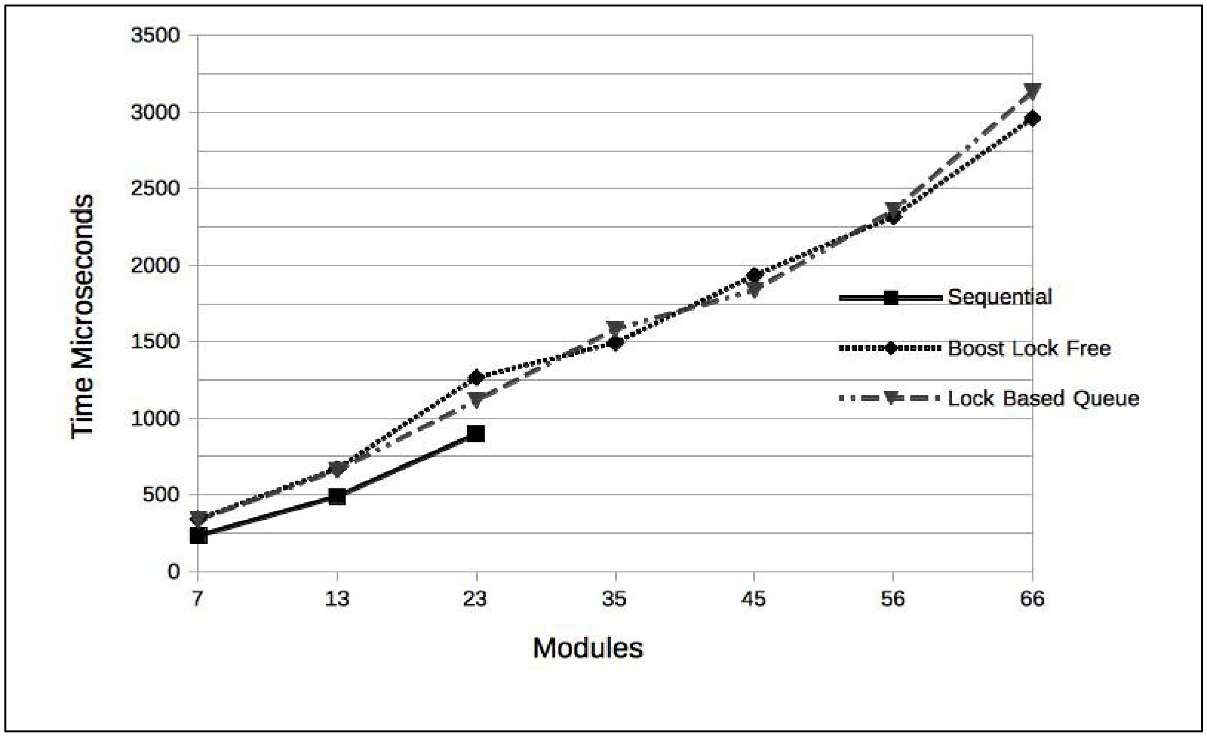


Figure 33: Complex Synth Test, Eight Voices: Outer Loop Parallelization comparing the performance of the Lock Based and Lock Free Queues

These results suggest that although the use of a lock-free data structure could be warranted for some patches, where there is a limited amount of strain placed on the queue, as more strain is placed on the queue, both the lock-based and lock-free queue perform just as well. This is contrary to [2], which suggests that audio programs should use lock-free data structures. These patterns, however, were based on the concurrency primitives that were available to programmers in 2005. The assumption now is that the newer C++ concurrency primitives perform much better than their older counterparts [24], meaning that for some applications, the implementation overheads of a lock-free queue may not warrant its use.

CHAPTER 7 SUMMARY, CONCLUSION AND FUTURE WORK

7.1 CONCLUSION

The parallelization techniques were based on the outer loop parallelization and the parallelization of the call graph. The three tests performed using the AMS were specifically designed to stress different aspects of the engine, in order to test the effectiveness of the parallelization techniques.

The outer loop parallelization technique using the independent nature of the voices of the AMS initially seems the simplest but is, due to data races generated between the modules, the most difficult to implement with any guarantee of correctness without compromising performance. The “naïve” method performs well with one thread in the thread pool but it is difficult to calculate performance metrics because the execution times of the independent voices are not measurable without using a barrier. When using more than one thread in the thread pool a data race occurs due to read-before-write errors occurring between the shared data of the AMS modules. Using futures as barriers, whether fine or coarse-grained, guarantees correctness but is too slow to offer any performance gains. The introduction of conditions and signals to avoid data races between threads doesn’t offer any gains relative to speedup but does show gains in throughput.

Initially the complexity of tracing the call graph of the AMS and then creating futures to be executed independently seems to require too much overhead to warrant attention. The decision to trace the call graph *on the fly* reduces this overhead significantly and allows for changes of the call graph in real-time. The fine-grained solution fails for more complex patches but the introduction of a coarse-grained solution increases both speedup and throughput. The coarse-grained solution limits the maximum number of futures that can be created in a patch and offers gains in two ways; the number of futures is reduced and so is the search for concurrent paths. Speedup is approximately 1.6 for more simple patches and there is a three-fold increase in the throughput. Greater throughput means the performance of the engine is more stable when there are more voices and modules been used in a complex patch.

The final tests using the Standard Async Library also performs well and, although slower than the implementation using coarse-grained futures, still offers maximum throughput for the

Complex Synth test. It also offers the programmer a simpler solution in that it no longer requires a thread pool but relies on the operating system to schedule parallel execution of the individual modules.

In fact the three parallel methods, the outer loop using conditions and signals, parallelizing the call back using coarse-grained standard futures and the implementation using std::async offer much better throughput and stability compared to the sequential version. Although speedup is nominal, and at times not measurable because the sequential version is not able run all the test cases, the throughput is at times three-fold, enabling the user to design more complex patches without the limitations inherent in the sequential engine.

7.1.2 Generalized Results

Although the results above are specific to the parallelization of the AMS, other audio programs follow the same design principles of the AMS and, although the modularity of design may not be apparent in the user interface, the design principles and the flow of control from one set of generating or transforming functions to another (e.g., filters, oscillators, etc.) are inherent in all audio programs and could be expressed as a directed graph similar to the call graph of the AMS. The concept of polyphony is also not unique to the AMS and, although the implementation details may vary, the fact that the individual voices of an instrument are independent is generally the case for all soft-synths.

Some important parallelizing techniques have also been evaluated. Thread reuse and limiting the instantiation of objects during the execution of the application are important design concerns. Reducing overhead whenever possible, specifically concerning the development of coarse-grained parallelization methods, showed that when used with care barriers and synchronization primitives may be used without compromising performance.

One of the three parallelization techniques that performed well for the AMS, (e.g., the coarse-grained call graph parallelization, the call graph parallelized using std::async or the outer loop parallelization using conditions and signals), could easily be adapted to parallelize other similarly designed software synthesizers. The use of the C++ Standard Library offers an easily adaptable interface to enable programmers to apply the techniques of parallelizing the AMS to other audio applications.

7.2 FUTURE WORK

Of all the different versions of the engine described, the naive implementation of the program, where no synchronization primitives were used, still performs as well as the other good performers, specifically, the coarse-grained call graph parallelization using standard futures and the implementation using the std::async library. The "naïve" version, however, cannot be accurately measured and the data race sets in when more than one thread is used in the thread pool. The limited use of synchronization primitives makes the "naïve" version attractive, as it is the use of synchronization that has a marked affect on performance.

It may be possible to obtain speedup as well as maximize throughput, by implementing non-blocking lock-free methods and try to avoid the use of standard synchronization primitives. If the use of locks can be decreased, or not used at all, by means of some lock-free synchronization techniques, the program may not only have greater throughput but also perform faster by increasing speedup, allowing for lower latencies in the system. The possibility of reducing synchronization primitives when parallelizing the AMS should be investigated further.

There is also another possible way to parallelize the AMS that hasn't been investigated. The call graph could be run for each independent voice. This would involve duplicating the call graph of a patch and then executing the complete call graph for each voice independently. This may perform well as there would be less synchronization between the call graphs as each voice can run concurrently and complete the call graph before needing to synchronize with the other voices. The added complexity from calculating all the modules for each independent graph may make this method similar in performance to the other parallelization techniques. Performance may suffer if the call graph is large and there are too many voices to process. Depending on the performance of the thread pool the wait time for data to be processed may become too long to benefit using this parallelization technique. The possibility of increasing concurrency and reducing the use of synchronization primitives using this method does, however, warrant further study.

BIBLIOGRAPHY

[1] M. Puckette. Thoughts on Parallel Computing for Music. *In Proc. Of the ICMC 2008*, 2008, pp. 187.

[2] R. B. Dannenberg and R. Bencina. Design Patterns for Real-Time Computer Music Systems. *In ICMC 2005 Workshop on Real Time Systems Concepts for Computer Music*, 2005.

[3] M. Nagorni. Modular Synthesis with AlsaModularSynth1.5.2. Available from: <http://alsamodular.sourceforge.net/>.

[4] 2011 C++ Thread Library Reference. Available from: <http://en.cppreference.com/w/cpp/thread>.

[5] D. Buttlar, J. Farrell, and B. Nichols. PThreads Programming. A POSIX Standard for Better Multiprocessing. O'Reilly Media, 1996.

[6] Threading Building Blocks (Intel® TBB). Available from: <https://www.threadingbuildingblocks.org/>.

[7] B. Chapman, G. Jost and R. van der Pas. Using OpenMP. Portable Shared Memory Parallel Programming. The MIT Press, Oct. 2007.

[8] S.Wilson, D. Cottle and N. Collins (Editors). The SuperCollider Book. The MIT Press, Apr. 2011.

[9] Faust du Grame. Available from: <http://faust.grame.fr/>.

[10] Pure Data [online]. Available from: <https://puredata.info/>.

[11] M. Puckette. Pure Data. *In Proc. of the ICMC 1996*. San Francisco, USA, 1996, pp. 224-227.

[12] SuperCollider Code. Available from: <http://supercollider.sourceforge.net/audiocode-examples/>.

[13] R. Bencina, "Inside Scsynth". The SuperCollider Book. The MIT Press. April 2011, pp. 273

[14] T. Blechmann. Supernova, a multiprocessor-aware synthesis server for SuperCollider. *In Proc. of the 8th International Linux Audio Conference*, Netherlands, 2010, pp. 141–145

[15] T. Blechmann, Supernova -A Multiprocessor Aware Real-Time Audio Synthesis Engine For SuperCollider. Masters Thesis, *Vienna University of Technology*, 2011.

[16] Y. Orlarey, S. Letz, and D. Fober. Adding Automatic Parallelization to Faust. *In Proc. of the Linux Audio Conference (LAC-2009)*, Parma, Italy. April 2009.

[17] MIDI specifications. Available from: <http://www.midi.org/>.

[18] Jack Audio Connection Kit. Available from: <http://jackaudio.org/>.

[19] H. Haas. The Influence of a Single Echo on the Audibility of Speech. *JAES*, vol. 20 no. 2, March 1972, pp. 146-159.

[20] D. F. Bacon, S. L. Graham and O. J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, vol. 26 Issue 4, Dec. 1994, pp. 345-420.

[21] A. Kogan and E. Petrank. A methodology for creating fast wait-free data structures. *In Proc. 17th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP),* 2012, pp, 141–150.

[22] Boost Lock-Free Reference Documents. Available from: <http://www.boost.org/>.

[23] A. Williams. C++ Concurrency in Action: Practical Multithreading. Manning Publications. February 2012.

[24] N. Pipenbrinck. Online discussion on lock-free data structures. Available from: <http://stackoverflow.com/questions/27738660/multithreaded-realtime-audio-programming-to-block-or-not-to-block>.

[25] Linux Professional Audio Documentation. Available from: <https://wiki.archlinux.org/>.

[26] T. Leng, R. Ali, J. Hsieh and V. Mashayekhi. An Empirical Study of Hyper-Threading in High Performance Computing Clusters. *In Proc. of Supercomputing 2002*, Baltimore, USA. 2002.

[27] B. Blarney. OpenMP Tutorial. Available from: <https://computing.llnl.gov/tutorials/openMP/>.