On the synthesis and processing of high quality audio signals by parallel computers

Nicholas James Bailey

ON THE SYNTHESIS AND PROCESSING OF HIGH QUALITY AUDIO SIGNALS BY PARALLEL COMPUTERS

Nicholas James Bailey,

B.Sc. J.Hons (Dunelm)

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE COUNCIL OF THE UNIVERSITY OF DURHAM FOR THE DEGREE OF DOCTOR OF PHILOSOPHY (PH.D.).

DECEMBER 1991

Declaration

I hereby declare that this thesis is a record of work undertaken by myself, that it has not been the subject of any previous appplication for a degree, and that all sources of information have been duly acknowledged.

N. J. Bailey, February 1992

Abstract

This work concerns the application of new computer architectures to the creation and manipulation of high-quality audio bandwidth signals. The configuration of both the hardware and software in such systems falls under consideration in the three major sections which present increasing levels of algorithmic concurrency.

In the first section, the programs which are described are distributed in identical copies across an array of processing elements; these programs run autonomously, generating data independently, but with control parameters peculiar to each copy: this type of concurrency is referred to as *isonomic*.[1]

The central section presents a structure which distributes tasks across an arbitrary network of processors; the flow of control in such a program is quasi-indeterminate, and controlled on a demand basis by the rate of completion of the slave tasks and their irregular interaction with the master. Whilst that interaction is, in principle, deterministic, it is also data-dependent; the dynamic nature of task allocation demands that no *a priori* knowledge of the rate of task completion be required. This type of concurrency is called *dianomic*.[2]

Finally, an architecture is described which will support a very high level of algorithmic concurrency. The programs which make efficient use of such a machine are designed not by considering flow of control, but by considering flow of data. Each atomic algorithmic unit is made as simple as possible, which results in the extensive distribution of a program over very many processing elements. Programs designed by considering only the optimum data exchange routes are

[1] from classical Greek: *iso-* meaning 'the same', *nomos* meaning 'law'.
[2] *dia-* meaning 'apart' etc. as in *diameter*.

said to exhibit *systolic*[3] concurrency.

Often neglected in the study of system design are those provisions necessary for practical implementations. It was intended to provide users with useful application programs in fulfilment of this study; the target group is electroacoustic composers, who use digital signal processing techniques in the context of musical composition. Some of the algorithms in use in this field are highly complex, often requiring a quantity of processing for each sample which exceeds that currently available even from very powerful computers. Consequently, applications tend to operate not in *'real-time'* (where the output of a system responds to its input apparently instantaneously), but by the manipulation of sounds recorded digitally on a mass storage device.

The first two sections adopt existing, public-domain software, and seek to increase its speed of execution significantly by parallel techniques, with the minimum compromise of functionality and ease of use. Those chosen are the general-purpose direct synthesis program CSOUND, from M.I.T., and a stand-alone phase vocoder system from the C.D.P..[4] In each case, the desired aim is achieved: to increase speed of execution by two orders of magnitude over the systems currently in use by composers. This requires substantial restructuring of the programs, and careful consideration of the best computer architectures on which they are to run concurrently.

The third section examines the rationale behind the use of computers in music, and begins with the implementation of a sophisticated electronic musical instrument capable of a degree of expression at least equal to its acoustic counterparts. It seems that the flexible control of such an instrument demands a greater computing resource than the sound synthesis part. A machine has been constructed with the intention of enabling the *'gestural capture'* of performance information in real-time; the structure of this computer, which has one hundred and sixty

[3]Also of classical Greek derivation: used in medical terminology to describe the contraction of the heart. Hence it describes a system through which data is 'pumped' continuously.

[4]The Composers' Desktop Project

high-performance microprocessors running in parallel, is expounded; and the systolic programming techniques required to take advantage of such an array are illustrated in the Occam programming language.

Acknowledgements

I would like to express my gratitude to the many people who have supported and encouraged my work over the previous three years, including the 'users' who have so carefully tested the software and hardware systems described here. I am especially indebted to my supervisor, Dr Alan Purvis, for his administrative support as well as his technical advice; also to my advisor on all musical matters, Dr Peter Manning, without who's enthusiasm for using the products of this research as teaching material, the project would have lacked a certain impetus. Dr Michael Clarke of the Music Department at Huddersfield Polytechnic was also of immense help in understanding the structure of the original CSOUND program, having himself previously written additional unit generator modules for it.

The final part of this thesis describes the construction and testing of a parallel computer using 160 INMOS transputers as its processing elements. I wish to acknowledge the generous donation of these parts by INMOS Ltd., without which the component cost of the machine would have made its construction impossible.

My thanks are also extended to my parents for patiently proof-reading the draft document, of which they claim not to understand a single word.

Contents

List of Tables

List of Figures

Chapter 1

Direct Audio Synthesis by Digital Computer

1.1 Introduction

DIRECT AUDIO SYNTHESIS refers to the method of sound generation where an algorithm, designed to emulate some real or imaginary musical instrument, is executed by a digital computer in order to produce a numeric representation of the required sound. The process may take place either in 'real-time', when computation proceeds sufficiently rapidly to provide the next digital audio sample as required; or in 'non-real-time', when the synthesising computer does not have sufficient processing power to provide samples at such a rate. In the latter case, the generated data may be stored onto some mass-storage medium whence it may subsequently be replayed.

Occasionally, the term 'near-real-time' has been adopted, meaning that the processing time required to produce a given sound is not much greater than that sound's duration. Strictly, this is an abuse of the term 'real-time', as in its true usage it denotes a system which responds apparently instantaneously to demand. Any system for the generation of music which requires that the sound be recorded and subsequently played back cannot be real-time. Even if a computer existed which could generate a whole hour of music during the space of a second, recording the program's results onto a hard disk, the system could not be accurately described as real-time because the entire input data set had to be

made available before processing could begin. In order to avoid this confusion, the term 'actual speed' is used. A machine which produces a second's worth of output sound in one second of processing time is working at actual speed, although not necessarily in real-time.

The direct synthesis of sound from computers dates back to the mid-1950s when Max Mathews began experimenting with the generation of waveforms using digital techniques at Bell Telephone Laboratories, New Jersey. These were quickly expanded into a series of software systems known generically as the MUSIC programs, subsequently developed and extended by Mathews and others at various computer music centres world-wide. Initially these programs were run on mainframe computers, the digitised output samples being accumulated on tape or disk for subsequent conversion via digital-to-analogue converters.

In due course versions were written for mini- and micro-computers, in particular, in 1979, MUSIC11 for the PDP11 and, in 1986, CSOUND for machines running C compilers, both produced at M.I.T. by Barry Vercoe.[1] These modern derivatives provide a wide range of facilities to the aspiring composer including the processing of external sound material acquired through digital capture. Software synthesis programs such as these offer the most general means for generating and manipulating sound material, but at a significant cost to the user in terms of computing time. Indeed, until very recently, it has been impossible to contemplate running such systems in real-time or even in near-real-time. Commercial synthesisers side-step this processing constraint by using custom-designed hardware and by restricting the choice of performance characteristics. The penalty is a restriction on flexibility of control for the composer/performer.

The prospect of running software synthesis at or near actual speed has stimulated a renewed interest. In the late 1950s, delays between submitting tasks to the batch stream of a computer and finally hearing the results were often measured in days. The advent of mini-computers dedicated to the application reduced typical turnaround times first to hours and then minutes as the speed of these systems improved. This progression has not been entirely consistent for, as

will be seen in due course, the desire to increase accessibility at low cost has led to the transfer of digital synthesis/processing programs[2] to microcomputers which are still considerably slower than their contemporary mini-computers. This has in turn stimulated a demand for a low cost computer which can not merely equal but considerably surpass the performance of these machines.

1.2 Review

Direct synthesis, then, is computationally expensive. The most flexible form of direct synthesis is *additive synthesis*, which relies on the fact that any periodic waveform may be represented to an arbitrary degree of accuracy by the sum of sufficient sinusiods of various frequency, amplitude and phase. This is an indisputably powerful technique, as it is possible to generate any imaginable sound using it. Unfortunately, although some implementations are available such as the Bradford Musical Instrument Simulator and Workstation,[3] and software modules which support additive synthesis are provided on many DSP oriented hardware accelerators,[4] the algorithm generally fails to render its potential because of the very high control bandwidth required: each oscillator, and there may be twenty or thirty required for adequate results,[5] requires updating rapidly (in principle, at the audio sample rate). The control bandwidth required is consequently very high, in fact exceeding the data-rate of the actual signal produced. Work by Serra, Rubine & Dannenberg[6] and Sasaki & Smith[7] promises to reduce the quantity of control information somewhat, although it remains a very significant demand upon the control software, as well as a severe challenge to the composer who must ultimately be the source of the control data. The advantage of the additive synthesiser it that modification of the control parameters produces intuitively obvious changes in the output signal.

Akin to additive synthesis is *subtractive synthesis*, where a signal rich in harmonics, or even noise, is passed through a filter to produce the desired result. This can be as intuitively accessible as additive synthesis for certain applications,

and has been used to generate realistic vocal timbres.[8] Although some optimisation of the processing is possible by careful design of the filtering algorithms, the control bandwidth a high-quality subtractive synthesiser remains necessarily high.

More specialist direct synthesis algorithms have been developed for the characteristic demands of the computer musician; they share the claimed improvements of reduced or more intuitive control requirements. *Granular synthesis*[9, 10] and *time-domain formant wave function synthesis* [11, 12] exemplify these: the former generates a signal by the summation of tonebursts with Gaussian amplitude envelopes, or 'granules'; the latter by modelling the resonances of the vocal tract in the time domain. Manipulations of a signal in a way which has some sonic basis is also possible: *ring modulation*, or four-quadrant multiplication, introduces sum and difference frequencies into the output signal; controlled harmonic distortion may be achieved by applying transfer functions derived from Chebychev polynomials, in a process known as *wave shaping synthesis*.

It is rarely feasible to produce high fidelity direct synthesis in real-time using a general-purpose microprocessor, still less often is it possible to supply the electroacoustic composer with real-time general-purpose direct synthesis software which will meet his requirements. The reason for the continued popularity of such programs lie in their flexibility and prototype development facilities. Direct synthesis was used in the testing of the frequency modulation algorithms subsequently implemented in VLSI integrated circuits and used in many commercial synthesisers in the popular music industry.[13] The alternative to direct synthesis is to use event-based synthesis systems, where remote dedicated synthesisers are controlled by relatively low bandwidth control signals. The most popular protocol which has evolved for this purpose, the Musical Instrument Digital Interface (MIDI), expects that all of the (real-time) performance information will be transmitted down a single 32kbaud asynchronous serial line. This is clearly aimed at the commercial musician, whose output is predominately note-event based and where easily achieving rhythmical and dynamic uniformity is considered more

important than the provision of extended expressive capacity. Conversely, the electroacoustic composer often writes gesturally, concentrating on the evolution of particular sounds as well as their temporal position.[14, 15, 16]

That, briefly, is the case for allowing the level of control of a synthesiser to be determined by the composer's demands rather than the restrictions of the control mechanism. However, it would be not be fair to suggest that the role of the MIDI synthesiser in 'serious music' composition is an inconsiderable one; moving on to examine a specific implementation would be unwise before establishing an understanding of the facilities of and concepts behind MIDI and similarly controlled systems.

One of the principal uses of MIDI in music technology research is as a 'synthesis back-end' to projects which are concentrating on some problem at a higher level than the synthesis algorithms themselves. Work by Zicarelly[17] resulted in the production of two commercial interactive programs: 'M' maps gestural control available from the computer's peripherals onto the parameter space defined by MIDI's command set, thus enabling direct access to the (albeit restricted) timbres available from a given synthesiser; 'Jam Factory' is a tool for algorithmic composition which permits the use of Markov Chains as a method of computer improvisation. The programs make no attempt to operate outside the MIDI command set, and are therefore still significantly restricted by it; but they represent an excellent example of the benefits of a sophisticated environment. Haus[18] uses MIDI to realise automatic and semi-automatic compositions, the description and performance of musical processes, musical transformation through homologies, automatic score transcription from tape, score analysis, score synthesis using two-variable functions, and transliteration of literary texts into music. Much algorithmic composition requires nothing more sophisticated than a MIDI synthesiser on which to produce its output: Langston[19] considers six different methods for machine composition; none of the algorithms is concerned with more than the generation of discrete note events. It may also happen that the gestural information which MIDI does capture (normally only the pitch and loudness of a note

is reported) is sufficient for the study being undertaken. The Kansei music system due to Katayose *et al.*[20] attempts the production of 'musically acceptable' performances from the written score by variation to only time and amplitude parameters; even a method for the analysis of the 'emotional content' of a piece of performed music has been put forward[21] although it seems hard to understand how to quantify such measurements. Finally, MIDI has been used as a shorthand method of entering the note-event information into direct synthesis programs, for subsequent, enhanced, non-real-time processing.[22]

1.3 Direct Synthesis Programs in Use

The composition of electroacoustic music presently requires a sound knowledge of the fundamentals of acoustics and digital signal processing theory and computer programming, in addition to the creative skills associated with the normal practice of a composer. Because the human performer is absent from the finished piece, it is possible to make demands of the "performer" (i.e. the computer performing the synthesis) which would normally be technically unacceptable. Consequently, the electroacoustic composer would be expected to have a greater knowledge of the theories of perception and cognition than the classical composer; McAdams & Bregman[23] write of the perception of separate musical streams in the context of electroacoustic composition, where it is possible to increase the number of events per second almost indefinitely. Haynes[24] reviews the musician machine interface in the context of non-real-time systems, and in the late 1970's, the Structured Sound Synthesis Project[25, 26] within the Computer Systems Research Group at the University of Toronto, Canada, suggested methods of optimising the ergonomics of a computer synthesis system.

CSOUND, a direct synthesis program with a wide range of facilities, has been written and placed in the public domain by Barry Vercoe at M.I.T.: since there was already much local experience in composition using a variant of this package,

it seemed appropriate to make it concurrent with a view to increasing its performance by at least two orders of magnitude. Other research within the Music Technology Group has shown the benefit of retaining the original program structure as a powerful host for new synthesis algorithms: the Vocel synthesis module coded by Clarke[27] in PDP11 assembler took less than a day to install in its C form on the new concurrent version. The CSOUND program is also distributed in the U.K. by the Composers' Desktop Project Ltd. (C.D.P.),[28] in a version for a microcomputer.[1] The C.D.P. was formed in the 1980's with the specific intention of bringing general direct synthesis financially within the reach of individuals rather than corporations, but the speed of their machine, chosen at the time for its low price/performance ratio, led to program execution times so protracted as to strain the patience of the most persevering user. Students of composition at the University of Durham's School of Music, even though they were running a similar program on a much faster PDP11 series minicomputer, would often sleep in the computer room awaiting the completion of the program: the microcomputer version was almost an order of magnitude slower still. Before explaining the strategy adopted in order to reduce execution times of the CSOUND package by using multiple processors, it is necessary to be familiar with its internal program- and data-structures.

1.4 Data Structures of a Direct Synthesiser

Input data for CSOUND are partitioned into two files: the *score file* and the *orchestra file*. The content and purpose of these two files is analogous to conventional score and orchestra. Information from these files is used by the direct synthesis algorithms to produce digital audio samples, which are recorded onto hard disk as a *sound file*.

Each line of the score file describes a single note event; each event has parameters appended such as duration, pitch, envelope information and so on, including

[1] the Atari Corp. range of machines.

```
c score 100
t 0 60
f1 0 32 10 1
f2 0 512 10 3 7 6 2 1
f3 0 8 -2 1 1.4 0.9 1.1 0.5 0.7 1.5 0.8
c start dur amp freq ind depth speed
i3 0 -1
i1 0 6 5000 440 0 0.8 25
i1 1.8 7 5000 875 0 1.7 15
i1 3 6 5000 1200 0 0.8 35
i1 4.9 4 5000 575 0 0.5 27
i1 7.3 3.4 5000 270 0 0.5 45
i2 0.8 3 5000 1700 0 0.8 25
i2 2.1 6 5000 320 0 1.4 15
i2 3.9 4.5 5000 650 0 1.4 35
i2 6.1 4 5000 940 0 0.8 40
f0 11.5
e
```

Table 1: An Example Score File

information fields specified by the composer and used to pass instrument-specific information into the program. An example CSOUND score is shown in table 1.

All lines have a similar format: the command, a single character sometimes followed immediately by an integer qualifier; the start time of the command in beats; a series of command specific parameters, the first of which will be duration in beats if the command was to start an instrument. The score commands are very much abbreviated, but some are recognisable musical directions: the tempo is defined as 60 beats per minute; three 'function tables' (look-up tables) are declared and initialised; instrument three is instantiated to run from zero time and to continue 'forever' (the value -1 in the second field specifies infinite endurance); instruments 1 and 2 then play various notes for the given start times and durations.

The orchestra file contains a definition of each of the instruments available to the score, using a simple declarative programming language. Part of an example orchestra, designed to be used with the above score, is given in table 2.

```
; orchestra 100
sr=48000
kr=2400
ksmps=20
nchnls=2

instr 1
ga1 init 0
k4 line p7 ,p3 ,p7/5
k5 line p8 ,p3 ,p8/5
a1 linseg 0 ,0.04 ,1 ,p3-0.09 ,1 ,0.04 ,0
k1 phasor k5/ftlen(3)
k2 table k1*ftlen(3) ,3
a2 oscili 1 ,k5/2 ,1
a3 oscili a2*a1*p4 ,p5+(p5*k2-p5)*k4 ,2
ga1 = ga1+a3
 outs1 a3
endin

instr 3
a1 reverb ga1 ,1.3
 outs a1 ,a1
ga1 = 0
endin
```

Table 2: An Example Orchestra File

The score is divided into two sections: the former providing global information about the output format required for the sound file that is generated, the latter defining each of the instruments used in the score.

The assignments to the special variable `sr` defines the sample rate of the output file in samples per second. One of the standard commercial sample rates is likely to be used (i.e. 32000, 41400 or 48000 samples per second), but lowering the sample rate obviously decreases the processor time spent in calculating output samples proportionately. For this reason, sample rates of 24kHz or even as low as 16kHz are often used for sketching. `nchnls` specifies the number of output channels which are to be recorded on the sound file; samples from each channel being interleaved. Permitted formats are monophonic or stereophonic. This permits algorithms containing realistic reverberation and spatial movement effects.

In order to achieve an economy of computation, CSOUND introduces the concept of a *control rate*. As will be seen, this is fundamental to the structure of the modules which actually perform the synthesis. Some variables, for example those used to envelope the notes, may be updated at a rate lower than the audio sample rate with little impact on the perceived quality of the sound. Such control-rate variables are calculated only `kr` times per second, or once every `ksmps` samples. The values seen in the example of a control rate update occurring every twenty audio samples is fairly typical.

The instrument definition part of the orchestra is a simple procedural programming language, although its syntax is rather cryptic. The general form is:

⟨*command*⟩ :=[⟨ *result_list*⟩]⟨*operator*⟩ {⟨*function*⟩}

⟨*result_list*⟩ := ⟨*variable*⟩{⟨*variable*⟩}

⟨*variable*⟩ := [*'g'*]⟨*'i'*‖*'k'*‖*'a'*⟩⟨*identifier*⟩

A function may be any one of the synthesis functions taken from the CSOUND command set; functions evaluate expressions involving constants including parameters to read at run-time from the additional fields in the score. Like the

FORTRAN programming language, variable names have connotations. A variable is local to the instrument in which it is used, unless prefixed with a g which causes it to be global to the entire orchestra. Depending upon the last (or only) letter, the variable may be evaluated once per note (i denoting initialisation), at the control rate ('k') or at the audio rate ('a'). During synthesis, execution proceeds through the currently active instruments line by line for every sample produced. It is because of the interpretive nature of CSOUND that its execution becomes so computationally expensive.

A significant detail in the above example orchestra is that the instrument which performs all of the sound sample calculations does not itself write the results to disk. Instead, the global ga1 is used to accumulate the results from all of the sounding instruments and pass the result to instr 3; a reverberation function is then applied, the result written, and ga1 reset. Intra-orchestra communication by global access shared variable, it will be shown, has significant impact upon the strategy chosen in making the program run concurrently.

1.5 The Software Structure of CSOUND

The CSOUND source listings, as supplied by MIT, contain approximately twenty separate source modules, written entirely in C. The interrelation between these modules is shown in figure 1. The Root Supervisor module, RSUPER, is an addition by Durham Music Technology which improves the efficiency of the program on transputers.

Running the CSOUND synthesiser on a particular score and orchestra is a two stage process. First, the score is pre-processed by the 'sort' program. This program runs through the score sorting each event by start time and instrument number. It is also necessary at this stage to perform an adjustment to the start time and durations, which the composer will have specified in beats, so that the version of the score available to CSOUND also contains absolute times in seconds. This is a non-trivial process because, as well as coping with abrupt changes in

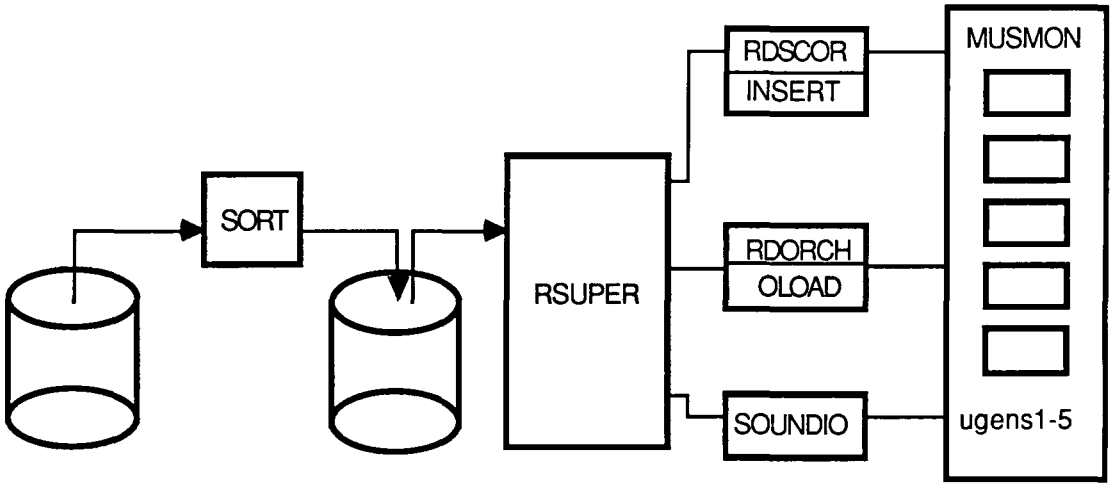


Figure 1: Software Structure

tempo, the translation algorithm must also be able to provide accelerando and ritardando capability (i.e. smooth increases and decreases in tempo). The manual refers to the score processing, including the insertion of absolute time information and rearrangement in order to produce the sorted version with all tempo changes into account, as "time-warping". The sorted and 'time-warped' version of the score shown previously appears in table 3.

This simple score has a tempo of sixty beats per minute throughout; consequently, the absolute time and duration (in seconds) will equal their original value in beats. The 'warped' (fourth and sixth) fields in each line therefore contain copies of the third and fifth.

The score having been sorted, CSOUND is now invoked specifying the desired orchestra. Program execution proceeds as follows: modules RDORCH reads the ASCII text file containing the orchestra description, and this is semi-compiled by OLOAD. The orchestra is stored internally as a structure containing pointers to functions which will be called in the synthesis of sound samples. Modules RDSCOR and INSERT then read lines of the score file, and construct a job-list

```
w 0 60
f 1 0 0 32 32 10 1
f 2 0 0 512 512 10 3 7 6 2 1
f 3 0 0 8 8 -2 1 1.4 0.9 1.1 0.5 0.7 1.5 0.8
i 1 0 0 6 6 5000 440 0 0.8 25
i 3 0 0 -1 -1
i 2 .8 .8 3 3 5000 1700 0 0.8 25
i 1 1.8 1.8 7 7 5000 875 0 1.7 15
i 2 2.1 2.1 6 6 5000 320 0 1.4 15
i 1 3 3 6 6 5000 1200 0 0.8 35
i 2 3.9 3.9 4.5 4.5 5000 650 0 1.4 35
i 1 4.9 4.9 4 4 5000 575 0 0.5 27
i 2 6.1 6.1 4 4 5000 940 0 0.8 40
i 1 7.3 7.3 3.4 3.4 5000 270 0 0.5 45
f 0 11.5 11.5
e
```

Table 3: A Sorted Score

maintaining details of the start time and duration of each instrument instantiation. The UGENS and FGENS modules contain respectively the 'unit generator' and 'function table generator' code: a unit generator is an algorithm which is responsible for assigning values to an orchestral variable; a function table generator can produce in-memory look-up tables for use by the unit generators. A range of unit generators is available, from basic oscillators and random number generators, to highly complex synthesis functions such as the FOF algorithm already mentioned.

The MUSMON ('music monitor') module invokes the unit and function generators in the appropriate order, which should result in the issue of synthesised sound samples. Input and output of sound files between the CSOUND program and the file server is performed by the module SOUNDIO.

1.6 Conclusion

The direct synthesis of audio signals in non-real-time provides the opportunity for the electroacoustic composer to specify musical constructions of arbitrary complexity. Direct synthesis programs have been written which enable the specification of a composition in terms of a score and an orchestra — entities more familiar to conventional composers. The CSOUND package, originating from M.I.T., embodies most of the synthesis algorithms which the composer is likely to need, and enables them to be coordinated through a simple declarative programming language. Unfortunately, execution times can be unacceptably long if the program is run on microcomputers. Familiarity with the data-structure and with the modules of the package allow a discussion of the large-scale modifications required to enable the program to take advantage of multiple processors running concurrently; such a discussion is contained in the following chapter.

Chapter 2

Exploiting Parallel Algorithms for Direct Audio Synthesis

2.1 Choosing a Multi-processing Strategy

In their perspicuous and comprehensive book on highly parallel computing, Almasi & Gottlieb[29] define a highly parallel processor as

> A large collection of processing elements that can communicate and co-operate to solve large problems fast.*(sic)*

Precisely the best method of achieving this end is dependent upon a number of factors, including amongst others: the performance of the individual processors; the capacity of the processing elements to perform input, output and communication tasks; the memory available to each processor; the type and availability of inter-process synchronisation; and the impact of performing communications upon the throughput of each processor. These attributes, coupled with an appreciation of the inherent granularity and extent of the application program's parallelism, must be considered in determining an optimal strategy for exploiting the speed advantages of a multi-processor.

For a large software system, where the rewriting of the whole would be impractical, there are two main methods of parallel decomposition which might be considered. It is possible either to configure a system of parallel processors according to the control structure of the program, or according to the data structure

of the program's result. The consequences of both of these points of view will be considered.

The program control mechanism is expressible as a language, as clearly indicated by the structure of the score and orchestra already described. It would be possible to parse the orchestra file, producing a list of required unit generators along with an estimate of their demands upon processor time. A configurable, parallel computer could be set up to emulate the data-flow in the orchestra; processing elements might be allotted in a way which best establishes an equilibrium between the production of data by one unit generator and the rate of data consumption by those nearer to the root of the orchestral parse tree. Messerschmitt & Lee[30] have pursued this course in the context of digital signal processing; they have developed methods of algebraic description of a network of sub-programs where the number of output samples for each input sample is strictly defined. It is possible, using this representation, to prove the correctness and establish the efficiency of a scheduling algorithm. In this scheme, processors operate essentially synchronously if a balanced workload has been achieved, pausing for a negligible time (compared with that taken in computation of the whole algorithm) if synchronisation is required. This corresponds to *systolic* parallelism. In a systolic program, the output is redefined as a function of some monotonically increasing variable. A network of processes may then be arranged such that time is substituted for the monotonically increasing variable, with input data fed into the system undergoing some operation as it passes (synchronously) through each node.[1] A popular example of a systolic algorithm is two-dimensional array multiplication, where a function of the array subscripts is used as the 'time' variable. Work which has been undertaken to better the accessibility of the orchestra definition procedure for those composers less experienced in computer programming would also lead in the direction of a distributed program paradigm. Many programs exist which enable the graphic definition of the orchestra rather than

[1] meaning, in this context, that the number of output data produced for a given number of input data is known and constant

a textual one; amongst them Patchwork[31] (written in Lisp), the window-based editor environment of Decker *et al.*[32] and Tarabella's graphical synthesis algorithm editor[33] for a system based upon a single digital signal processor.

Whilst there is clearly a *prima facie* case for adopting a systolic approach, there are also considerable drawbacks. The unit generators in the CSOUND algorithm operate in non-real-time; not being constrained by a hard deadline by which a sample must be produced, they may be arbitrarily complex. The required processing time may be difficult to estimate, or perhaps even impossible to estimate without prior knowledge of the data presented at their control inputs. Indeed, Sedgewick[34] asserts that:

> Given a deadline and a set of tasks of varying length to be performed on two identical processors, can the tasks be arranged so that the deadline can be met?

is an NP-complete [2] problem. Heuristic solutions have been attempted, but they have usually demonstrated encouraging results only for smaller systems.[35] Because the problem is NP-complete, as the orchestras increase in complexity, the point will soon be reached when determination of the optimum schedule takes longer than the execution of the score by a single processor!

An alternative solution, and in fact the one that was chosen, acknowledges the program's data structure rather than its control structure. The output of the program consists of a series of time-samples representing an audio signal. These audio signals arise as a result of various orchestral instruments being invoked to produce data by the commands contained within the score. We further observe that most musical examples contain many score events, and that it is normal for the final output to be the summation of the individual, concurrently sounding notes. An appropriate data-structure-driven multi-processor implementation uses a pipeline of processors, subdividing the control data rather than the control structure. Flynn[36] classified the pipeline structure as 'M.I.S.D.' (Multiple

[2] Nondeterministic-Polynomial-Complete: that class of problems for which the only known method of finding a best solution is to test all possible solutions

Instruction, Single Datum), because a single data-stream arises from the action of multiple instruction streams. It is felt that this is possibly a misleading label for the application currently under consideration; although the output data is indeed the result of the summation of the results of diverse instruction streams, it is interpreted at the output as a super-position of many data rather than a single datum. It is certainly the case that the pipeline is an isonomic structure, where many copies of an identical program operate concurrently with differing control data to produce the output; this is the preferred term in this instance.

Bowler[37] has demonstrated that an isonomic software architecture performs well in the construction of a real-time additive synthesiser; this solution is very similar to the proposed structure for CSOUND in that the final signal is composed of the summation of sinusoidal oscillators each operating with different control data. Gould[38] has also shown that a systolic approach may be reduced in efficiency if the communication and processing capacity are ill-matched; in some cases, the addition of an extra processor has been demonstrated to *reduce* drastically the throughput (on a per-processor basis) in a digital signal processing application. An isonomic structure was therefore preferred for the CSOUND implementation, as it offers two fundamental advantages: the communications bandwidth required to pass samples between processors does not increase as further processors are added; and that processing power to control data flow increases linearly with the overhead.

2.2 Practical Multi-processing

Whatever parallel strategy were chosen for the multi-processor CSOUND, it would be necessary to simplify as far as possible the communications between the subtasks running in the network, and the host computer which is responsible for screen/keyboard and mass storage I/O. The M.I.T. modules were therefore modified so as to exclude any explicit reference to host services: calls to `open()`, `printf()`, `scanf()`, `write()`, etc. are now placed in a single module called

RSUPER.C (Root Supervisor). Separate modules are also required to interface between RSUPER, and the sound, score and orchestra I/O routines within the CSOUND program. CSOUND runs as a thread separate from the root supervisor, although both share the same task to ease the passing of user command line parameters. The Communicating Sequential Process model provides for inter-thread communication *via* 'channels'. A channel is a shared variable which enforces synchronisation when accessed. A reading thread is automatically suspended until a writing thread places data in the channel; this data structure is supported in transputer hardware. Communication between the main CSOUND thread and the supervisor thread uses such channels.

When this strategy was decided upon for the sake of ease of maintenance and code partitioning, it was expected that a sacrifice in absolute processing speed would be required. It is interesting to note that, in fact, the opposite is the case, and that the supervisor thread spends a good proportion of its time suspended awaiting the lumbering machinations of the host and its associated mass storage. Because the entire data flow between the application program and the host computer's file system is now forced to pass through this additional supervisor thread, it is possible to insert a buffering program very easily. The cost of a context switch on a transputer is very small indeed; CSOUND is therefore able to spend usefully the time previously used in waiting for the file I/O functions to return, by beginning to calculate the next buffer-full of samples. Hence the multi-thread version actually executes up to 5% faster than the early single-thread development versions, despite considerable added complexity.

The hardware used for this implementation of the CSOUND package was determined in the first instance by the hardware available to the Music Technology group. Since a large proportion of the time spent in execution requires floating point operations to be carried out, it was decided that the use of T800 transputers was a necessity. The package would probably run with as little as 512KB of memory, although as explained later, increasing the degree of parallelisation

requires that more memory per node be fitted. Hence a T800 with 1MB running at 20MHz is used in the development system. Whilst faster transputers are available, there is some difficulty in exploiting their higher clock speed in large software systems which may not make best use of the on-chip memory, unless expensive high-speed RAM is also provided.

2.3 Workload Allocation Strategies

Isonomic concurrency allows processor load to be allocated by a modified version of the score sorting program. An optimum load-allocator requires knowledge of the orchestra definition, as well as good estimates of the computational cost of each orchestra command. Execution profiling tools are not currently available, so two sub-optimum methods were tested: round robin and balanced job queue.[39] The former consists of allocation of note initialisation commands contained within the score on a revolving basis in order of start time. The latter is based on the assumption that all orchestra commands have equal computational expense, and then proceeds as follows:

1. The score is sorted according to the normal sorting rules.
2. Each processor is allocated an empty list of jobs.
3. For each note in the score:
   (a) Delete all jobs in all job lists with an end time prior to the current note's start time
   (b) *Either:*
      i. Allocate the current note to the processor associated with the shortest job list, *or:*
      ii. Mark the event as 'global interest' for broadcast to all processors

Table 4 is the parallel-sort version of the score example presented previously, configured for three processors.

```
!w 0 60
!f 1 0 0 32 32 10 1
!f 2 0 0 512 512 10 3 7 6 2 1
!f 3 0 0 8 8 -2 1 1.4 0.9 1.1 0.5 0.7 1.5 0.8
#1#i 1 0 0 6 6 5000 440 0 0.8 25
!i 3 0 0 -1 -1
#2#i 2 .8 .8 3 3 5000 1700 0 0.8 25
#0#i 1 1.8 1.8 7 7 5000 875 0 1.7 15
#1#i 2 2.1 2.1 6 6 5000 320 0 1.4 15
#2#i 1 3 3 6 6 5000 1200 0 0.8 35
#0#i 2 3.9 3.9 4.5 4.5 5000 650 0 1.4 35
#2#i 1 4.9 4.9 4 4 5000 575 0 0.5 27
#1#i 2 6.1 6.1 4 4 5000 940 0 0.8 40
#2#i 1 7.3 7.3 3.4 3.4 5000 270 0 0.5 45
!f 0 11.5 11.5
!e
```

Table 4: A Parallel-sorted Score

A line of score needs to be broadcast to the entire pipeline if it does not contain note information; for example, lines beginning with the letter `f` are used to invoke function generators which are responsible for building data structures within the CSOUND program. As each copy of the program must be capable of executing any of the following lines of the score, it follows that each of the CSOUND processes must generate its internal data structure before synthesis can proceed. It is also assumed that any instrument which is set to run indefinitely is to be a broadcast line in the score. Such instruments are usually designed to perform some sort of post-processing of the synthesised signal, and are therefore to be instantiated on all of the processors.

It may be seen that the sort program has correctly identified the statements that are of global interest and must be received by each copy of the CSOUND program; these lines have an exclamation mark prefixed. The configurer assumes that all events with a negative ("forever") duration are global; this permits the correct operation of the reverberation instrument initialised by the `i 3` statement. For other events, the destination processor number is delimited by hash

signs. The above example actually reduces to a round-robin allocation, and it seems that, as a general rule, so do a large part of many real score examples.[3] It is quite possible that some processors in a multi-processor system using this type of task allocation may, from time to time, be producing silence or all-zero samples. In fact, this is not as inefficient as it may seem: CSOUND is very efficient at the production of zero samples, there being a special routine provided in the SOUNDIO module for this purpose. Little time is lost in skipping between periods of sample generation provided there is enough RAM at each node to buffer the periods of relative inactivity while other processors 'catch up'; this explains the paradox concerning memory requirement and the number of processors put forward above. The fact that an unbalanced workload leads to a waste of processor time was taken into account in writing the task allocation algorithm: when all processors have an equal number of active instruments, the allocator resorts to a round-robin approach in recognition of the desirability of a balanced computational load. It is certainly the case that the balance of load between processors is improved if the score contains many overlapping notes of short duration.

2.4 Benchmarks Against Existing Implementations

The first version of CSOUND to be tested and working on a transputer system was run on a 15MHz T400 with 1MB RAM. This was being compared with similar code running on an Atari ST as part of the Composers' Desktop Project (C.D.P.)[28] — a 16-bit 68000 running at 8MHz — and with a PDP11/23 with floating point processor previously used for teaching students of electroacoustic composition within the School of Music at the University of Durham. The PDP version of the program, MUSIC11, is coded in assembler. Results of these benchmarks, and results of the same score compiled on a 20MHz T800 are set out in

[3] A Mendelssohn Organ Sonata and one of Elgar's Enigma Variations also have this characteristic, being two extended 60-second score segments coded by users.

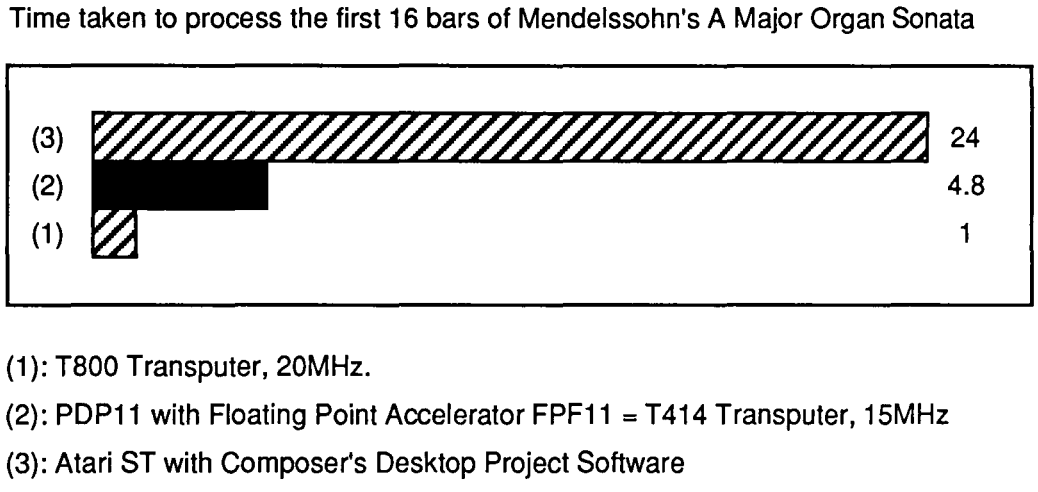


Figure 2: Relative Performance of Differing Implementations

figure 2.

The fact that the main CSOUND program uses floating-point operations internally is responsible for the large difference in performance between the floating point (T800) Transputer, and the fixed point (T400) Transputer using floating point library routines. What is more encouraging is the increase in speed of the T400 compared with the 68000-based system; although the memory bandwidth and clock speed are both approximately doubled, the Transputer system compiled the score to produce an audio output file almost exactly five times as quickly as the 68000. It would seem that, at least for the single processor case, the implementation is efficient.

We now continue to consider the finer details of a fully multi-processor version of this program, the efficiency of adding extra processors to the system, and the impact upon programming both at the user and system levels.

Chapter 3

A Processor Pipeline Synthesiser

3.1 Introduction

THE PREVIOUS CHAPTER HAS SHOWN that the speed of execution of CSOUND can be considerably enhanced by running the program on an INMOS transputer, simply by virtue of this processor's proficiency in general purpose, floating point calculation. Using only one T800 floating point, 20MHz transputer, the speed of execution is some 24 times as great as a desktop P.C. based on an 8MHz 68000. An approximate load-balancing algorithm for mapping standard scores onto multi-processor arrays has also been presented, as has a suitable topology which avoids communications bottlenecks to a large extent for many digital signal processing algorithms.

This chapter falls into three sections: firstly, the detailed structure of the additional program modules required for genuinely parallel execution is examined, and their protocols and data structures explained; secondly, the I/O library available to the user in the writing of new modules for execution in this environment is documented; thirdly, extra constraints placed upon the user by the new environment are put forward. Consideration is given to advanced techniques available to a user in the writing of score and orchestra files for a multi-processor network, which should enable the best possible performance to be obtained.

Additional material relating to the concurrent version of this software is included as an appendix. This should enable the extension of the system to be as easy as for the original CSOUND program, but avoids the technical minutiae

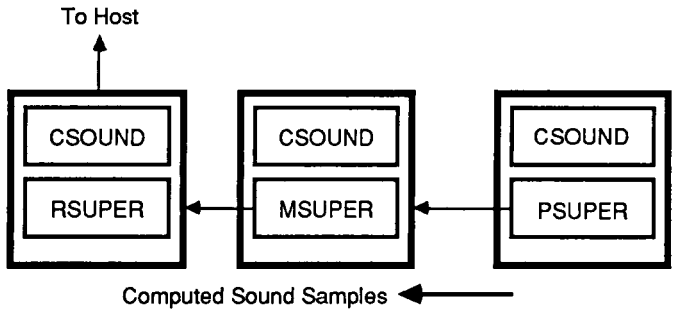


Figure 3: Processor Pipeline Architecture

obscuring the relevant direction of the argument.

3.2 Language Extensions for Multiple Processors

The processor network architecture on which the parallel version of CSOUND was implemented is shown in figure 3. This requires that the CSOUND source code be modified in order to take into account its isolation from the host machine, and also that the three different supervisor programs be written: RSUPER, residing upon the root transputer, supervises file I/O and other communication with the host machine's file system and terminal; MSUPER, which is a 'pass through' module for data from more remote transputers, provides the CSOUND program with which it shares the processor pseudo-file and -console I/O facilities; and PSUPER, which is a special subset of the supervisor program found in the middle of the pipeline, coded so as not to require input from a subsequent processor. Addition of extra processors is simply a matter of inserting copies of the middle processor, each running the main CSOUND program and the MSUPER supervisor program.

The following brief review of terminology and notation includes a description

of the extensions to the C programming language intended to support concurrency. Execution of a concurrent system or program ('*task*') may be considered as the simultaneous execution of many sub-programs or *threads.* Each thread has its own private workspace, and shared access to global static variables and system memory. It may or may not have unique code; sometimes different threads may execute the same subroutine simultaneously, relying on the private nature of their local data. In this case, the subroutine in question is said to be *multi-threaded.* Communication between threads may be through shared memory, or through *channels.* A synchronisation capability is provided automatically when channel I/O begins, but an additional data structure, the *semaphore*[40, 41] is provided where no data need actually be passed. Semaphores appear as 'flags' upon which two atomic (i.e. indivisible) operations are provided: signal and wait.

Waiting on a semaphore or channel I/O causes negligible overhead in processing terms, as the latter is directly supported by hardware, and the former can be implemented efficiently in assembly code.[42] Both operations, normally associated with computationally expensive operating system features, are implemented in hardware on the transputer. However, semaphores are essentially a special shared memory attribute, and may therefore not be used for the synchronisation of different tasks. Tasks can communicate only through channels. Channels are unidirectional data paths which may connect different tasks on the same transputer, or connect tasks on adjacent transputers *via* the serial data links provided.

To avoid confusion as to the context of certain names, the following typographical rules apply where possible: names in Sans Serif font refer to conceptual division or classification such as source code modules; names written in `Typewriter font` refer to entities which actually exist within the CSOUND source code or one of the supervisor modules.

3.3 Inter-module Data Flow and Protocols

There are two major data-flow paths through the parallel CSOUND system: the dissemination of score and orchestra information along the pipeline, and the collection of the resulting sound samples. In the multi-processor environment, however, it is necessary to provide special communications modules to link the source of the data to its destination.

3.3.1 Order of Events

Before commencing a full explanation of the data paths in the parallel system, the order of events in the execution of a standard CSOUND program is recapitulated.

- The initial phase of the CSMAIN module reads the command line parameters (these are fully documented in the CSOUND reference manual) and saves them in global variables. These will be accessed later in order to determine information about the desired file format, orchestra file name and so on.
- The orchestra file is opened and read from the host file system. Its contents are converted into semi-compiled form.
- The sound output file is opened which will eventually contain the samples generated by CSOUND. A dummy header of length `SIZEOF_HEADER` is written onto the hard disk. Sound sample generation begins.
- Score information is read from the file `stdin`. Sound samples are generated and sent to the host file system.
- The end of the score file is detected. The output buffer is flushed and a `seek()` performed on the output file.
- A new header is written at the beginning of the file containing the correct information about file format, number of channels etc.. The output file is then closed.

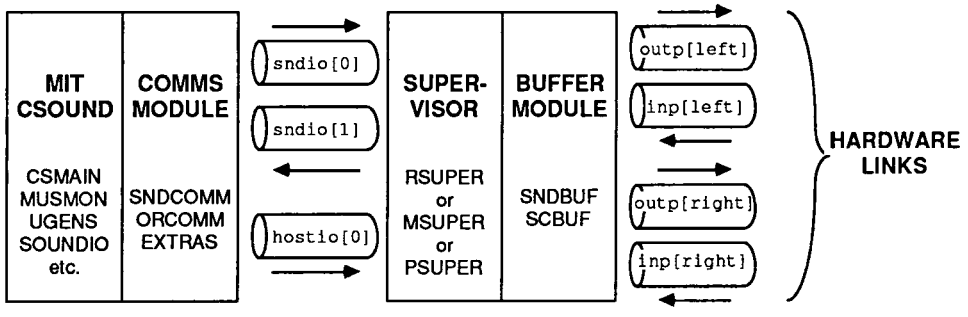


Figure 4: Internal Communication Channels

- Execution terminates.

Throughout the above operations, a constant dialogue is maintained with the host computer's terminal, *via* which the program advises the user of the progress of the sound compilation, run-time errors, or overflows that may have occurred.

3.3.2 Protocols in use during Initiation

A minimal protocol is provided for the broadcast of command-line parameters and orchestra data to the transputer network, since this process occurs exactly once and before any other communication can take place.

Data transfer between a supervisor and its associated CSOUND code is *via* three channels: `sndio[0]` and `sndio[1]` which are primarily intended for communication of data from or to the host file system, and `hostio[0]` which is used by data flowing towards the host machine's console (see figure 4).

At the very start of execution, the two threads `startsuper` and `main` begin to run. `main` is halted by waiting on the semaphore `args_valid` while the command line arguments are broadcast to the rest of the network. The commands protocol is:

$$\mathsf{word}\langle n\rangle{:}(\mathsf{asciiz})^n \rightarrow \mathtt{outp[right]}$$

where asciiz represents a string of zero or more ASCII characters terminated by a NULL. Having noted and re-broadcast the values of the command line

arguments, the supervisor routine raises the `args_valid` semaphore; execution of CSOUND proper then continues.

A similar procedure is followed for the broadcast of the orchestra. Before any sound samples can be produced, an identical copy of the orchestra must be loaded into each transputer. This transfer is achieved by the transmission down the pipeline of a sequence of words representing the data in the orchestra file, and terminated by an `EOF` (end of file) word. Just as with the command-line arguments, each transputer copies this information to its CSOUND thread and re-transmits it if it is not the last processor in the pipeline.

3.4 Score Demultiplexing

Conventional CSOUND expects score data to appear from `stdin`. In the parallel environment, however, there is no direct attachment of any of the CSOUND threads to the host file system, so the usual file system support calls to read or write information are not allowed. Additionally, the score data has been prefixed by the parallel sort program to indicate the destination of each line of data: the prefix is either an '`!`' which indicates that the line should be sent both to CSOUND and to the rest of the pipeline, or '#⟨d⟩#' where ⟨d⟩ represents the destination processor number — a non-negative integer.

The score demultiplexor has to run concurrently with the main code thread, so it is important that it makes efficient use of processor time. In practice, this implies that any waiting should be performed using either a semaphore or channel I/O calls, and that 'busy waiting', or polling, must be avoided. The demultiplexor must perform three functions: read input lines from the host and mark them according to their destination; send the necessary lines of the score to the rest of the pipeline; strip the prefix characters and send the raw score data to the CSOUND thread. In this transputer implementation, these functions are, broadly speaking, mapped onto three separate threads executing in parallel.

FIFO (First In, First Out) buffers for single thread environments are well

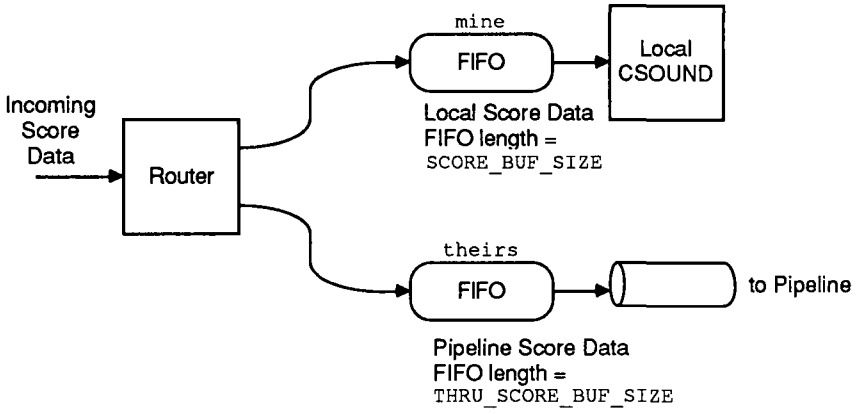


Figure 5: Queues in the Score Buffering Module

understood[43] and several optimised algorithms exist for their implementation, but the solution of the above problem requires that multiple threads share a common data structure. The use of FIFO buffering for both local and egressant data is highly advantageous; this avoids the processor waiting for score simply because pipeline communications are not available, and likewise avoids the starvation of the pipeline when the local CSOUND does not require any score data. Unfortunately, if a single FIFO buffer were used, whilst the resulting algorithm would be deadlock free because it is topologically free choice[1][44], it would be possible for either the pipeline or the local main thread to be suspended unnecessarily because of the order in which the data arrive. It is possible to avoid such inefficiency without the use of multiple FIFO queues by use of the 'Coloured Ticket' algorithm [45] but this introduces an (albeit minor) increase in processor overhead, and requires a non-trivial investment in writing the necessary code in transputer-C.

The solution arrived at for the score buffering module (figure 5) implements two autonomous FIFO queues: these are labelled `mine` for data destined for the local CSOUND thread, and `theirs` for data to be transmitted along the pipeline.

[1]For every possible data item appearing at its output, there is a transition (or event) which can (eventually) remove that item.

After the command line arguments and orchestra have been transmitted, the only data-flow away from the host machine is score data[2]; consequently no complex transport protocol is necessary and the score data is simply issued as 32-bit words along the hardware links. Further, the size of `theirs` need only be small in comparison with `mine`; exiting data will be read quickly by the next processor in the pipeline unless its score buffers are full. The sizes of the buffers in the current version are set at compile time to `SCORE_BUF_SIZE` = 32768 and `THRU_SCORE_BUF_SIZE` = 2048, which appears satisfactory.

Figure 6 shows a Petrinet[46, 47] representation of the concurrent score buffer algorithm, excluding the decision process concerning the destination of the characters as they arrive. This may be determined by a simple state machine which feeds the input places with the correct number of tokens, corresponding to the number of characters read, as each input line arrives.

It has been shown that a system of a number of asynchronous processes competing for a single resource require no additional arbitrating process;[48] the Petrinet clearly demonstrates the interaction between the three threads achieves this. The novel aspect of the control of the FIFO buffers is that instead of relying upon the more usual pointer comparisons to determine the presence of data within the queue, this test is performed using their associated semaphores. Taking the pipeline output buffer as an example, the related semaphores would be `theirs_free`, `theirs_empty`, and `their_data`. The initial values of these semaphores are respectively 1 (True), `THRU_SCORE_BUF_SIZE`, and 0, indicating that the pipeline output buffer is available (i.e. not being accessed by another thread), has `THRU_SCORE_BUF_SIZE` free locations, and contains 0 items of data.

A thread writing to the queue first suspends until empty space is available by waiting on `theirs_empty`. It then locks access to the queue and its associated pointers by waiting for `theirs_free`. When permission has been granted, the character is placed at the head of the queue according to the traditional

[2]This places some restrictions on the functionality of the CSOUND package — these are covered in section 3.10.

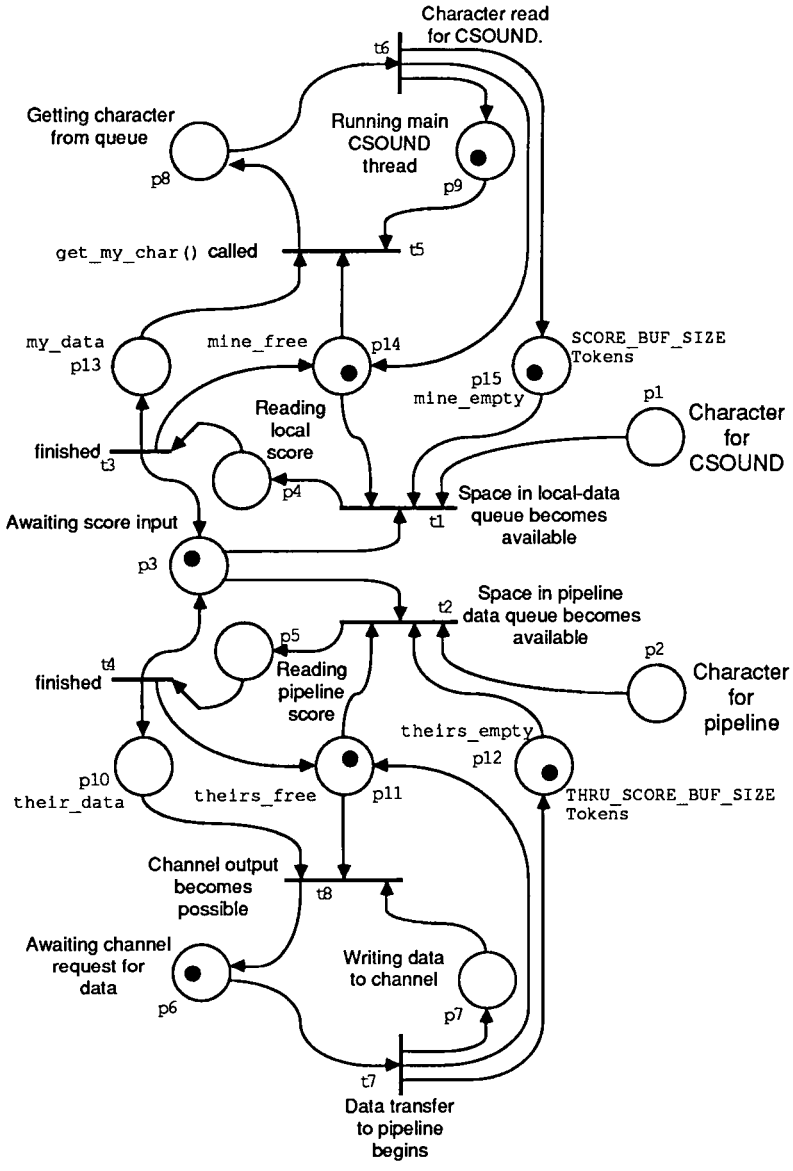


Figure 6: Petrinet of the Score Buffer Algorithm

algorithm, and its presence signalled on `their_data`. Finally, the queue is unlocked by signalling `theirs_free`. Conversely, a thread requiring input from the queue waits for `their_data` before locking the structure, removing a character, signalling on `theirs_empty` and finally unlocking it. This strategy would seem arduous in the extreme to those familiar with using semaphores implemented by operating system software, but since the I/O primitives in the transputer instruction set can be used to enforce synchronisation, the processing overhead which accompanies this algorithm is marginal. Also, there is a saving in that it is no longer necessary to test for buffer overflow explicitly; this test is inherent in waiting on the `theirs_empty` semaphore. Neither is any busy waiting required.

3.5 Sound Sample Recombination

Having demonstrated how each transputer in the pipeline receives a complete copy of the orchestra and its own unique subset of the score, all that is required to provide an understanding of the concurrent CSOUND system is an explanation of the linear sound sample recombination and their storage in the host's sound output file. Various simplifications which are valid in the case of the score demultiplexing algorithm are not applicable to data flow towards the host; in particular, the information flowing in this direction may serve a variety of different purposes, and a transport protocol is therefore required.

The transport methodology is a simple prefix-tagged protocol; each data block travelling towards the host machine is prefixed by a word defining its class (or meaning). The information received from the pipeline is processed by a separate thread `pipe()` in module PIPE.C. The protocol definition is as follows:

word`PIPE_ABORT` |	Finished communication
word(`PIPE_HOSTMSG`, ⟨origin⟩, ⟨size⟩) msg(size) |	Display msg on terminal
word`PIPE_SAMPLES` msg(`BUFSIZE`) |	Block of samples
word`PIPE_LASTBLOCK` |	Last samples; flush buffers

`wordPIPE_OVERFLOW msg(sizeof(float) value)` Remote arithmetic overflow

where `msg(n)` represents a string of exactly n characters (without a trailing NULL).

If the avoidance of starvation is important in the case of score routeing, then for the temporary storage of sound samples it is doubly so. A fundamental requirement for the significant increase in speed of computation is that processors which are less loaded for a given section of the score may 'rush ahead'. Such processors store their output data locally in a large FIFO queue until such time that data from the rest of the pipeline can be summed with it to produce the required output. Similarly, data emerging from the pipeline must, if space permits, be read and stored locally if the processor falls behind as the score becomes particularly computationally intensive; failure to do so would block any other messages from the pipeline, and may even result in a CSOUND `main` thread becoming suspended. Whilst the score sorting algorithm takes steps to distribute the workload evenly, there is inevitably some imbalance which must be absorbed by local buffering. The conceptual interrelation of the three executing threads, the buffer space, and a notional access permission arbitrator is shown in figure 7.

Because of the relatively large size of the data items involved (sound samples are computed in buffers of 8KB length in the current implementation), it is unacceptable to use two discrete buffers as for the score buffering example. Instead, FIFO queues are built from 8KB blocks of RAM using three lists. Because of the close analogy to the operation of a simple disk filing system, these 8KB blocks of RAM will be referred to as *sectors*. The data structure and its initial contents used for sound buffering is shown in figure 8. All operations on this structure are performed by routines in the module SNDBUF.C. As in the case of the score demultiplexor, there are three threads to support this algorithm: one reads data from the pipeline and enqueues them; one reads data from the local processor and enqueues them; a third dequeues a sector of data from each queue, performs a vector sum of their contents and sends the result towards the host.

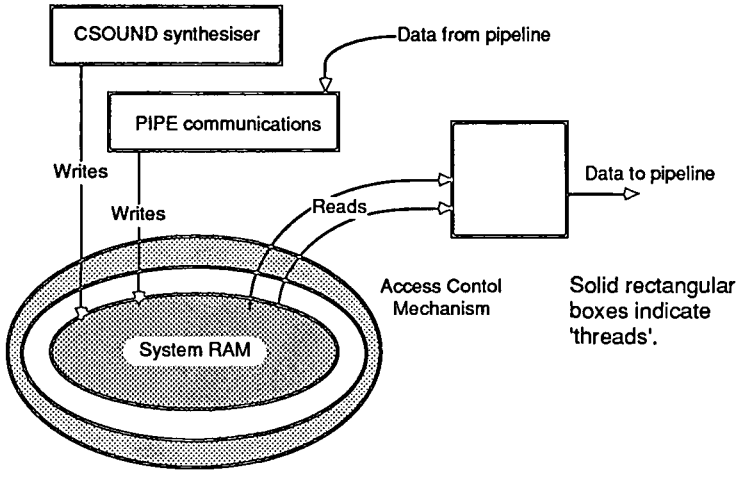


Figure 7: Interrelation of Threads in the Buffering Algorithm

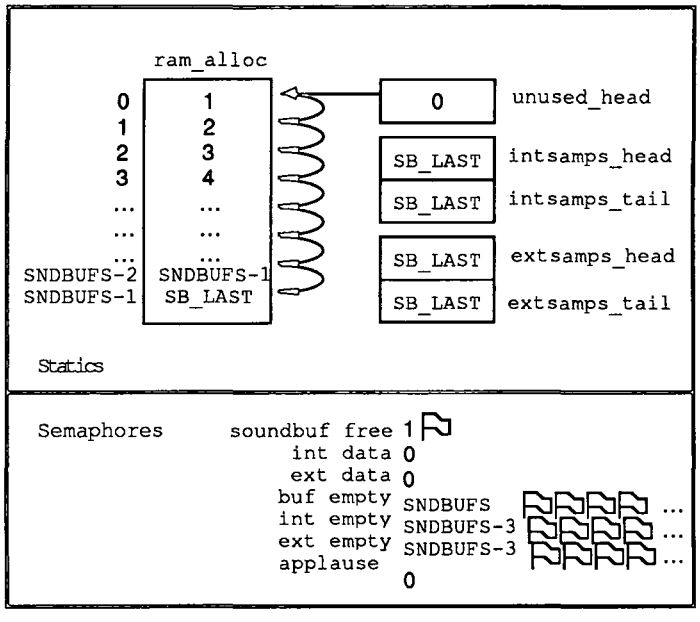


Figure 8: Data Structure of the Recombination Buffer

The word array `ram_alloc` contains three interleaved lists of integers representing the two FIFO queues and a LIFO (Last In First Out) stack within which a free sector list is maintained. A valid entry in this array must be either a non-negative integer pointing to the next element in the queue, or a termination word. A valid termination word is `SB_LAST` which normally signifies the end of the list, or `SB_FINISHED` which indicates that the marked element is not only at the end of the list, but that the thread which adds to that list has ceased the production of samples because it has reached the end of the score. The head of the stack is pointed to by the variable `unused_head`. Both FIFO queues require one variable to point to the beginning and one to the end of the queue: for internally generated samples, these are called `intsamps_head` and `intsamps_tail`; for incoming samples from the pipeline the variables are `extsamps_head` and `extsamps_tail`. Data present and space present conditions are flagged by semaphores following a similar naming convention to the score buffering module: `int_data` and `ext_data` flag the presence of sectors containing respectively internally or externally generated samples; `int_empty` and `ext_empty` are raised if space exists for more samples; `buf_empty` is raised if there are free sectors in the buffer. The entire structure is locked using the semaphore `soundbuf_free` to prevent interference between concurrent threads attempting simultaneous access. All of the semaphores are shown in the Petrinet description of the sound buffering module in figure 9.

3.6 Deadlock Avoidance in the Sound Buffer

The access of a shared database by processes which may read or write information therein is a well documented problem[49], and whilst the writing threads in this instance should never be able to produce update anomalies (their write data spaces being disjoint), errors may occur if poorly synchronised threads were to update the control structure. The RAM allocation table is a relatively simple multiple threaded list, so updating it requires little computational effort and hence takes little time; a simple mutual exclusion flag (`soundbuf_free`) proves

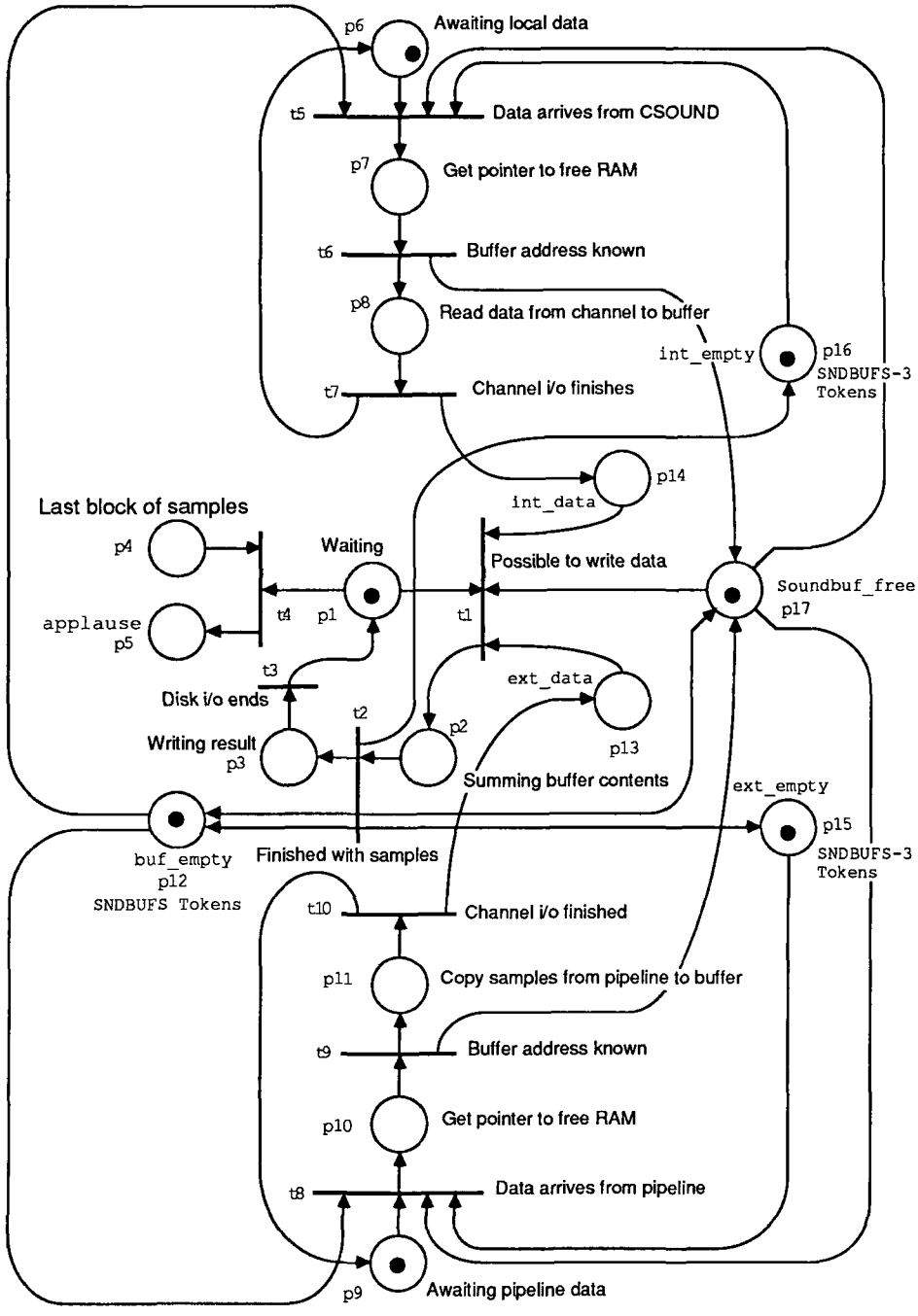


Figure 9: Petrinet of the Sound Buffering Algorithm

satisfactory. It is an advantage of this data structure that modification of the buffer space supplied from the list may occur at any time, since space is allocated to threads uniquely; the buffer need only be locked during RAM allocation and freeing. It is therefore most unlikely that threads will be suspended for long whilst awaiting access to a buffer.

There is a potential deadlock situation in the sound buffer module which cannot occur in the demultiplexor. Suppose the local processor has raced ahead of the processors further up the pipeline and filled all of available RAM. The sample output thread cannot output a sector of samples because it has no sectors from the pipeline in RAM with which it may perform its vector summation (`ext_data` is lowered). The thread which reads samples from the pipeline is suspended waiting for space in the external data queue to become available; this will never occur, however, because data from the pipeline must be written into a buffer before this can happen. This problem is similar to the 'dining philosophers problem';[40, 50] there are no more chopsticks, and each philosopher has exactly one chopstick and is therefore unable to eat anything. The solution implemented in SNDBUF.C, without resorting to polling, is to introduce two further semaphores; one for each writing process. For a process to be allowed to allocate a sector of RAM, is must have either `int_empty` (for local data) or `ext_empty` (for remote data) *and* `buf_empty` raised. Noting the initial values of the semaphores in figure 8, this effectively extends the preconditions for writing to "There is at least one empty sector in RAM and not more than `SNDBUFS` − 3 sectors are occupied by data from this source". This removes one of the necessary conditions for deadlocks to occur in a message passing system.[51]

After CSOUND has finished writing samples to the output file, a seek is performed so that an updated file header may be written. Clearly, it would be unacceptable to permit the seek operation to proceed as soon as the CSOUND thread had produced its final sample; there may be much information stored in the sound buffer awaiting output. If a file seek was performed immediately, this information may be written to the wrong place in the file, or the host machine

may simply crash.

The mechanism provided to avoid the premature termination of the CSOUND program is the `applause` semaphore. This semaphore is raised when the sound writing routine has attempted to get a further RAM sector from the format array, but received instead a `SB_FINISHED` token to indicate that the source routine has no intention of writing further data into the buffer. The supervisor file seek routine is called just before rewriting the file header; this awaits `applause` before returning the 'success' code to CSOUND.

3.7 Analysis of the Petrinets

Figures 6 & 9, the Petrinet models of the score and sound buffering modules, permit a further insight into how the three threads interact dynamically; in particular, they present concisely the resource sharing system. Beginning with the simpler score buffering module (figure 6), the initial marking yields the following interpretation: place p_{15} (`mine_empty`) has one token for each character of free space in the local score buffer; likewise, p_{12} (`theirs_empty`) has a token for each empty location in the pipeline forwarding buffer. The presence of a token in p_{14} or p_{11} represents the availability of the buffers and their associated pointers for access; these are the mutual exclusion semaphores. Tokens in p_3, p_6 and p_9 represent the program counters associated with the three buffering threads; their relocation represents execution of the associated code.

When an input character becomes available, the routing algorithm will make a decision about its destination and place a token in one of the input places p_1 or p_2. The data input thread may then proceed *via* the firing sequence (t_1, t_3) if the data is destined for the local CSOUND, or *via* (t_2, t_4) if the item is to be passed along the pipeline. Following the token around this route gives an indication of the conditions necessary for execution to proceed, by observing the labels associated with the transitions and their input places. The pipeline output thread circulates through transitions (t_7, t_8) as data is transmitted along the pipeline;

similarly, the CSOUND service routine `get_my_char`[3] enables the sequence (t_5, t_6). By following the execution tokens around these transitions, the movements of the memory resource tokens between the 'data' and 'empty' semaphores may also be observed. A required property of the memory resource tokens is that they are conserved; that is to say, the amount of memory which they represent is constant. Conservation with respect to storage is now demonstrated formally, using the matrix definition of the Petrinet.

Let the quadruple $C_1 = (P, T, D^+, D^-)$ define the Petrinet of figure 6, where $P = \{p_i\}$, the set of places,$T = \{t_i\}$, the set of transitions. D^+ and D^- are matrices defined by the forward and backward reachability functions as follows:

> Let $\#(p_i, I(t_j))$ be the multiplicity of place p_i with respect to the input function of transition t_j. This takes a non-negative integer value equal to the number of arcs connecting p_i to t_j. Also, let $\#(p_i, O(t_j))$, the multiplicity with respect to the output function, be the non-negative number of arcs between t_j and p_i. Define
>
> $$D^-_{j,i} = \#(p_i, I(t_j))$$
>
> and
>
> $$D^+_{j,i} = \#(p_i, O(t_j))$$
>
> Define also the *composite change matrix*
>
> $$D = D^+ - D^-.$$
>
> The Petrinet C_1 is conservative with respect to a weighting vector $\vec{w}$ if and only if
>
> $$D.\vec{w} = \vec{0}$$
>
> Element i of this vector, w_i, is referred to as the weight of place p_i.

[3] An additional character input function provided for use within parallel CSOUND, called when a supervisor module receives a `SF_KEY` token: see section 3.9 on page 46.

The appropriate weighting for each place in the network will now be considered. A token in `theirs_empty`, `their_data`, `my_data` or `mine_empty` is a direct representation of a memory location; hence assign $w_{10} = w_{12} = w_{13} = w_{15} = 1$. Data storage locations are also 'possessed' by tokens outside these places for the duration of the execution of the critical sections of code. These places must also be considered in the weighting vector, because the buffer space which they use is returned to the global buffer pool when the critical section of the code completes. Thus we also assign $w_4 = w_5 = w_6 = w_8 = 1$. D is a fairly sparse matrix; the non-zero elements are listed below:

$$D_{1,4} = D_{2,5} = D_{3,13} = D_{4,10} = D_{5,8} = D_{6,15} = D_{7,12} = D_{8,6} = 1$$
$$D_{1,15} = D_{2,12} = D_{3,4} = D_{4,5} = D_{5,13} = D_{6,8} = D_{7,6} = D_{8,10} = -1$$

Direct substitution into the equation of conservation yields:

$$D \cdot \vec{w} = \begin{pmatrix} D_{1,4} + D_{1,5} + D_{1,6} + D_{1,8} + D_{1,10} + D_{1,12} + D_{1,13} + D_{1,15} \\ D_{2,4} + D_{2,5} + D_{2,6} + D_{2,8} + D_{2,10} + D_{2,12} + D_{2,13} + D_{2,15} \\ \vdots \\ D_{8,4} + D_{8,5} + D_{8,6} + D_{8,8} + D_{8,10} + D_{8,12} + D_{8,13} + D_{8,15} \end{pmatrix}$$

Substitution of the values from the above lists shows directly that

$$D \cdot \vec{w} = \vec{0}$$

A similar procedure may be followed for the sound sample recombination buffers, although in this case there is the added complication imposed by the deadlock avoidance strategy described in section 3.6. A corollary of imposing this extra condition in order to avoid a deadlock in the shared memory system is that a RAM sector can not be described as either 'in use' or 'empty'; a third state is introduced where the RAM contains no useful data, but cannot be used by a particular thread because it is reserved for data from another source. In this case, the sector is said to be *meta-empty*.

The introduction of another state in describing a RAM sector makes the representation of a buffer element by a single token in the Petrinet far more difficult. The system used here is that a sector is represented by a pair of tokens; a token in place p_{12} (`buf_empty`) symbolises the meta-emptiness of a sector, but can not reflect emptiness unless combined with a token from one of places p_{15} (`ext_empty`) or p_{16} (`int_empty`). Thus a sector in use is represented by the circulation of a doubly heavy token.

The non-zero elements of the weighting vector $\vec{w}$ for the sound buffer may now be assigned.

- p_{12}, p_{15}, p_{16} have weighting 1 as described above.
- `int_data` and `ext_data` (p_{13}, p_{14}) contain tokens which represent data held in store and are therefore have a weighting of 2.
- The states of requesting a sector and reading data into the sector require an area of RAM to be both meta-empty and empty, so places p_7, p_8, p_{10}, p_{11} also have a weighting of 2.
- The action of summing the sound samples requires two full data blocks, so place p_2 has weighting 4.

Following a similar procedure to that used in the analysis of the score buffering module, non-zero elements in the sound buffer's composite change matrix are listed below.

$$D_{1,13} = D_{1,14} = D_{2,2} = D_{5,12} = D_{6,7} = D_{7,8} =$$
$$D_{8,12} = D_{8,15} = D_{9,10} = D_{10,11} = -1$$
$$D_{1,2} = D_{2,15} = D_{2,16} = D_{5,7} = D_{5,16} = D_{6,8} =$$
$$D_{7,14} = D_{8,10} = D_{9,11} = D_{10,13} = 1$$
$$D_{2,12} = 2$$

$$D \cdot \vec{w} = \begin{pmatrix} 4D_{1,2} + 2(D_{1,7} + D_{1,8} + D_{1,10} + D_{1,11} + D_{1,13} + \\ D_{1,14}) + D_{1,12} + D_{1,15} + D_{1,16} \\ \\ 4D_{2,2} + 2(D_{2,7} + D_{2,8} + D_{2,10} + D_{2,11} + D_{2,13} + \\ D_{2,14}) + D_{2,12} + D_{2,15} + D_{2,16} \\ \\ \vdots \\ \\ 4D_{10,2} + 2(D_{10,7} + D_{10,8} + D_{10,10} + D_{10,11} + \\ D_{10,13} + D_{10,14}) + D_{10,12} + D_{10,15} + D_{10,16} \end{pmatrix}$$

$$D \cdot \vec{w} = \vec{0}$$

$\Rightarrow$ the sound buffering algorithm is conservative with respect to memory resource.

As an addendum to this analysis of the multi-process multiplexing and demultiplexing buffer algorithms, it should be observed that in the case where there is only one reading thread and one writing thread, then an access-locking semaphore is not required. In this instance, the writing process and reading process can communicate using only the semaphores which indicate the number of spaces occupied and free in the buffer, which makes for a particularly elegant solution to a classical buffering problem in a concurrent environment. It may be pointed out that there is still a restriction imposed by the fact that the buffer array is used cyclically, which reduces the viability of a version of the algorithm using dynamic memory allocation, but this is of little consequence in transputer-based systems where each processor has a single user associated with it. In the single user case, the static allocation of uninitialised data-space in large quantities becomes less irksome.

3.8 The Supervisor Protocol

The previous paragraphs described in detail how information of all sorts is passed along the processor pipeline; the communication protocol between the CSOUND modules and the supervisor module remain to be explained. The whole of CSOUND has been modified to remove calls which assume direct connexion to the host machine, such as `fopen()`, `printf()`[4] and so on.

Communication with the supervisor is via three channels: `sndio[0]` carries data from CSOUND to the supervisor; `sndio[1]` carries data in the opposite direction; `hostio[0]` transfers data from CSOUND to the host's terminal display, and is a free-protocol character stream. The following is a summary of the supervisor protocol:

(wordSF_REQUEST ⇐ sndio[0]	Name output file
asciiz⟨output filename⟩ ⇐ sndio[0]	
word⟨file handle⟩ ⇒ sndio[1]) |	
(wordSF_WRITE ⇐ sndio[0]	Samples from CSOUND
word⟨file handle⟩ ⇐ sndio[0]	
word⟨block length⟩ ⇐ sndio[0]	
msg⟨block length⟩ ⇐ sndio[0]) |	
(wordSF_SEEK ⇐ sndio[0]	Unix-style seek
word⟨file handle⟩ ⇐ sndio[0]	
word⟨offset from start⟩ ⇐ sndio[0]	
word "1" ⇒ sndio[1]) |	
(wordSF_CLOSE ⇐ sndio[0]	Close a file
word⟨file handle⟩ ⇐ sndio[0]) |	
(wordSF_KEY ⇐ sndio[0]	Read a score character
word⟨score input character⟩ ⇒ sndio[1])	
wordSF_ABORT ⇐ sndio[0] |	Kill thread

[4]Software authors wishing to provide their own additional CSOUND modules which might use these calls are referred to section 3.9 where their replacements are described

(wordSF_OPEN† ⇐ sndio[0]	Open input file
asciiz⟨input filename⟩ ⇐ sndio[0]	
word⟨file handle⟩ ⇒ sndio[1]) |	
(wordSF_READ† ⇐ sndio[0]	Samples to CSOUND
word⟨file handle⟩ ⇐ sndio[0]	
word⟨length⟩ ⇐ sndio[0]	
word⟨length read⟩ ⇒ sndio[1]	
msg(length read) ⇒ sndio[1]) |	
char⟨display output character⟩ ⇐ hostio[0]	Display text

Facilities marked † are not available with multiple processors — see section 3.9.

3.9 The Programmers' New I/O Library

The previous section documents the hidden communications protocols and data structures within the supervisor modules; this section continues to describe the modifications of the standard CSOUND source, concentrating on the special I/O routines available to the writer of new software modules in the insular context of multiple processors.

Most modules of the conventional CSOUND package make reference to the normal C I/O library functions, and these must all be removed for the multi-processor version. Special interface calls to the low-level orchestra and score support routines provided by the supervisor modules simulate a subset of the UNIX-like I/O routines usually expected of C run-time libraries. These interface calls are to be found in the modules ORCOMM.C (orchestra communication), SNDCOMM.C (sound file communication) and EXTRAS.C (the rest). Probably of most interest to software writers are the functions provided by the SNDCOMM.C and EXTRAS.C modules.

Software authors wishing to make calls to the host file system in order to perform functions normally undertaken by the `fread()` and `fwrite()` calls can

use the equivalent functions provided by the SNDCOMM.C module. These are well documented in the source code, and broadly speaking have the same name as their familiar counterparts except that the letter 'f' is replaced by the prefix 'snd_'.

Those wishing to modify or introduce CSOUND source which directly reads the stdin stream will need to be aware of the routines which are used to replace the scanf() function call. The following functions are included in the module EXTRAS.C:

Functions which perform low-level reads in the standard input stream:

h_getchar()

No formal parameters.

Gets a character from the standard input stream.

Returns an integer.

h_getfp1(flp)

Formal Parameter: pointer to float.

Reads a floating point number into the indicated variable.

Returns 1 on conversion, 0 otherwise.

h_getfp2(flp1, flp2)

Formal Parameters: pointers to float.

Reads two floating point numbers.

Returns number of successful conversions: 0, 1 or 2.

The other functions which are provided are designed to enable output to the host screen and perform various housekeeping functions. The two most often used are to_host() and sxt(). sxt() sign extends a 16-bit number into an integer, and has to be provided because many of the M.I.T. routines unfortunately assume that integers are 16 bits in length instead of the 32-bit entities supported by transputers. to_host(l,p) provides a route by which messages can be displayed on the host's terminal. It requires two parameters: l is the message length in bytes, and p is a pointer to the first character of the message. Modules written

using this call will link and execute correctly on any of the transputers in the pipeline, as it ensures that the lower level data transport functions are called correctly. In fact, this call is invoked so often that a global workspace, `host_msg` has been provided. In order to display an arbitrary string, it is possible to use the fact that the library function `printf()` returns the number of characters converted and written into RAM; the following code fragment is repeated many times throughout the package:

`to_host(sprintf(host_msg,` *format string*`,` *args*`), host_msg);`

The low-level score and orchestra communication functions are not described here, as their use is rather limited outside the score reading and orchestra compilation modules supplied by M.I.T.. Readers who desperately need to modify these routines for some reason are referred to the source listings which contain the necessary comments. In the vast majority of cases, it will be sufficient to rely upon the data structures and global variables available to user routines from the existing source.

3.10 Differences between Standard and Parallel CSOUND

Whilst every effort has been made to ensure that the syntax and semantics of M.I.T.'s CSOUND program have been maintained, there are inevitably some differences between composing for a single processor and composing for multiple processors. The following sections highlight the differences, both technical and stylistic.

3.10.1 Restrictions with Multiple Processors

There are two main restrictions in composing for parallel CSOUND in its current state of development. These are:

- *No input from sound files already stored on disk.* The soundin facility is supported on the single processor version, but not with multiple processors. The reason is quite straightforward: if many processors were to attempt to run whilst accessing the host's file system concurrently, an I/O bottleneck would almost certainly occur. In fact, the process of continually performing random seeks on multiple input files, each of which is being multiply accessed, would so increase the complexity of the root supervisor and make so heavy a demand on the host's operating system that there would surely be a massive *decrease* in performance as more processors were added.

- *The output instrument must be linear;* that is to say, the output from instruments playing together must be identical to the summed output of each instrument playing separately. This requirement is imposed because the score is fragmented across different processors by the SORT program, and has not been found too great a disability in practice.

These restrictions do not apply when using a single processor.

3.10.2 Extended Command-line Options

Now that the sorting program is responsible for the automatic partitioning of the score and its allocation to different processors, the SORT command has been extended to include two optional command-line parameters:

SORT ⟨Score file⟩[⟨Processors⟩ [⟨Output file⟩]]

⟨Score file⟩ specifies the path of the input score file to be sorted, and must always be supplied; ⟨Processors⟩ specifies the number of transputers in the pipeline, and defaults to 1; ⟨Output file⟩ specifies the path of the file into which the sorted score is written. The default is ".\SCORE.SRT".

The score sort package reports how well the score has been allocated to the given processor array by generating an arbitrary number referred to as the "niceness" of the score. This concept was introduced as an aid to judging the effectiveness of the score sorting algorithm, but provided sufficient amusement to users to warrant its continued existence. It is calculated as follows. As each note is placed on a job list, the internal data structures of the sort program are scanned by a simple loop. The local nastiness is assessed as the difference in length between adjacent job queues. If any processor has no jobs, the nastiness is doubled, because in this situation large amounts of buffer space will be used to store silence. The accumulated nastiness is incremented by the local nastiness multiplied by the duration under consideration, and divided by the number of processing elements. Niceness is simply the total play time divided by the accumulated nastiness.

Some extra switches have also been added to the CSOUND package itself. The `-f` flag now produces integer output files, but uses floating point communication along the pipeline. This eliminates the effects of truncation to 16-bit integers at each node, which potentially reduces signal-to-noise ratio if a large number of processors are in use.

When using a large number of processors, the quantity of text displayed can quite quickly reach levels where the host machine spends sufficient time printing to slow down its disk operations quite significantly. The sheer quantity of text produced under these circumstances presents quite a formidable and rather confusing display, so parallel CSOUND has two extra command line switches available which help in these circumstances. The `-s` ('silence') switch suppresses messages from all processors except the root, and the `-S` switch suppresses all messages except for those giving reasons for termination

3.10.3 Extended Overflow Reporting

When the program is being used in the single processor version, the user is informed at regular intervals of the maximum amplitude of the output signal, and

the number of arithmetic overflows that have occurred. It is important that the parallel version is also able to provide some method of reporting overflows occurring in the output data, but here the problem is compounded by the fact that such errors may occur not only at the sound synthesis stage, but in the addition of sound samples as they travel down the pipeline of transputers towards the host. To this end, code was added to the supervisor modules which detects the overflows as they occur, and passes a token to this effect down the pipeline.

In practice, this information proved insufficient to the composer using the package; knowledge of the magnitude of the overflow is as important as its existence if appropriate corrective action is to be taken. If a score produces output which overflows the range of the 16-bit output device, but the magnitude of this overflow is not reported, the composer may be forced to compile the score twice more: once with very much reduced output level to assess the amount of attenuation required, and again to restore the proper dynamic range to the output file.

When data is passed along the pipeline as integer values, there is little that can be done to remedy the situation. Under these circumstances, when an overflow occurs, data is lost as the carry bit of the result is discarded. However, when the floating point operation of the pipeline is selected, the effect of the overflow does not become apparent until the final type conversion of the floating-point data to 15-bit integer just before delivery to the host. This enables the data following the overflow token to be read and used with meaning; the supervisor program stores the maximum absolute values between reporting overflows and displays them as part of the supervisor message. To prevent the remote possibility of an update anomaly occurring and loosing data contained in the variable `biggest`, the semaphore `big_lock` is provided.

Overflows at any stage of the pipeline are flagged in this way, so as to indicate problems which would occur in fixed point mode and provide a consistent environment which gives increased guidance in adjusting the score.

3.11 Better Programming for Effective Parallelism

It has been shown how CSOUND requires the composer to divide synthesis tasks into two distinct components: the score, and the orchestra. These entities are broadly analogous to the conventional ones: the latter provides basic definitions of each sound-producing module, or 'instrument'; the former its performance parameters. The scope to vary the content of these components as well as their functional relationship is considerable. At one extreme, it is possible to generate a complete musical gesture of considerable complexity from a single line of score; only the starting time and overall duration for an elaborate set of synthesis functions, all contained within one instrument, need be stated. At the other extreme, a comparable degree of activity can be described by a score which invokes multiple copies of much simpler instruments, each contributing to the overall effect.

The scheduling algorithm discussed above naturally favours the second approach to musical composition (complex score, simple instruments) if the full potential of the target multi-processor machine is to be realised. In this particular implementation of Parallel CSOUND, no facility exists for mapping individual components of an instrument across a network. In practice, the musical constraints arising from this technical restriction are not too limiting. The majority of composers seem to compose by building up individual layers of sound, rather than constructing highly complex and completely self-contained instruments from scratch. It therefore requires only a relatively minor change in approach to ensure that the resulting orchestral definitions are split up into a number of less complex instruments sounding concurrently, which may be assigned (automatically) across the computing network.

Two interrelated factors materially affect the working environment for the CSOUND composer: the first concerns the overall response of the system in terms of the typical time delay which intervenes between presenting and auditioning a

synthesis task; the second concerns the variation in this delay as a function of the task's complexity. In a conventional implementation of CSOUND the second characteristic is closely related to the complexity of the score, no matter what the speed of the processor in use. An increase in the number of voices in the score results in a corresponding extension of the execution time, as does increasing the number of algorithms within a single instrument. This is inevitable in a situation where all the parallel processes of musical composition have to be simulated by processing small segments of each component in turn and then accumulating the result.

Musically, such an environment is most unsatisfactory, and compares badly with traditional concepts of composition and performance. One would generally expect that the rehearsal time for a large orchestra should compare favourably with that required for a small ensemble, since every player is learning their part at the same time as every other player. Any steps which can be taken to reduce the fixed correlation between score complexity and processing time, as well as the overall response time, are thus to be welcomed. The concurrent implementation presented here achieves both objectives, and therefore encourages the composer to take full advantage of the powerful synthesis and signal processing algorithms offered by CSOUND. Indeed, simple orchestras are capable of 'Actual Speed' execution on a single fast floating-point transputer at the time of writing, and no doubt the speed of such devices will continue to increase. However, CSOUND was never intended as a real-time compositional tool, and some fundamental issues regarding its musical characteristics must be addressed if this is to be its ultimate goal. It may well be that the flexible development environment, which permits the user to add further synthesis operations without regard to the complexities of machine-dependent I/O, is the program's main benefit; particularly successful algorithms may be adopted for further optimisation, perhaps using dedicated hardware, so that they can be used in real-time performance.

3.12 Additional Score Commands

The method by which instrument initiations are distributed amongst the processors in the pipeline involves the use of balanced job queues, as described in section 2.3. It is necessary for the SORT program to make assumptions about which lines of score are of global interest, and therefore to be disseminated to all processors, and which are genuine instrument instantiations and therefore to be allocated a particular processor. It will be recalled that the rules currently in force are as follows: any line in a normally sorted score which does not begin with an 'i' is marked as global; all 'i' statements with negative duration parameters (CSOUND's "forever" marking) are also to be considered global; all remaining 'i' statements are to be assigned a particular processor according to the rules of the task allocator. This permits special 'instruments' which run for the whole duration of the score, such as reverberation units,[5] to be started on all processors automatically.

Composers will frequently write instrument specifications which cause the above algorithm to fail. The fairly limited control structures available to the electro-acoustic orchestrator demand the frequent use of global variables. A popular way of dynamically adjusting the values of such global variables within an orchestra is to write an instrument which produces no audio output, but simply performs such adjustment under the control of its parameter fields. This fails disastrously when executed on a parallel system. The SORT program recognises an 'i' statement with a non-negative duration, and accordingly assigns it to a particular processor; instead of affecting all copies of the global variables in all of the processors, only the orchestra of the (arbitrarily assigned) transputer experiences any change. This is particularly troublesome, as the resulting output file may sound almost correct, especially in a system with few transputers; one transputer will have indeed executed the score and orchestra as the composer

[5]it should be remembered that the term 'instrument' in this respect refers only to an algorithmic unit within the orchestra, and does not imply the generation of sound as in a conventional orchestra.

intended, while the others produced samples with constant values of orchestral variables.

Clearly, a mechanism was required which enabled the composer to force a particular 'i' statement to be executed on all of the transputers in the pipeline. This is achieved by the addition of the new 'g' command to the score sorting program. A command line beginning 'g' has exactly the same semantic meaning as one commencing with 'i', except that the resulting line in the sorted score file is an 'i' statement marked as global. If only one processor is specified for the sorting program, the 'g' character is simply replaced with 'i'. Judicious use of the 'g' statement with instruments which rely upon the modification of the orchestra's global variables ensure that a score will be portable between one- and many-transputer systems.

3.13 Conclusion

This chapter studied the features and structure of concurrent CSOUND in depth. Methods of running the standard sequential program in parallel form were discussed, and the data structures of the selected method presented in detail. The multi-thread algorithms for buffer/multiplexing have been examined formally using algebraic and pictorial representations. The differences arising from using the program in an isonomically concurrent environment have been listed, and their impacts on the user established.

In the next chapter, the increase in speed is measured for two or three processors running a standard score and orchestra, and an estimated profile demonstrates how possible future work may further enhance the system.

Chapter 4

Performance Evaluation of an Isonomic System

4.1 Performance of a Multi-processor Systems

AN IMPORTANT MEASURE OF THE UTILITY and performance of a computer which executes programs concurrently on more than one processor is the degree by which the program is scaleable. Using the optimum algorithm becomes less important than the ability to run the program efficiently where there is a larger number of processors, because even if some sacrifice in terms of absolute speed is made initially, an *effective* concurrent solution (assuming one exists) can always be made to perform better than the optimum single processor case. An aid to judging the effectiveness of the concurrent solution compared with the sequential one is the *speed-up factor* or *performance index*.

The performance index of a parallel computing system is calculated by dividing the speed-up relative to a single processor system by the number of processors in the network. A good parallel system will have a performance index close to 1, so that doubling the number of processors very nearly doubles the speed of execution. In fact, this index may itself be a variable; if a program requires that much computationally intensive work be done and the results written to a mass-storage device, as is the case with CSOUND, the performance index for a carefully written algorithm may be close to 1 for a small number of processors, but may fall away as the available computational power becomes so large that the fixed overhead

involved in writing to the mass-storage device becomes significant. In such a parallel, non-real-time system, the von Neumann bottleneck has been removed from the processor-memory interface and placed at the mass-storage device.

Other authors have addressed the problem of the most appropriate scheduling techniques, but most have concentrated on real-time systems which present slightly different problems. Holm uses a similar scheduling algorithm to that used by the SORT program, where events are placed in "Fundamental Clock Pulse Lists".[52] MIDI 'operating systems' which support multi-tasking and pipelining from a common controller have also been written.[53] Of greater significance to the CSOUND scheduling algorithm are the results of W.F.Walker[35] which suggest that the scheduler of this type may saturate for large numbers of processors, especially in a real-time context.

The analysis presented here assumes that the processing time is broken into three elements: *useful computation*, while the processor is performing calculations which are directly involved in the production of sound samples; *necessary computation*, while the processor is busy performing buffering or communication subroutines which do not in themselves contribute to synthesis; and *suspension*, an 'all else fails' state during which the processor awaits host I/O with insufficient buffer space to continue calculation. By manipulating the program source code and input data, information can be gained as to the relative time spent in each of these states.

Two scores and orchestras were used to benchmark the system. Programme 1[1] is an example with many short events in the score and an orchestra which is computationally inexpensive; programme 2[2] has fewer events in the score, but has an orchestra which demands a great deal of computation.

[1]Extract from Elgar's "Enigma Variations" performed on a simple pipe-organ model
[2]Extract from J.S.Bach's "Die Kunst der Fuge" performed on a complex FOF model

Test	Description	Time/Sec.
(1)	Normal Execution	497
(2)	No Output from Program	332
(3)	No Output to Disk	377
(4)	Sparse Score	93

Table 5: Experiment 1 Execution Times.

4.2 Execution Profiling

The first experiment attempts to estimate how much time is spent in I/O overhead, and how much is spent in useful computation. Programme 1 was compiled by three processors in four different contexts:

1. Normal execution – sound output file produced on disk;

2. Output suppressed with the -n (no output file) option in the CSOUND command line;

3. Normal execution with patched code in SNDBUF.C – all of the buffering and data transfer takes place, but the final fwrite() statement is not executed so that the data is discarded;

4. As 3, but the score is sorted for ten processors and all score lines referring to processors which were not fitted removed.

The execution times are shown in Table 5.

By far the longest execution time is taken by test (1), indicating that, even for a system expanded to only three Transputers, the delay incurred in writing to the host's file system is already significant. When no output is produced by the program, as in test (2), the execution completes in 67% of the time taken when writing the sound-file to the disk. However, this does not necessarily mean that the whole of this delay is introduced by the host machine and the server program which performs the disk access; when CSOUND is invoked with the -n

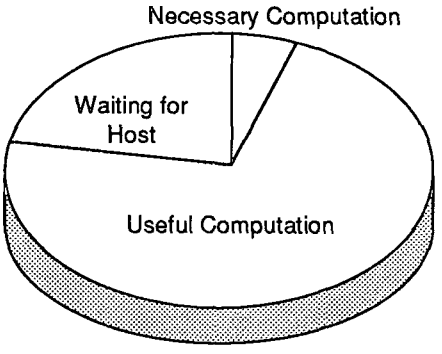


Figure 10: Profile of CSOUND Execution

command line option, the sound sample buffering and recombination code will never be used. The wasted 33% will be spent not just in a suspended state, but also in necessary computation.

An estimate of the time spent in necessary computation can be obtained from test (3). In this case, the main program is instructed to produce sound output, but the sound output buffering module on the root processor is modified. Instead of the output data being written to disk, the samples are discarded at the last moment. All of the buffering and communication overhead is thus retained, but no extra delay is incurred by waiting for mass-storage to be ready. From this benchmark, we deduce that of total computation time, 88% is spent in useful computation and 12% in necessary computation. Put another way, of the total time spent in the compilation of this example: 24% is spent suspended pending mass-storage operations; 67% is spent in useful computation; only 9% is spent in necessary computation. Figure 10 presents these results graphically. This is an encouraging indication of the ability of the Transputer to perform communication operations between concurrent processes without great impact on the general processor throughput.

Since resources did not permit a test of the parallel program on more than three processors, a simulation was devised to estimate the behaviour of a larger system. The score is sorted using the standard utility, but specifying a pipeline

of ten processors instead of three. The sorted score is then passed through a filter program, which removes those lines marked to be routed to non-existent processors. This produces a score with notes allocated sparsely amongst the processors (test (4)), which is what one would expect in a large system where the number of Transputers greatly exceeds the number of concurrently sounding notes. If there is sufficient buffering to allow each processor to 'run ahead' freely, the execution should take approximately 0.3 times as long on the ten-processor simulation as it does running in its entirety on three processors (writing the sample data to disk is suppressed as for test (c) in order to avoid introducing mass-storage delays). In fact, table 5 shows that the execution time is cut to 25% of the original — faster than expected. The additional speed-up results from the reduced overhead in managing concurrently sounding instruments on a single processor, as searching instrument instantiation tables, memory allocation overheads, etc., occur less frequently if a smaller number of currently active instruments are present. This is an example of a system which exhibits 'superunitary speedup' as modelled by Helmbold & McDowell.[54]

4.3 Buffer usage

Having produced an estimated execution profile of the parallel CSOUND package, memory usage must also be considered. The prototype three-processor system has 1MB of memory at each node. Of this, 256KB is allocated to the sound sample recombination buffers. The quantity of memory available for this purpose is defined by the manifest constant SNDBUFS in file BUF.H. The system normally operates with 32 sound buffers, each 8KB in length.

A common failing of parallel programs which rely for their speed increase on buffering between processes is that as more processors are added, the memory requirement for buffering at each node also increases. In such a situation, the parallel program cannot be extended beyond a limited number of processors,

SNDBUFS	RAM Allocated	Time/Min.
4	32KB	11.98
8	64KB	11.50
16	128KB	10.35
32	256KB	9.43

Table 6: Experiment 2 Execution Times

because to do so would require an excessive amount of buffering. The second experiment investigates the effect of variation of the number of sound buffers upon the compilation time of Programme 2. Programme 2 was chosen because it contains fairly sparsely distributed score events, and each event is computationally expensive. It is this combination which places the greatest stress on the buffering mechanism. Table 6 shows the execution times for Programme 2 compiling with three processors and various buffer sizes.

There is a 27% increase in compilation time with 4 sound buffers over the time required with 32 buffers. This is a significant change, although it is quite favourable compared with the performance of some algorithms mapped *in corporæ tota* into a parallel environment. Increasing the number of buffers to 48 only marginally increases performance, indicating that even this awkward example has begun to converge at this level. This underlines the need for composers to generate dense scores at the expense of orchestral complexity if the full potential of the system is to be realised. To this end, it is encouraged that a set of score generating tools be made available, which simplify the generation of scores with a large number of simpler instruments.

4.4 Measured Performance Indices

As a final experiment, Programme 2 is compiled (and the output written to disk) using one, two and three processors, and the performance index calculated. The results are set out in table 7.

Processors	Performance Index
1	=1.00
2	0.90
3	0.89

Table 7: Performance Indices

This is a very useful parameter, especially as it is an overall performance estimation in a real synthesis situation. The single-processor version of the program runs with the greatest efficiency; this is as would be expected, as it has a far lower communications overhead than the pipeline versions. The addition of an extra processor (with the attendant communications software) reduces the performance index to 0.90, but the addition of a third processor has a negligible effect upon the performance index. It is this stability in the performance index with the addition of an extra processor which indicates a successful mapping onto a limited parallel environment.

4.5 Conclusions

The CSOUND program, originally developed by Barry Vercoe at M.I.T., provides a flexible environment for the composition of electro-acoustic music and the investigation of novel synthesis algorithms. The penalty for this level of flexibility is slow execution of the program, the execution times of complex algorithms on personal computers being unacceptably protracted.

The use of a pipeline architecture with distribution of sound generation tasks followed by signal superposition has been shown to be an efficient approach to real-time synthesis. This architecture is suitable for CSOUND provided that acceptable routeing and allocation algorithms are available, and that the communications protocols do not place undue extra processing load on the chosen processors.

Inmos Transputers perform particularly well in a multi-processor machine,

even when the software package is written in a language with poor support for concurrent processing. A speed increase approaching a factor of 30 is achieved when using one such processor, compared with a conventional 16-bit microprocessor, by virtue of the built-in floating-point hardware. This increases to over 80 times speed increase using three processors. Scores previously taking hours to compile now complete in as many minutes.

Whilst there are a few extra restrictions placed upon the authors of new synthesis modules, these are easily understood and do not generally effect the algorithms directly. The time taken to transport the FOF unit generator module from the conventional environment to the parallel one was tens of minutes for an experienced C programmer.

Most scores and orchestras which perform correctly on one processor are executable without modification in parallel. Certain advanced instruments require the use of special score commands, but these are now supported by the single processor package ensuring that all new scores may be executed in parallel after a system expansion. There is every indication that significant increases in performance are achieved with the addition of further processing elements.

This concludes the section concerning the re-writing of an existing, sequential piece of software for an isonomic array. The next part moves on to describe a different problem, for which an isonomic architecture does not provide an efficient solution.

Chapter 5

Sound Modification through Signal Processing

5.1 Introduction

THE SECOND PART OF THIS THESIS deals with the problem of applying arbitrary manipulation to real sounds. This differs fundamentally from the action of synthesis, as the quantity of input and output data is broadly similar. Because the kinds of sound manipulation which are found interesting by composers tend to require a large amount of computation, they are usually performed outside real-time, using files previously recorded on some mass storage medium. A result of increasing the sound I/O rate to and from the data storage device, whilst the quantity of control information remains unchanged, is that far heavier demands are made of the file-server node of the processor network.

It is a further property of the signal modification software that it is rarely suitable for isonomic decomposition. The difference between a synthesis and a sound manipulation program is that, in the former case, the control data structure specifies discrete, autonomous events explicitly. Whilst algorithms such as digital filters may be coded systolically after mathematical simplification, they are, in essence, state machines; their output depends upon their current input *and upon a memory of their previous input.* It is impossible to guarantee the present state of an infinite impulse response filter without evaluating its response up until the current time for all of the input data presented so far. A time-division, isonomic

decomposition (as was used in the case of CSOUND) of such a problem is therefore highly inappropriate, as each of the processors would need to perform all of the operations from the beginning of the input stream before the correct initial state could be established to continue the evaluation of a particular time-slice of the output signal.

A popular vehicle for the manipulation of recorded sounds outside of real-time is the *phase vocoder*.[55] Since the final part of this work considers the use of this program in a more novel application, it seems appropriate to delay a more detailed examination of its development and operation until then. This brief, second section contains sufficient introduction only to permit a discussion of the concurrent program's mechanics.

The object of the phase vocoder is to present the electroacoustic musician with the input signal in a form which is more intuitively useful and relevant for the purpose of manipulation than the a series of time-samples. This is achieved by taking overlapping Fourier transformations of the input signal to produce a 'spectrum' represented as a set of points on the reciprocal-space complex plane. After conversion to phase and amplitude representation, the frequency of each of the spectral components is determined by a differentiation of the phase data against the stored previous frame. If the assumption is made that only a single spectral component of the input signal is represented by each complex Fourier transformation result, this technique allows the frequency of that component to be determined to a resolution in excess of the bandwidth of the Fourier 'bin'. The spectrum may then be modified by the user in an arbitrary fashion, before the inverse operation is undertaken, followed by an overlap-and-add process[56] to yield a modified time signal. Typical transformations possible using a non-real-time phase vocoder might be to change the pitch of a sound without changing its duration, or *vice versa*, or to perform a timbral interpolation between two differing sounds.[57] Pitch changing and time stretching have been demonstrated by Lent[58] using computationally more efficient algorithms. His solution uses

windowing to reduce harmonic aberration and an adaptive 4th order I.I.R. pitch-tracking filter to retain the formant of the signal though the transformation. For more ambitious effects, however, the fidelity and flexibility offered by the phase vocoder are widely regarded as unrivalled.

It is the requirement to maintain the current state of the stored previous phases of both input and output signal which makes the phase vocoder program inherently unsuitable for isonomic implementation. This is not because of some algorithmic optimisation (as is the case with, for example, minimum cross-entropy analysis[59]), where several potentially parallel processes have been combined in order to increase the efficiency of execution on a sequential machine, but because of an intrinsic property of the problem: the calculation of the phase change between analysis windows requires the storage of the phase information from the previous analysis. This imposition of execution order denies concurrency.

5.2 An Alternative Concurrent Strategy

The most demanding part of the phase vocoder process, computationally, is undoubtedly the Fourier transformation. Typically, the user requests 512 or 1024 channel transformations, each window being separated by about half that number of samples. The forward transformation is followed by a rectangular to polar conversion, involving two products, a sum, a square root, and an arctangent operation on each pair of data from the Fourier transformer. The inverse transformation is preceded by a polar to rectangular conversion of each data couplet, requiring two products, a cosine and a sine operation.

Clearly, the data structure of the problem leaving no room for parallelisation, it is the program structure which must be addressed. A systolic approach could be feasible, as such algorithms for the implementation of the Fourier transformation are readily available. However, the data-flow through the system is not strictly defined at compile-time, instead being determined by the command line parameters supplied by the user. For example, if a pitch change down one octave

is requested, this may be achieved by padding the forward transformation data with an equal number of zero-amplitude couplets, followed by a reverse transform of twice the length. Further, the scaling may be entirely arbitrary, resulting in variable numbers of channels in the transform process, and therefore requiring the use of mixed-radix algorithms. If a systolic approach were to be adopted, the program would have to take the form of a dynamically configurable systolic array of some considerable sophistication.

Since the program essentially generates work-packets, consisting of a forward or inverse Fourier transformation in addition to some simple data manipulation, interspersed by some relatively computationally trivial user-defined manipulation, summation and buffer management, a dianomic ('processor farm') approach seemed suitable where a small or medium-sized array of Transputers was available. A program is dianomic if it sends out jobs to remote processors as they become available, collecting results from the network when jobs terminate. Koikkalainen & Sauer[60] have shown this approach to be a practical solution for problems which can be broken down into an number of sub-tasks greater (preferably, much greater) than the number of processors in the network. As well as suiting the demands placed upon the software system, it is possible to construct a dianomic program which configures itself automatically to the network available as the program is booted. Indeed, the Transputer C compiler from 3L Ltd.[61] provides function calls which send and receive jobs across an arbitrary network. Kamangar, Duderstadt & Smith[62] have shown this technique to be efficient in neural network simulation for up to 30 transputers.

5.3 Preparing a Concurrent Phase Vocoder

Rather than attempt the construction of a new phase vocoder from scratch, it was decided to start from a reliable program and add the modules necessary to support concurrency. To this end, the phase vocoder program used by the C.D.P., written by Bentley and Henderson, was obtained in the source code. This program

uses a mixed-radix algorithm as for Fourier transformations; this has clearly been transliterated from a FORTRAN version, as the C source retains the structure of its predecessor exactly with regard to labels and the use of 'goto', although no source is credited in the comments. It therefore carries with it, implicit in the age of the original, a certain accepted reliability and numerical accuracy.

The first task for the programmer preparing to make the system operate concurrently is to separate the program into three parallel threads: the first reads data from the input file stored on the host file system and sends them for forward Fourier transformation; the second performs the manipulations on the results as specified in the command line, and then reissues the data to the network for a reverse transformation; the third performs the weighted-overlap-and-add operation on the returned, modified time-samples, and writes the results to disk. Principles borrowed from the development of the buffering algorithms used in the CSOUND package are used to ensure minimum waiting, and these, together with the modifications made to the structure of the sequential program, will now be described.

Figure 11 shows the new program structure. When a program is configured for a 'flood fill' application, which uses all the processors in an array of arbitrary topology as 'slave' or 'worker' processes to the root's 'master', two executable binary files are required. The first, 'master' process can be linked with the normal C language run-time library which permits access to the host computer's file system. The second 'worker' process, however, must be linked with the vestigial stand-alone library, which assumes no direct connexion to the server's operating system facilities. All input and output for the worker tasks is *via* the C language extensions which provide network facilities.

The master task, running on the root transputer, has the additional responsibility of maintaining the ordering of work packets arriving back from the network. Because all the processors in the network are running asynchronously and reading data on a demand basis, the order in which work packets are returned is not guaranteed to be the same as the order in which they were sent. An example program

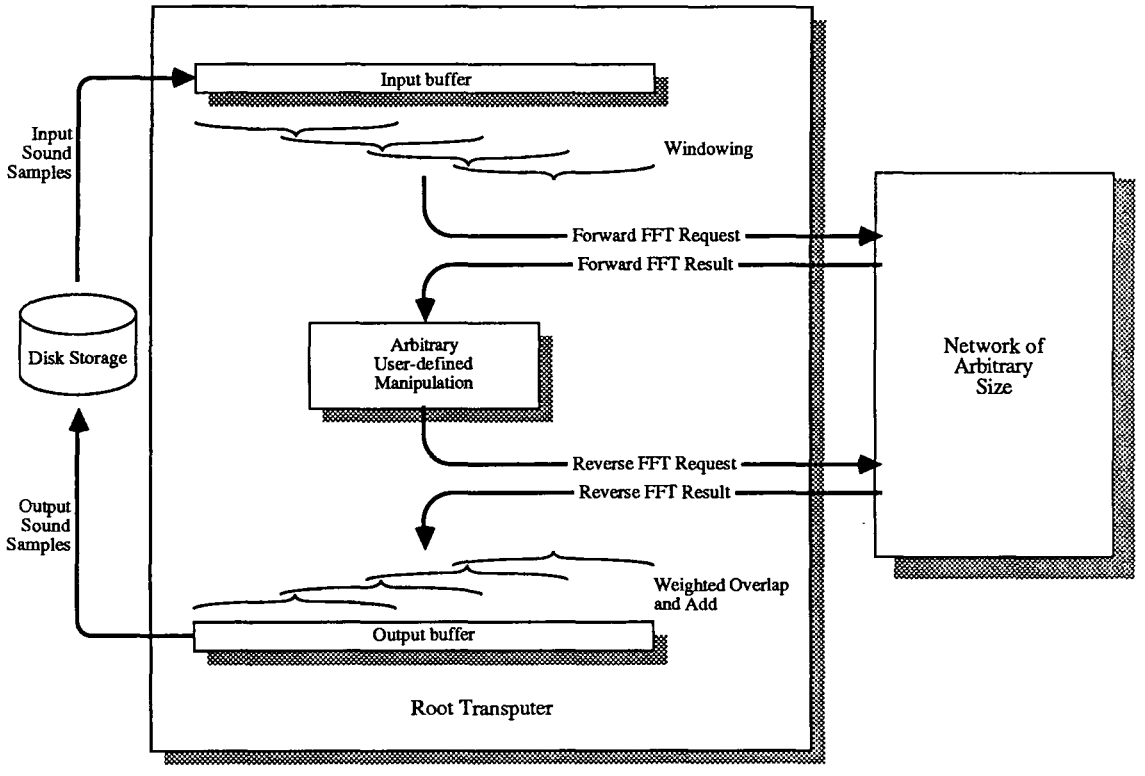


Figure 11: Software Structure of the Parallel Phase Vocoder

provided by 3L Ltd.[63] performs the calculation of linear prediction coefficients on an array of Transputers, using master and worker tasks written in FORTRAN. The solution adopted is to force the host to re-order the result data: as the result packets containing the predictor coefficients are received across the network, a serial number is examined to establish the position in the output file in which the data should be placed; the analogy to the FORTRAN 'DIRECT ACCESS' file in the C programming language is to use the `seek()` function call to reposition the output file pointer before the data are written.

Experience shows that maintaining as low a performance demand on the host computer and its server program as possible is beneficial in the long term; so for the phase vocoder, an alternative solution was sought. Indeed, the necessity to perform the weighted overlap of the result packets enforces the serial writing of the manipulated sound file, unless this summation process is to be delegated to the host machine as well. The 3L technical note specifies the conditions to be satisfied before an application might be considered suitable for dianomic re-coding: amongst these is that the result packets contain data which are *completely independent*, and that

> If the results of one sub-job are needed for the next (or some later) sub-job, a processor farm can't be used, and some other technique of parallelizing must be found.

5.4 Supporting Dianomic Concurrency

The method adopted to overcome the data-dependence objection is a modification of the window-lookahead technique, well known from the field of telecommunications. By inserting a buffer pool at the point where packets are sent to the network, it is possible to reorder the received data without causing the master task to wait, so long as the number of buffers in the buffer pool exceeds the error in the ordering of the result packets. For example, suppose four buffers are

available. Data packets requiring the forward F.F.T. processing are assembled in these four buffers, and are transmitted to the network in the order (1,2,3,4). Suppose further that the packets are returned in the sequence (1,4,2,3) due to the different completion rates of the slave processors. Writing the second thread of the program to remove in-sequence data as soon as it arrives means that the fifth data packet can be sent to the network as soon as the first has been removed, which may be before any of the other results have actually arrived. Thus, the error in ordering of 3 has been absorbed because the buffer pool is more than 3 buffers in size.

It might be argued that this is a potentially restrictive solution, because the buffer pool size must increase with the number of processors in the network. This is a misunderstanding based on the (erroneously assumed) connexion between the sequence number modulus and the number of slave Transputers. It would lend more credence to this case if there were a single type of job and single buffer pool, but in fact there are two buffer pools supplying forward and reverse F.F.T. data, so that results are removed from the first as soon as they become available. Even in the worst case of a very large transformation demanding more time of the slave than it takes to fill the entire network with data read from the host file system, the implementation limit of 16384 reals (65536 bytes) transforms imposed by the original program means that a root node with only 1MB of RAM can easily support upwards of 10 buffers. With a more normal 1024-point F.F.T., somewhere in the order of 100 slave processors can be used with a one-to-one match with the buffer space in the root's memory. The medium scale parallelism which is desired can therefore be achieved with absolute task independence, even placing only modest demands upon the memory capacity of the root processing element. It should be emphasised that each of the slave tasks requires only sufficient memory to hold and process one buffer of data, and that the slave memory requirement is therefore relatively tiny.

5.5 Implementation of the Buffer/Flood Algorithm

Memory is allocated to the buffer pools by calling the `nq_init()` subroutine in the module `netq.c`. A constant `MAX_NQ_MEM` bytes is divided between the buffers according to the following rules:

If the data on the disk is analysis (i.e. frequency domain) data, create two analysis buffers and allocate the remaining memory to synthesis buffers;

Else if only the analysis phase is required, allocate all of the available memory to analysis buffers;

Else divide memory such that the number of analysis buffers equals the number of synthesis buffers.

Thus `nq_init` recognises that the disk file may contain data from a previous analysis run, in which case forward F.F.T. requests will never be made to the network, or that the user may request analysis only, in which case the results from the forward F.F.T. are written straight to the output file, and the third thread of the program (usually performing the weighted-overlap-and-add operation) is never started. Having generated the buffer pools and assigned values describing the number and size of buffers to appropriate variables, `nq_init` proceeds to set up the buffer access locking semaphores. It will be recalled that the access barring mechanism used in the CSOUND program used three semaphores to implement a buffer/demultiplexor structure, or five semaphores for a deadlock-free buffer multiplexer. An extra level of sophistication is demanded in this case, however, because the order of validation of buffers in the buffer pool is not guaranteed to be sequential. An efficient solution is achieved by creating one (initially lowered) semaphore *for each buffer*. The general network receiving thread, `nq_receive()`, is started, and `nq_init` returns.

The detail of the buffer-pool mechanism is best understood by first examining the data structures involved in the communication phase. Each message, be it either a network request or a network result, consists of a task descriptor followed by the contents of a buffer. The most concise way of describing the structure is simply to quote its definition from the NETQ.H file:

```
typedef struct {
        int   sequence; /* Index into buffer array */
        int   action;   /* Place result in anal or syn buffer */
        int   size;     /* Length of following buffer in bytes */
        float *buffer;  /* Locataion of source/result buffer */
        float pitchfac; /* Pitch change factor */
    } NQ_TASK;
```

Each field of this structure must be returned unmodified by the slave process to the master; this is the mechanism by which the order of arrival of the results is established. The integer `sequence` is the subscript into the array of pointers to buffers associated with the appropriate buffer pool. The integer `action` can take one of the special values `NQ_FORWARD_FFT` or `NQ_REVERSE_FFT` according to the direction of the transformation required by the master. This field is also used by the worker to determine whether the polar-rectangular or rectangular-polar conversion should be performed, and by the master's network receiving thread to determine whether an analysis or synthesis semaphore should be raised to validate the returned data. The `buffer` field points at the buffer in which the original data was stored. This is meaningless to the worker thread, as it represents an address in the memory of the root Transputer, but is included to avoid the necessity for the receiving thread to have to perform its calculation once again. The variable `pitchfac` is a floating point scaling value used by the worker, so that as much computation as possible is decentralised.

When the first thread has read a buffer-full of sound samples from the input file stored on the host machine's disk, a descriptor is constructed, specifying which

buffer is to be transformed and in which direction the transformation is required. The task descriptor and the buffer are sent to the network with one simple call to the routine `nq_send`. The calling program then simply loops so long as there are buffers free in which to place the input sound samples, and there is still data in the input file; the former condition is established by the state of the semaphore containing the number of free buffers, in much the same way as the multi-thread buffer algorithms of CSOUND.

Eventually, the transformation will have been completed, and the (unmodified) task descriptor is sent back to the root Transputer. It is received by the `nq_receive` thread, and the data in the source buffer is replaced by the result of the transformation, which immediately follows. `nq_receive` then calls `nq_mark_as_valid()`, passing a pointer to the received task descriptor as an argument. This enables the semaphore associated with the new data's buffer to be raised, indicating that it is ready for use by the second thread.

The second thread is responsible for transferring the data from the analysis buffers as it returns from the workers to the synthesis buffers whence it is sent to the network for resynthesis, with some user-defined modification *en route*. It waits on one of the members of the array of semaphores which are associated with the validity of data in the analysis buffers; the subscript of the semaphore which is used in incremented cyclically modulus the number of analysis buffers. This approach simultaneously provides inter-thread synchronisation and the mechanism for coping with out-of-sequence data being returned by the network; the thread will simply wait until the buffer with which it is concerned is marked as valid, and will continue to remove data from it and subsequent buffers until it reaches one which has not yet been received. Having performed the manipulation specified by the user, the frequency domain data (now placed one of the synthesis buffers) are sent to the network, using a fresh descriptor which will request an inverse transformation. The analysis buffer is now free, and this is indicated to the first thread by signalling on the semaphore `anal_free`, indicating that one more analysis buffer has become available.

The third, and final, thread operates in much the same way as the second: it waits on one of the array of semaphores associated with the synthesis buffers; and reads data from valid result buffers and performs a weighted summation into the output buffer, before signalling on `syn_free` that the number of buffers available to the middle thread has increased. The master task is therefore described in terms of three loosely coupled threads which collectively perform the function of the phase vocoder, plus one very simple 'tooth-fairy' thread to receive results from the network, and replace the original data with its transformation before the receiving thread can awake.

Worthy of mention are the two exceptions to the above course of events, when the user specifies either the synthesis only or analysis only option in the program command line. In the case that only analysis is required, the second thread writes frequency-domain samples onto the disk immediately the results are returned from the network, and the third thread is never started; when only a synthesis is expected, the course of events is largely unchanged, except that the number of analysis buffers is reduced to two, and that the first thread marks data as valid as soon as it is placed into an analysis buffer rather than sending it to the network with a forward F.F.T. request.

The remaining code to describe is the worker task, and this is not too arduous. The worker comprises two modules: MXFFT.C, which is an entirely unmodified copy of the C.D.P. source code to perform mixed-radix fast Fourier transformations; and FFTW.C, written to provide a harness through which the mathematical routines may be called. Operation of the main program is simple: it loops infinitely, receiving a task descriptor and a buffer full of data from the network; then, depending upon the `action` field of the task descriptor, it either performs a forward F.F.T. followed by a rectangular-to-polar conversion, or a polar-to-rectangular conversion followed by an inverse F.F.T.. The task descriptor is sent back to the master task, followed by the result of the F.F.T., and the code repeats *da capo*.

5.6 Conclusion

Isonomic decomposition has shown to be effective for large-scale programs such as general purpose direct synthesisers. Unfortunately, some algorithms, the phase vocoder being one of them, cannot be reduced in this way; their subsequent output relies upon storage of the previous outputs. This necessarily imposes a sequential order of execution, and makes independent copies of some program running in isolation unable to produce an equivalent result. The parallel phase vocoder which has been coded relies upon a concurrency of delegation rather than cooperation; a master program issues slave programs with work packets, the results of which are collated by the master regardless of the order in which they actually arrive. The next chapter contains the results of the benchmark tests and an evaluation of performance of this system, comparing it with the case where both control and data are distributed.

Chapter 6

Performance Evaluation of the Distributed Phase Vocoder

6.1 Anticipated Benefits

THE STRUCTURE OF THE DISTRIBUTED PHASE VOCODER described in the previous section derives its speed-up from the delegation of the most computationally intensive part of the task, the Fourier transformation, to a network of processors. There is therefore an innate restriction on the ultimate performance of the system, because whatever the speed of the slave processors, some data manipulation has to be performed by the master prior and subsequent to the transformation taking place. Some performance enhancement is to be expected as extra processors are added, but as the slave processing resource increases, the point is approached where the master is using its entire capacity for the windowing, user-defined data manipulation, and weighted overlap part of the process.

Another bottleneck occurs at the access to the host file system. The master transputer communicates with the host through a server program written in C. The data rate of the program in passing file information to and from the hard disk is a limiting factor. Mention has been made in the first section that a speed enhancement was achieved by buffering the output data path of a synthesis program; this enabled the transputer to continue useful calculations of sound output samples while waiting for the host file-system to respond. Users have reported that programs to generate sounds simple enough to require less than

	Number of Processors			
(N,D)	1	2	3	4
(2048,128)	291	169	No further reduction	
(1024,256)	78	56	55	No further reduction

Table 8: Execution Time in Seconds for Unmodified Resynthesis

approximately one-fifth[1] of their specified duration to be computed suffered from suspension due to limited I/O bandwidth beyond the absorbtion capability of a buffering algorithm.

6.2 Benchmark Results

The flood-fill phase vocoder program automatically configures itself to use all the processors in a transputer array when it is loaded; all that is necessary to perform the speed test with a varying number of processors is physically to remove processors from the array, by unplugging their link cables, before loading the program.

The results of the benchmarks are interesting, because they illustrate the efficiency of the fast Fourier transformation module. A monophonic soundfile was generated by recording the utterance "Durham Music Technology Group" by a female speaker. This lasts for 2.06 seconds, and occupies a disk space of about 132KB at a sample rate of 32KHz. The distributed phase vocoder program was then used to perform an analysis/synthesis of this file, producing an output file sonically identical to the original. The test was performed twice, using 2048-point transformations spaced at 128-sample intervals at first, and then using a 1024-point transformation at every 256th sample. Table 8 contains the reported timings: N is the number of points in the Fourier transformation; D is the (unfortunately named) decimation factor, being the interval at which the transformations were taken.

[1] Using an Intel 80286-based file server running at a clock rate of 20MHz

The additional processors were connected as a ternary tree, so as to reduce the path length to the master as far as possible. Where the table carries the legend "No further reduction", the addition of extra processors made absolutely no difference to the speed of processing. In fact, it was even possible to disconnect these processors *while the network was running the program* without any interruption.

The maximum speed-up achieved with the 2048-point transformations is a factor of 1.7, using two processors. This corresponds to a performance index of 0.85. The speed-up in the second case where 1024-point transformations were used is only a factor of 1.4 times, yielding a performance index of 0.7. In view of the fact that host file-system I/O is already buffered in this program, the most likely cause of the bottleneck which prevents any further speed increase is the processing power of the master node. Each worker packet containing time-samples needs to be multiplied by the windowing function specified by the user in the command line. This function has to be performed independently for each of the overlapping windows, and requires at least N multiplications in floating-point. Data received must then be moved to a different area of memory before being sent for reverse transformation, and the result of this transaction is to provide N further data points which must be scaled according to a different windowing function, and then summed with previous data to produce the final output signal. With at least 3N floating-point operations performed by the master transputer, it is easy to see that the computing demands are beginning to catch up with the NlogN requirement of the Fourier transformation process as they are distributed across more and more processors. With additional work, it would have been possible to migrate the windowing function onto the slave processors by adding a field to the task descriptor specifying the window length and type required. Since it is guaranteed that, at most, two different windows will be used, worker tasks could hold these window functions in arrays between receiving task packets, avoiding the calculation of a trigonometrical function for each task. This would require a slight increase in the slave node memory requirements, currently an

	Number of Processors		
(N,D)	1	2	3
(512,128)	129	122	No further reduction

Table 9: Execution Time in Seconds for Analysis Only

extremely modest 128KB., and so was not undertaken. It might be considered possible as a practical future extension, especially as the users of the system have a three transputer machine fitted with 1MB per node. However, it will only postpone the onset of the bottleneck demonstrated so effectively above.

6.3 Checking the File Server Interface

The explanation of the limited speed-up potential of the distributed phase vocoder makes the assumption that the master transputer is compute-bound, and that the I/O bandwidth of the host machine is not causing data starvation. This assumption was made on the observation that the lamp on the front panel of the host machine which indicates that the hard disk drive is being accessed remained unilluminated for the most part. However, this result is also consistent with a very fast disk drive fitted to a computer with a processing power too limited to access it continuously under the control of the server program. A further test was made to eliminate this possibility.

The phase vocoder was instructed to produce an analysis file containing 512 points of spectral information from the original utterance, with the Fourier transformations being taken at a decimation factor of 128. This results in 512 complex data points being generated, each of eight bytes in length, for every 128 two-byte input samples; the length of the output file is 16 times the length of the original, or some 2.1MB. Clearly, this not only places a greater strain on the file server, but also halves the demand placed upon the transputer system; no inverse transformation is required. The results are presented in table 9.

Since it requires at most 120 seconds to write a file of this size, the time taken

to write a resynthesis file should only be in the order of 25 seconds[2] — much less than the times recorded for the resynthesis experiments. Recalling that the host I/O is buffered, the previous assertions regarding the fact that the master processor is compute-bound are substantiated.

6.4 Consequences of these Results

The general rule that limited capacity to provide raw data and control information is the cause of low performance indices in multi-processor systems is upheld. Delegation of the mathematical operations in this system demonstrated the fact that the speed-up factor tends towards a limit if there is no also way of delegating control functions. The ramifications of this statement are that systems which separate functional elements by control paths of limited bandwidth, such as the Project Phi proposal by Cassady,[64] must ensure that there is a mechanism of increasing this bandwidth as extra processors are added. This was the case with the CSOUND example presented in the first part, as it is with systolic systems based on fixed-ratio data movement,[3] like the ring system proposed by Kirk.[65] In the third and final section, armed with this knowledge and with the experience of users' requirements, a machine with a large number of processors which avoids these limitations will be described.

[2]since the data set is only two ninths of the size, one might expect the file access time to be scaled accordingly, making the rather naïve assumption that the time to write and read a given data block are similar.

[3]by which it is meant that the ratio between the number of input data and the number of output data is fixed in advance and known.

Chapter 7

Considerations in the Construction of Real-time Electroacoustic Instruments

7.1 Introduction

THE FIRST TWO PARTS have discussed the technical benefits of running modified, existing software on an array of processors, in order to achieve a significant increase in processing speed. In this, the final part, we turn to discuss the implications of very high speed computers. It is suggested that, once sufficient processing power is available to perform even the most demanding synthesis algorithms in real-time, the traditional man-machine interfaces become redundant or unsuitable.

The sophistication of input and output devices is generally required to increase as the available computing facility grows in power.[66] In the context of a real-time instrument, the approach to the design of man-machine interface must be to reduce the number of input parameters required for the control of an instrument to the absolute minimum consistent with the desired level of control. This has the effects of increasing the ease of use of a particular item of equipment, and enabling the user to make intuitive or investigative changes in real time. Thus a suitable man-machine interface to a given software or hardware machine is essential to realise the machine's functionality.

Numerous approaches to man-machine interfaces exist for non-real-time systems, and these tend to fall into graphics driven or command driven partitions. Invariably, the command language or graphics environment is evolved rather than designed; the features that are presented to the user are those which occurred to the programmer of the system in question and are those which drive the application in the easiest way. This is not the same as presenting the user with an interface which conforms as closely as possible to the requirements which the application was designed to satisfy. For example, many early digital synthesisers permitted the direct input of a time-waveform *via* a graphics tablet: whilst this gives the user total control over the *timbre* of a musical note in principle, the interface is unsatisfactory because a change in the input parameter is by no means intuitively linked to the result.

7.2 M.M.I. and Application Program Design Methodology

The point of connecting an application to the user through a Man-Machine interface module is that the possibly conflicting interests of ergonomics and program efficiency are more rigorously partitioned. This is especially the case where parallel processing is available: under such circumstances, the M.M.I. module of a finished application may be run on different physical processors; no impact whatsoever on the performance of the rest of the system accrues.

It would seem to be the case that little exists in the way of graphic or other representation of a single note's characteristics. Those definitions currently in use are either couched in psychoacoustic terms, which are too loosely defined to lend themselves to computer implementation, or else rely on scientifically concrete ideas such as frequency, envelope and so on, which are too low level to express conceptual ideas about a sound sufficiently concisely. What work has been done in this field seems to be concentrated on the macro-definition of sound:

Schaeffer's[67] syntax for *Musique Concrète* maps the whole of a musical sound onto three two-dimensional graphs, but becomes less well defined where the specification of physical note attributes (as may be required to control a synthesiser) are concerned. The three *plans* which are used to describe a sound are based on well-understood psychoacoustic parameters, but as Schaeffer's work was directed more towards the physical manipulation of sound recorded on tape, it does not deal with the complex conversion procedure which must be used to extract physical control parameters from their psychoacoustic correlates.

7.3 Man-Machine Interfaces from Scratch

Before considering the implementation details or even undertaking the functional description of a man-machine interface, it is wise to spend some time defining exactly what such an interface should do. The following attributes are required:

- The input to the M.M.I. is in a form most acceptable to the user, and therefore, *ipso facto*, contains an element of user-definition. Little can be achieved to aid composers unless they give some indication of what they want.

- The output from the M.M.I. is in a form most acceptable to the applications program writer. This implies a high degree of cooperation between the author of the real-time system and the author of the M.M.I.

- Any real-time *general purpose* machine for the reproduction or synthesis of sound inevitably has a very large number of control parameters associated with it. The M.M.I. should successfully map any (user-defined) combination of these control parameters onto the parameters required as inputs for the rest of the machine. This is best thought of as mapping points in a low-dimensional domain onto points in a higher-dimensional range. The exact configuration of the M.M.I. determines the nature of the mapping. Hence the range of *control inputs* which can be achieved using the available *input*

controls can be fine-tuned to permit the exploration of the desired part of timbre-space.

7.4 More Traditional M.M.I.s

Assuming that the main problem of a man-machine interface module would be to map an n-dimensional input control to an m-dimensional control parameter space, it may be asked how many control parameters may the performer reasonably expect to provide simultaneously. Let us consider the violin as a machine which is to be controlled, and list some relevant control parameters.

Parameter Name	Method of Control	Physical Effects
Pitch	Choice and stopping of string	Frequency
Volume	Bowing (including weight, speed, angle of bow)	Amplitude, Harmonic Content
Timbre	Distance of bow from bridge	Amplitude, Harmonic Content
Vibrato	Movement of left hand	Frequency
Envelope	Bowing, plucking	Amplitude/time

This table illustrates that the concept of a 'note' as imagined by the performer must be expanded into several control parameters for a synthesising machine; what the performer sees as a one-dimensional quantity in fact requires many control dimensions. A M.M.I. may map a number representing a single 'envelope' parameter (e.g. 0 for staccato through to 1 for legato) onto a curve in a far more complex control-space (say, ADSR envelope, or even a more complex set of parameters). Furthermore, the elements of the input vector need not be orthogonal: in this example, both the bowing action and the distance of the bow from the bridge affect the amplitude and timbre of the note. Clearly, the interpolation required between adjacent points in the control space is in itself a non-trivial problem and may require some considerable processing power. It must also be

taken into consideration that the input parameters themselves may be non-linear: pitch is related to log(frequency), and volume to log(amplitude), for example.

As if the problems of non-linear, multi-dimensional interpolation were not enough, many synthesisers exhibit changes in output for smooth changes in input parameters which verge on the discontinuous. A prime example is that of the over-driven frequency modulator, where a change in the modulation depth causes the partials of a tone suddenly to be shifted in frequency.

A successful M.M.I. kit will require tools to specify the interpolation laws along a given dimension in the performer's control space, apply weighted averaging to any of the specified control parameters, and even cope with functions which are only piece-wise continuous. It will almost certainly make use of graphic presentation methods and have sufficient built-in processing power to produce the synthesiser performance parameters in real time.

7.5 Real-time Methodology at IRCAM

A new approach to the control of real-time instruments was suggested concurrently with the construction of the 4X multi-processor machine at IRCAM, Paris.[68] Xavier Rodet pointed out that a 'Patched Note-list Function' system, as used by CSOUND and many other non-real-time programs, is necessarily inapplicable to the control of a real-time machine, relying as it does upon events with *a priori* unknown duration. Since the duration of an event cannot be specified by an observer until its completion, the note-list cannot be produced instantaneously, but must lag behind the source by at least the duration of the event. It is therefore impossible to perform a real-time control function using a 'Score and Orchestra' system analogous to CSOUND's.

The proposed alternative, implemented on the Sun Mercury Workstation and others, is that of 'Algorithmic Synthesis'. The rationale is not to define the precise characteristics of each note explicitly by means of an orchestral command, but instead to supply the synthesiser with generic information as to the desired

quality of the sounds. The subtleties of each note are provided by the performer when the instrument is played; the same approach will be advocated in the following chapters for real-time control. However, it seems that instead of becoming an immediate success with composers, the prevalent feeling was that the original 'note-list' format was usually preferable.[27] The reasons for this should be properly understood before a starting on the control software for a real-time machine, lest we reproduce such an undesirable environment too accurately.

7.6 Components of an Algorithmic Synthesis System

Typically, Algorithmic Synthesis systems retain the orchestral definition from their imperative predecessors written in advance; the real-time part of the system replaces the score. Since the score is no longer an event-list which can be compiled ahead of the output sample stream, the syntax of the orchestra is necessarily modified. The orchestra is now rule-based rather than imperative; an analogy can be made with the distinction between functional and imperative programming languages. Just as functional programming has failed to gain a mass following for practical use by computer programmers, I suggest that composers may have been reticent about adopting such methods of specification for their orchestras. Despite the more concise nature of functional programming languages and their close approximation to rule-based semantics, the fact remains that many composers prefer the syntax of imperative languages to the syntax of a functional specification; the former is more familiar and is therefore generally found easier, even though it is not inherently so.

It has been proposed that the solution to the control of a real-time instrument is to limit the number of control parameters collected from the performer to direct the trajectory of the sound through a more restricted range of timbres.

The particular range of timbres available becomes the choice of the 'orchestrator'; an analogy with conventional instrumental practice. However, this proves a little too restrictive for the control of the vast range of sounds available from an electroacoustic instrument. The solution is to divide the score into two components: a real-time source which is taken directly from the performer, and a pre-programmed source which modifies orchestral parameters in the manner of presets on an organ console. This permits the composer to specify an 'orchestra' with a very broad range of possible timbres, and then to select an available set which the performer may use. This is similar in some respects to the organ builder providing various stops for the organist to select as he plays. The methods of controlling an electroacoustic instrument are more sophisticated, including time-varying functions. An approach similar to this is used by the IMPAC system at Stockholm,[15] where composers start with a granular synthesiser and sculpt the range of timbre-space available to the performer.

The necessary components of a successful Algorithmic Synthesis system seem to retain the score and orchestra entities, albeit in a modified form, and include an additional score-like control for direct use by the performer. It is often difficult to know in advance how a performance gesture should be translated into control parameters suitable for use by the instrument. The performance interface should therefore allow automatic assimilation of the significance of various performance data, possibly using one of the 'teach-by-example' algorithms common in robotics.

Previous attempts at solving the M.M.I. problem have concentrated on building machines which follow a human performer. *inter alia*, Morita[69] demonstrated a system in 1989 which processes the image from a C.C.D. camera to enable a computer-music system to follow a human conductor; Vercoe[70, 71, 72] has tracked polyphonic music using cognitive models to enable the construction of a synthetic accompanist; and Dannenberg[73] has demonstrated a MIDI toolkit which permits various sophisticated real-time interactions with live musicians. A comprehensive review of the control possibilities offered by MIDI has been presented by Yavelow.[74] These techniques have all exerted a considerable influence

upon the interaction specification presented here, but are too loosely coupled to be used directly in an electronic musical instrument. Several commercial units are available which perform the conversion of an acoustical input into a MIDI command sequence, and some of these apply advanced artificial intelligence techniques to increase the level of interaction.[75] Indeed, some demonstrations of gestural capture devices have shown the MIDI interface to be capable of impressively accurate control. Waisvisz' performance at the Second STEIM Symposium on Interactive Composition in Live Electronic Music received a critical accolade from Curtis Roads who wrote:[76]

> After several minutes, Waisvisz stood up and pulled on a belt attached to a long ribbon cable and two performance control devices that he strapped to his hands... These devices, called 'The Hands', sense relative position and orientation in space, and contain switches and potentiometers for controlling pitch, timbre, and selection of instruments (patches) via the MIDI protocol. ...Thanks to Waisvisz, computer music in performance has finally come into its own as a completely new virtuoso medium, quite apart from traditional instruments or traditional clavier controllers

However, the final result, even of Waisvisz' impressive performance, must be constrained by the limited bandwidth of a MIDI control wire, which is around 32Kb/s. The direct MIDI controller is still designed to collect data of interest to the synthesiser, rather than data of interest to the musician.

In summary, patched note-list functionality is inadequate for the control of a real-time system. Algorithmic Synthesis goes some way towards alleviating this deficiency, but a purely rule-based system requires the use of functional techniques which are often unpopular with users. The following chapter contains the description of a hardware and software approach to real-time gestural capture. In attempting to collect control information which corresponds with the

performer's conception of a musical rendition rather than the sythesiser's control requirements, it is hoped to improve not only the range expression of which an electronic instrument might be capable, but also to make the medium more accessible to the less computer-literate musician.

Chapter 8

Towards the Systolic Synthesiser: Craftsman, Composer, Performer

8.1 Introduction

THE PREVIOUS CHAPTER GAVE AN OVERVIEW of the problems facing the designer of an electroacoustic instrument; this chapter proceeds to explain a technique by which such an instrument might be constructed. By dividing the task into three discrete fields, it should be possible to make the instrument accessible to performers without special knowledge of computer music.

In the context of traditional music, the three skills of the instrument builder, the composer and the performer are distinct and individual. In the field of electro-acoustic music, however, the composer is often expected to be fluent in the 'construction' of his orchestra, often using highly specialist computer algorithms, as well as composing the music and possibly providing the mechanism for interaction with human performers. A tripartite approach to the problems of electro-acoustic music enables a more structured solution to the problems of using this medium. One possible way for these three different skills to operate independently is to provide a mechanism for the performer to select from a wide range of possible timbres by means of gestural control. Since acoustic instruments already provide an extremely well developed method of gestural capture, it is desirable to be able to use the information which their sound contains to derive control parameters for

the electro-acoustic instrument. A move towards this end is the development of a real-time phase vocoder, and its implementation on an array of microprocessors functioning concurrently.

8.2 Music and Electronic Techniques

In his study for the International Music Council[77] entitled "Composer Performer Public", Everett Helm states:

> Since the early '50s when electronic compositions began to appear, enough time has elapsed to afford a certain perspective. ... It is clear that the possibilities of electronic music are not infinite as was suggested in the euphoria of the early days. The kinds and varieties of sounds that can be created electronically seem to be limited to certain typical ones which recur constantly. ... The enormously complicated rhythmic, pitch and interval relationships that can be created electronically have proved to be more advantageous in theory than in practice, for it has become apparent that the human ear is not always capable of perceiving the complications which the human brain can devise.

These problems associated with electronic music from its beginnings in the 1950's may be the result of a misunderstanding of the purpose the new technology is trying to achieve. It is indeed all too easy to produce "conveyer-belt compositions, hardly distinguishable one from the others" (Helm again). It is no coincidence that the title of this chapter is not dissimilar from the title of Helm's treatise: the following sections will present a suggestion as to how the "infinite possibilities" of electronic synthesis may be harnessed in live performance as well as non-real-time ('tape') compositions.

Traditionally, a piece of music is performed by a specialist 'performer', whether 'live' in a concert hall or in front of a microphone in the recording studio. The

performer need have no training in composition; it is his job only to realise the musical notation on the page in front of him as sound. Similarly, the composer need have no performance capacity other than a technical appreciation of the capabilities of the instruments for which he writes. Few symphonies would be written if it were a prerequisite that the composer should be competent to perform each orchestral part. It is also unnecessary for the craftsman who built the violin to be skilled either in performance or composition; his function is to understand intuitively the physical acoustics of the instrument he is constructing, and to build one capable of performing well its orchestral rôle. By admitting the specialisation of the delivery of a piece of music into these three separate phases, a vast level of expertise has been accumulated over the past millennium, handed down by oral tradition.

One of the aims of this chapter is to propose a solution to the problem of providing such specialisation for the electro-acousticians of the 1990's, and to explain the internal mechanism of a real-time electro-acoustic machine based on these principles. With knowledge of the internal operation of this machine, it is hoped that it should be evident why precisely that knowledge is *not* necessary for the machine's tailoring by the specialist electro-acoustical *craftsman*, or its use by the specialist *composer*.

As soon as such a machine begins to work in real-time, it becomes an *instrument*. As soon as the machine becomes an instrument, its operator becomes a *performer*. It is hoped that this approach will enable a greater number of musicians to be involved in electroacoustic performances than has been possible hitherto, and will ultimately be superior to any attempt to design special electronic human-computer interfaces[78, 79], by virtue of not requiring the performer to learn the use of any new instrument, but rather to modify the technique applied to his existing one.

8.3 Consideration of Control and Synthesis

In fact, there is not an infinite number of perceivable sounds. From signal processing theory, it is known that a signal of given bandwidth may be represented by a series of time-samples; the sample rate must be at least twice the signal bandwidth, and the accuracy of these samples is related directly to the quantity of noise in the reconstituted signal. By setting the sample rate sufficiently high to capture all signals which the human ear is able to perceive, and by taking samples of sufficient accuracy to equal the ear's dynamic range, a sound may be regenerated which is acoustically an identical copy of the original. Furthermore, it is possible to synthesise any repeating waveform to an arbitrary accuracy by the summation of sine waves of different frequencies. This method of synthesis, referred to as 'additive synthesis', has the advantage that it is capable of producing arbitrary timbres, but requires that a large quantity of information is passed to control the amplitude and frequency of many individual oscillators. However, it is eminently suited to parallel processing, where each of the processors may contribute a small number of partials towards the final signal. This approach has been realised by the Durham Music Technology Group, using a small array of Inmos transputers connected in a pipeline configuration.[37]

If the samples thus obtained are concatenated and read as one long number, then the set of all sounds lasting a given time can be enumerated simply by writing down all of the integers between zero and the largest possible within the limit imposed by the duration of the signal. The problem of the electro-acoustical craftsman is to build an instrument which synthesises only a fraction of these possible sounds under the gestural control of the performer. Unfortunately, the size of the set of possible sounds is rather large: sounds lasting up to one second, at C.D. quality, number approximately 2^{100000} (about 10^{30000}); since the universe has been in existence for a mere 10^{17} or so seconds, a listener who began to audition these sounds at the creation would so far have heard an insignificantly small

fraction.[1] Clearly a method of gestural control is required which is extremely selective.

8.4 The Phase Vocoder and Intuitive Control

The Digital Phase Vocoder described by Flanagan & Golden[80] was intended for use in reducing the quantity of data transmitted along a telephone line during speech transmission. A more computationally efficient implementation of the Phase Vocoder, using the Fast Fourier Transform to replace a bank of filters, was introduced by Portnoff.[81] A comprehensive introduction to the principles involved is given by Dolson[82] and by Gordon & Strawn.[83]

An essential feature of the Phase Vocoder is that the output signal consists of a sequence of spectra, illustrating the evolution of the partials present in the input sound with time. This separates the *timbral* information of an input signal from its *temporal* evolution. The former is represented in the relative amplitudes of the partials in a given spectrum; the latter as the change of partials' amplitudes and frequencies between the times at which the spectra are taken. An important feature of the Phase Vocoder is that its output retains information about the *phase* of the partials as well as their amplitudes. By using this phase information, it is possible to interpolate a frequency trajectory as well as an amplitude trajectory for each partial as the input sound evolves. Therefore, the output of the Phase Vocoder accurately tracks the timbral information of its input signal.

Since the output of the Phase Vocoder isolates the partials present in its input signal, it lends itself immediately to the transformation of digitally sampled sound in real- or non-real-time. A brief summary of the musical applications of Phase Vocoders has been presented by Moorer.[84] An example of the sophisticated use of the Phase Vocoder in electro-acoustic composition occurs in the

[1] In fact, this number is only an upper bound on the 'number of one-second sounds'; it ignores psychoacoustic results which demonstrate that large groups of the set thus enumerated would sound identical. The point remains, however, that the synthesiser has a huge variety of timbres from which to select according to a far more restrictive set of gestural controls.

now renowned piece 'Vox V' by Trevor Wishart, which exploits the intuitive nature of the program's output by devising complex transformations between two different sounds. However, this exemplifies the use of technology by technically-knowledgeable craftsmen-composers and, by virtue of its non-real-time nature, fails to provide a pathway to direct control of a musical instrument.

Speaking at the International Computer Music Weekend at the University of Keel in 1991, Simon Emmerson said:

> We need to evolve systems which give back to the performer a sensory relation with the result; one that he/she can monitor and control through this feedback *in real-time* – the *expressive* component.

A major problem in the control of our proposed electro-acoustic instrument is one of permitting its use in performance by the technically naïve user without compromising its flexibility and diversity of timbre. It would seem that if a machine existed which could track accurately the timbre, pitch and other relevant parameters of some control signal, the problem of performance could be reduced to asking the performer to generate a certain range of sounds in a familiar way for instantaneous, arbitrary but repeatable transformation by the instrument. This 'range of sounds' might arise from the performer's voice, or from their playing a conventional instrument with which they were familiar. It would then be possible for to control the electro-acoustic performance by using a traditional musical discipline: the consummate isolation of the gestural-capture mechanism. Additionally, the intermediate representation of the control data is intuitively simple, being a list of instantaneous partial amplitudes and frequencies. This is compatible with the requirements of the instrumental craftsman, whose responsibility it is to design the transformation between the input and output timbres.

8.5 An Analytical Approach

Most documented implementations of phase vocoders in musical applications are directed towards their use for analysis of a sampled sound in a non-real-time environment; the analysis is performed off-line using previously recorded data, and a separate computer program is used either to modify the intermediate reciprocal-space representation before re-synthesis, or simply to display the results graphically. This is the case with the Composers' Desktop Project implementation, as with the transputer farm version of it described in the second part. There have recently been some notable attempts to move towards real-time working: Real-Time CSOUND from M.I.T.[71] provides for a real-time re-synthesis phase under the control of a mouse or some other such device; McGee & Merkley[85] have shown how an entire real-time system with logarithmic frequency bands can be implemented using one digital signal processing I.C..

A pivotal problem in the design of the real-time phase vocoder is the response accuracy of the filter-bank. Whether these filters, which separate the partials present in the input signal into their related output channels in the frequency-space representation, are implemented directly or by Fourier transformation, there exists a fundamental limitation: the closer to the ideal response that a band-pass filter is designed, the slower its output responds to an input signal. McGee & Merkley use twelve filters per octave in order to imitate the piano keyboard, and point out the fundamental limitation that the delay through the filter is roughly one half of the inverse of the bandwidth. Considering a Fourier implementation, this is equivalent to the bandwidth of the output bins being dictated by the number of input data to each transform.

Because of the intention to use this implementation in a metrical application, it may be desirable to sacrifice some speed of response of the filter bank in order to improve not the number of filters, but rather the *flatness* of the filters in their respective pass-bands. Figure 12 illustrates the frequency response of a four-bin Fourier transform, ignoring alias products from presented signals which lie outside

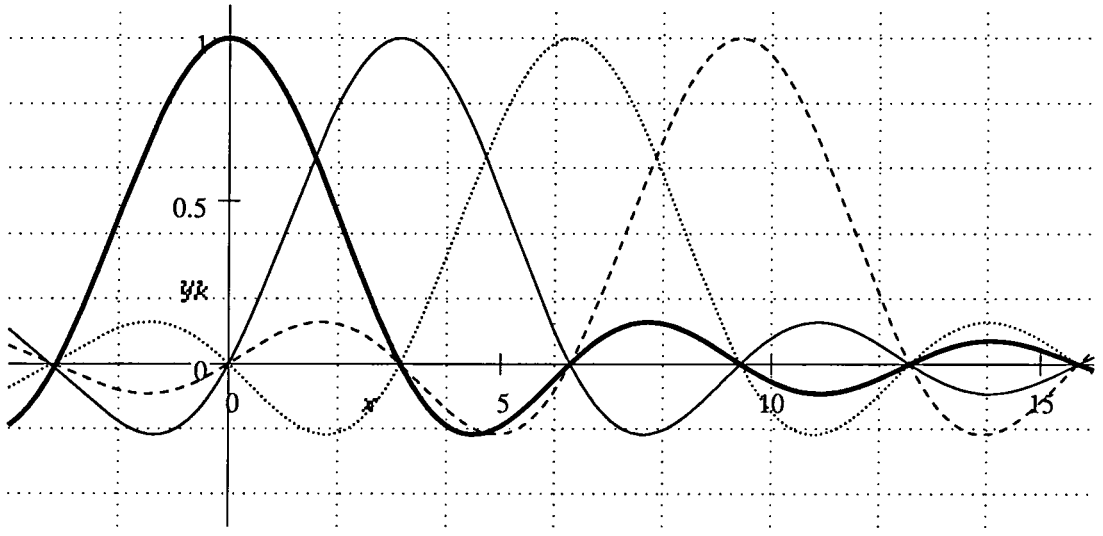


Figure 12: Graph of the Frequency Response of a Four-bin Fourier Transform

the Nyquist sampling limit. These are similar in shape to the filters described by McGee & Merkley in their real-time implementation. In fact, these authors advocate the 'fine tuning' of their phase vocoder by variation of the sample rate until the filter frequencies coincide exactly with the pitches in the diatonic scale of the instrument being analysed. This is intended to avoid errors in amplitude owing to partials missing the exact centre frequency of the appropriate filter. It is not difficult to appreciate why this should be necessary; a signal of a frequency lying just within the boundary of a given bin (the worst case) experiences an error in measured amplitude by a factor of $\mathrm{sinc}(\pi/2)$. Since signal power is proportional to the square of the signal amplitude, the measurement will be in error by a factor of approximately 0.45; more than half the energy in the signal appears smeared into other bins.

Having decided to use the phase vocoder as a data acquisition mechanism, an occam version was produced with a view to installing a parallel version on sufficient transputers to permit real-time operation. A different solution from the fine-tuning one is provided in the occam implementation. An ideal low-pass filter constitutes the multiplication of the frequency spectrum of an input signal

by a 'top hat' function; partials within a given range from 0Hz are passed unattenuated, and those outside that range are removed. By the law of convolution, multiplication by this ideal filter-response function in frequency space is equivalent to convolution with its Fourier transformation in time. An idealised filter with a cut-off frequency of $\pm(1/\tau)$ is realised by convolution with

$$h_{ideal}(n) = \frac{\sin(n\pi/\tau)}{n\pi/\tau},$$

which, of course, stretches infinitely along the ordinate axis in both directions. Shifting this ideal response away from the origin in frequency is achieved by the multiplication by a complex exponential in time; this is the tenet of the Fourier transformation. Thus a bank of equally spaced ideal filters $X_k(n)$, representing the frequency-space transform of $x(t)h_{ideal}(n-t)$ (where $x(t)$ is the time-varying input signal) can be realised by a summation of the infinite series

$$X_k = \sum_{s=-\infty}^{\infty} x(n+s)h(-s)W_n^{-(n+s)k}$$

where $W_n = e^{j(2\pi/N)}$ and with the substitution $s = t - n$ having been made. Portnoff goes on to show by mathematical manipulation that this may be realised efficiently using well-understood Fourier transformation algorithms.

Since a profound understanding of the implications and effects of these processes is desirable before any attempt at an effective software implementation, an alternative analysis of the effects of lengthening the sampled-data window and a graphical presentation follow.

Restating the problem: starting with a larger number of discrete Fourier transformation channels of poor frequency response, it is desired to exchange some plurality of analysis channels for improved flatness of the pass-band within a channel, resulting in a smaller number of channels of better frequency response. Portnoff has achieved this with a two-fold manipulation in the time domain: aliasing and windowing by a $\frac{\sin x}{x}$ function. Recalling that the Fourier transform is its own inverse, and that aliasing in frequency is a result of undersampling in time, an intuitive grasp of the purpose of the time-aliasing is possible. The effect of the

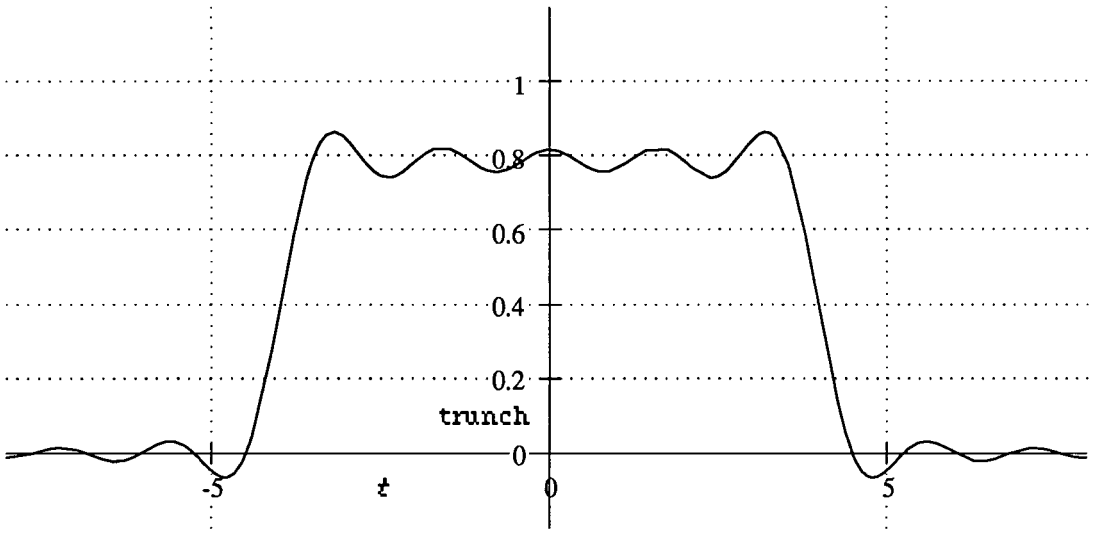


Figure 13: Graph of a Band-limited Top-hat Function

time-aliasing is to reduce the number of analysis channels, whilst maintaining the sharpness of response of each channel. Considering as an example a 1024-point transform with four-fold aliasing, the result would contain only 256 complex frequency samples, but the response of each of these filters would be as sharp as a filter in the full 1024-point transform. The 256 filters in the time-aliased transform have higher quality factor (Q) than a 256-point transform without aliasing. It is now required to widen the response of these filters without decreasing the sharpness of transition between the pass-band and the stop-bands.

A method of increasing the bandwidth of each filter without such a reduction would be to convolve the frequency response with a 'top-hat' function:

$$\mathrm{hat}_\lambda(x) = \begin{cases} 1 & -\lambda < x < \lambda \\ 0 & \text{otherwise} \end{cases}$$

$\mathrm{hat}_1(x)$ is referred to simply as $\mathrm{hat}(x)$, and $\mathrm{hat}_t(x) = \mathrm{hat}(x/t)$. Figure 13 illustrates the effect of this convolution: this curve was obtained by calculating the band-limited transform of a hat function $(\mathrm{hat}_4(t) * \sin(4t)/4t)$.

Convolution is a computationally-intensive operation, but in this instance an equivalent effect is achieved by multiplication in the time-domain by the required hat function's inverse Fourier transformation. As stated previously, the inverse

transformation of this function is $\text{sinc}t = \frac{\sin t}{t}$, which unfortunately stretches to $\pm\infty$. In the real case, the sinc function which defines the band-pass filter response is convolved not with a perfect hat function, but with the Fourier transform of a truncated sinc function. The gain of the of the filter A_v is now of the form:

$$\begin{aligned} A_v(f) &= \text{sinc}(rf) * \mathcal{F}[\text{sinc}(rt\pi)\text{hat}_{r\pi}(t)] \\ &= \text{sinc}(rf) * \text{hat}_r(f) * \text{sinc}(rf), \end{aligned} \tag{1}$$

where $G(f) = \mathcal{F}g(t)$ is the Fourier transform of some function g, and the binary operator $*$ denotes convolution. The integer r is the rarefication factor: $r = 8$ indicates that there are one eighth as many bins in the final transform as there are input data points, $r = 2$ indicates half as many, and so forth.

Before constructing a final graphical representation of these responses for various r, it may be noted that some simplification is possible.

$$\begin{aligned} A_v(f) &= \text{sinc}(rf) * \text{hat}_r(f) * \text{sinc}(rf) \\ &= \text{hat}_r(f) * [\text{sinc}(rf) * \text{sinc}(rf)] \\ &= \text{hat}_r(f) * [\mathcal{F}\text{hat}^2_{r\pi}(t)] \\ &= \text{hat}_r(f) * \text{sinc}(rf) \qquad (\text{hat}^2(x) = \text{hat}(x)) \\ &= \int_{-\infty}^{\infty} \text{sinc}(r\tau)\text{hat}_r(f-\tau)\,d\tau \\ &= \int_{-r-f}^{r-f} \text{sinc}(r\tau)\,d\tau. \end{aligned} \tag{2}$$

Except for the limits of integration, this is identical to the sine integral function $\text{si}(x) = \int_0^x \text{sinc}(\tau)\,d\tau$; the three-term convolution in equation 1 may therefore be expressed by re-arranging in integral form equation 2 and then applying

$$A_v(f) = \text{si}(r-f) - \text{si}(-r-f). \tag{3}$$

The analytical solution of this integral is possible using Maclaurin's series:

$$\begin{aligned} \sin(x) &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - + \ldots \qquad \text{for all x} \\ \Rightarrow \quad \frac{\sin(x)}{x} &= 1 - \frac{x^2}{3!} + \frac{x^4}{5!} - + \ldots \\ \text{whence} \quad \text{si}(q) &= \left[x - \frac{x^3}{3(3!)} + \frac{x^5}{5(5!)} - + \ldots \right]_0^q \end{aligned} \tag{4}$$

More generally, it is possible to evaluate to any degree of accuracy the frequency response of the prototype filter with r-fold rarefication by direct substitution using (2), (3) and (4) above:

$${}_rA_v(f) = \left[\sum_{n \geq 0} \frac{(rq)^{2n+1}}{(2n+1)(2n+1)!} (-1)^n \right]_{q=-r-f}^{r-f} \tag{5}$$

The family of curves thus obtained is presented in figure 14; it is clear that the response of the Fourier bins improves as the number of input samples available to the transform before time-aliasing increases. The number of alias points starts with $r = 1$ to the left of the figure, increasing to $r = 6$ on the right.

Before moving on from the theoretical summary, it is worth mentioning a further advantage of the frequency-space intermediate signal representation available when using the phase vocoder as a gestural capture device. Psychoacoustics has investigated the characteristics of the human ear, and has demonstrated its limitations with respect to masking and critical bandwidth.[86] Terhardt's modification of the Fourier transformation data by exponential windowing[87] has been shown by Heinbach[88] to simulate the *Auswerteinterval* (the characteristic pattern sampling interval) of the human ear; Schlang & Mummert have shown how a second-order windowing function can mimic the ear still more accurately.[89] It may become possible to use these results at a later date to impart some attribute of human perception to the gestural capture process.

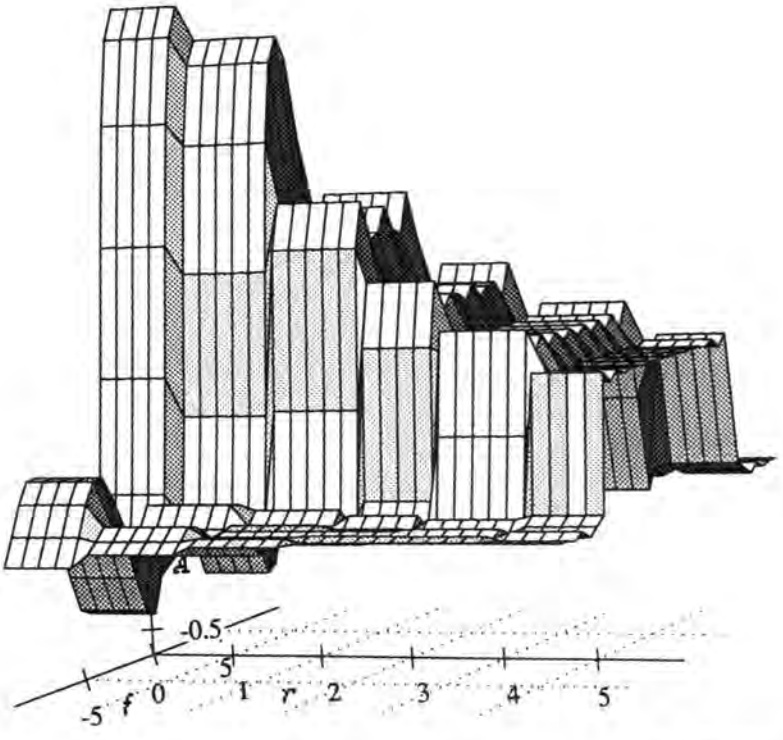


Figure 14: Frequency Response of a Fourier Bin for Increasing r

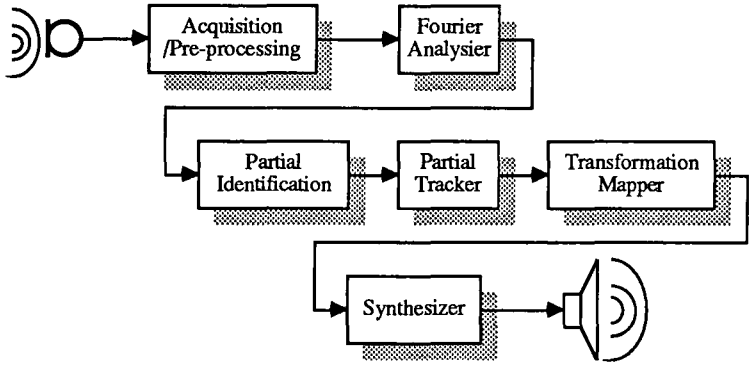


Figure 15: Communicating Sequential Processes in the Occam Phase Vocoder

8.6 Phase Vocoders in a Control Applications

The system chosen to support the implementation of a real-time phase vocoder control system is the ternary X-tree developed for audio-rate signal processing and control application.[90] This topology is conceptually similar to the Berkley X-tree[91] but is of third order, rather than binary. It is fully described in the following chapter. As the machine uses INMOS transputers as processing elements, there is direct support at the hardware level for the C.S.P.[2] model of concurrent computation and message passing. Using the Occam programming language, it is possible to define and code a concurrent program using only one processor, and then to delegate the constituent parts to the processors in the transputer array after verification. The laws of Occam are designed to ensure that the execution of a concurrent program on a single, multi-tasking, processor is logically equivalent to its execution on a multi-processor array. Program design and verification is therefore possible using relatively inexpensive hardware comprising a single processor with sufficient memory to simulate the entire target network.

Figure 15 shows how the highest level data-flow specification of the problem maps directly onto a CSP model. The rest of this section presents a more detailed description of each module.

[2]Communicating Sequential Process

8.7 Acquisition, Pre-processing and Analysis

The controlling data for this system is collected from a microphone via suitable digital-to-analogue converters. The continuous data stream must be divided into a series of overlapping frames, with the length of the frame and the overlap defined by easily-alterable compile-time constants. Pre-processing of this frame consists of the time-aliasing and windowing procedure already defined. The number of time-aliased points relates the length of the frame (measured in samples) to the number of bins in the frequency analysis. It is important to specify sufficient bins to guarantee the capture of the audibly significant partials for the lowest frequency signal of interest, but too great a number makes it impossible to maintain sufficient time-aliasing for the frequency response of each bin to be close to the ideal. Increasing the number of bins permits the partials to lie close together in frequency, but at the penalty of increasing the number of input data samples required before calculating the frequency transform of each frame. Increasing the rarefication factor improves the response of each individual filter, but is similarly penalised. In either case, the effects are undesirable because an increase in the frame length results in greater delay between the onset of the control signal and synthesiser control parameters becoming available; too great a delay will make the difference between real-time and non-real-time response.

8.8 Partial Selection and Parametric Transformation

The result of the analysis is an array of frequency components, with one complex number representing the portion of the input signal present in each frequency band. There may be a large number of such values; 512- or 1024-point Fourier transforms are quite feasible in real-time, especially where multiple processors are available. In order that this intermediate representation of the input signal can

be manipulated in an intuitive way, the data must be converted from complex-frequency to amplitude/phase format. This is achieved by applying the familiar identities:

$$\begin{aligned} \text{Amplitude}_n &= \sqrt{\Re a_n^2 + \Im a_n^2} \quad \text{and} \\ \text{Phase}_n &= \arctan\left(\frac{\Im a_n}{\Re a_n}\right) \end{aligned}$$

where a_n is the Fourier transform result for bin n.

The initial aim was to use a look-up table to provide arbitrary transformation between the timbre of the control signal and of the synthesised signal. We are assured in this respect by Wessel,[92] who has used the concept of timbre-space to provide musically expressive manipulation in the context of additive synthesis control: using line-segment approximation as a data reduction technique, ten and fifteen points stored as ordinates in, respectively, two and three control dimensions proved to be psychoacoustically sufficient. Moving across such a timbre space for consecutive notes resulted in auditory stream segregation (for larger distances) or melodic fusion (for smaller distances); Translations in this space from different origins were also identified as consistent by listeners. To perform this isomorphism for a vector of the same dimensionality as the result of the Fourier transformation would require impractically large quantities of memory, and would also make the design of the table intractable, notwithstanding the economies possible from Carbonneau's study of the perceptual effects of data reduction.[93] The constituent partials are therefore extracted, and sorted into order of decreasing magnitude. Only those which of the highest amplitude are used; the rest are discarded. An arbitrary limit is thus placed on the dimensionality of the look-up table, which is under the control of the program designer. From results in the analysis and reconstruction of the sounds of natural instruments[94, 6, 88, 93] in conjunction with experimental work undertaken by the Durham Music Technology Group[3]

[3]Experiments to test the implementation of an additive synthesiser using a pipeline of Inmos

the required number of partials has been estimated. Using three processors, 21 oscillators with control-parameter interpolation have been implemented.

The storage of the conversion values in such a table, even when the dimensionality is as low as ten in order of magnitude, can still require very large amounts of memory if conventional hyper-cubic grids are used. However, Bowler[95] shows that large savings can be made by storing hyper-simplex grids[4] and by employing a multi-dimensional co-ordinate transformation of the orthogonal abscissæ.

8.9 Synthesis and Reproduction

A simple method for the generation of arbitrary timbres is additive synthesis, although there is no fundamental reason why other techniques should be excluded. Each entry in the look-up table of timbres may consist of a vector of control parameters for an additive synthesiser, and simple linear interpolation might be used as partials are tracked between frames. This explains how to cope with the evolution of a continuous control signal, but some difficulty may arise during transients or rapid changes at the system's input. The problem is to determine whether the presence of a partial, indicated by a significant amplitude in one of the Fourier analysis bins, is the result of the same partial having been tracked from the previous analysis frame, or of the onset of a new control event. A tracking algorithm described by Bowler[96] is used to isolate distinct new partials from those which have been carried over from an already existing input signal. Partials which do not persist for more than a given time may be safely synthesised using an inverse Fourier transformation to generate an approximation to strictly band-limited noise; appropriate amplitudes inserted into the bins with randomised phase will be perceived by the ear as a noise-like signal, because the time-sampled output will not be strongly correlated over the period required by the ear for pitch

Transputers used the reconstruction of a clarinet tone as a test example; although successful, results have not been formally presented

[4] A hyper-simplex is an n-dimensional space-filling figure in the sequence point, line, triangle, tetrahedron...

estimation. An equally suitable pseudo-random residual for addition to the main additive synthesiser output may be derived by subtractive synthesis based on a wide-band noise source.

The final signal is reproduced by unbuffered digital-to-analogue conversion followed by suitable amplification. The initial aim is to produce sufficient volume to close the control loop from the performer's action back to his ear, *via* the instrumental sound. However, there is no *prima facie* reason not to use the synthesised signal as reinforcement for the controlling acoustical source.

8.10 Suggested Refinements and Extensions

This completes the outline description of a real-time, interactive electronic machine for the performance of music. It should be emphasised that several refinements and extensions are necessary before it can be considered a viable performing instrument.

One possibility might be to provide the 'instrumental craftsman' with a diverse range of synthesis modules to produce the final output signal. Whilst this has no theoretical advantage over a sufficiently sophisticated additive synthesiser in the range of timbre available, it may ease the design of the translation map by making the connexion between the input partial set and the consequent control vector more intuitive. To take a simple example, if it were desired to provide real-time control over the timbre space defined by an FOF synthesiser, it would seem perverse to insist that the designer calculate the vast set of partial/formant information for a sufficiently large number of FOF control vectors; better to use an FOF synthesiser to generate the output than an additive one.

Whilst a satisfactory implementation would overcome the complex problem of gestural capture by employing the services of highly developed conventional instrumental skills, the electro-acoustic designer (or 'craftsman') will still require the use of a battery of as yet unspecified tools. Such tools must facilitate the construction of databases in higher dimensions; the optimum ergonomic for this

process is unclear, being in itself a suitable subject for extended research.

This detailed design needs special hardware and software for its execution; the solutions which have been adopted are described in the final chapter.

Chapter 9

Hardware and Software for a Systolic Machine

9.1 Algorithms which Exploit Concurrency

THE SUPPOSITION THAT ADDING EXTRA PROCESSING ELEMENTS to a computer proportionally reduces the processing time to solve a given problem is usually false. An additional processor brings with it an additional communication and resource management overhead; after a certain number have been assembled, a maximum benefit is accrued, and any further processors can only reduce the efficiency of the system by increasing the burden of communications.

Let W_S be the quantity of work which must be performed sequentially in a given program, and W_P the quantity which may be performed concurrently. On the assumption that a machine with n processors can perform the parallel portion of the program at n times the speed of a sequential computer, but is constrained to perform the sequential part of the program no faster, the "speed-up factor" of the n-processor machine compared with the single-processor one is given by

$$S(n) = \frac{W_S + W_P}{W_S + \frac{W_P}{n}}.$$

Because W_S, W_P and n can all be assumed to be greater than zero in a real parallel system, it is a simple matter to derive the inequality

$$S(n) \leq 1 + \frac{W_P}{W_S}.$$

On this assumption an upper limit is placed on the speed-up possible for any

given program, regardless of the number of processors available. It also follows that

$$S(n) < n,$$

since $W_S + \frac{W_P}{n} > \frac{W_S+W_P}{n}$; i.e. superunitary speed-up can never be achieved.

However, there exist certain classes of algorithms which implicitly lend themselves to being written for a machine with a highly parallel architecture, and there is experimental evidence that superunitary speed-up can be attained.[97] The possible classes of speed-up have been described by Helmbold & McDowell[54] in the following categories, in order of decreasing desirability:

Class	Characteristic
Superlinear	$\lim_{n\to\infty} S(n)/n = \infty$
Linear Superunitary	$\lim_{n\to\infty} S(n)/n > 1$
Unitary	$\lim_{n\to\infty} S(n)/n = 1$
Linear Subunitary	$\lim_{n\to\infty} S(n)/n < 1$
Sublinear	$\lim_{n\to\infty} S(n)/n = 0$

Helmbold & McDowell go on to extend the speed-up model by putting forward sufficient conditions for $S(n) > n$. In summary, it may be possible to produce superunitary speed-up by: reducing overheads in task or operating system management; increasing cache available through the introduction of an additional processor; hiding latency; and taking advantage of the variance of execution times for random algorithms. Of these, only the first three are relevant to the application under consideration.

The following paragraphs give detailed consideration to the most complex modules within the gestural capture program: the Fourier transformation process and the partial-tracking process.

9.2 The Fourier Transformation Process

Much work has been undertaken on the optimisation of the Fast Fourier Transform algorithms originally proposed by Cooley & Tukey[98] and Winograd.[99] Since the advent of Very Large Scale Integration, it has been possible to realise the Fourier Transform in hardware, as well as in software. This has given rise to two genres of optimisation techniques: those designed to increase the speed of computation on a general-purpose computer, and those designed to ease the construction of a dedicated F.F.T. calculation unit 'on a chip'. The former tend to reduce the number of calculations at the expense of storage, with regard to the inevitable slowness of a general purpose sequential computer when compared with the same algorithm 'set in silicon' (which is to say, constructed from dedicated logic elements); the latter befit the inherently concurrent nature of an integrated circuit, where, naturally, all of the constituent circuit elements operate continuously and simultaneously.

Many of the improvements intended for general purpose computation have been cited by Macnaghten & Hoare;[100] for example: calculation of factors in advance; improvements which can be made when the number of time-samples is exactly a power of two; and recursive program structure. Fay[101] uses their algorithm as a basis for an inherently parallel design written in Occam, which might be a choice for implementation. However, when a system with a large number of processors is available, the most suitable algorithm grows more similar to those intended for hardware implementation. The reason for adopting algorithms which were originally hardware oriented has to do with planning communication paths; a hardware array will tend towards *systolic* concurrency, because the clocked data flow through an integrated circuit essentially corresponds to the substitution of one of the problem's variables for time.[102] Bergland[103] gives an extensive survey and overview of implementations of the fast Fourier transform in hardware, and the application of the transform in a massively parallel environment is covered by Snyder[104, 105] and Kailath.[106] Thompson[107]

surveys a number of hardware implementations with reference to the silicon area occupied and the throughput of the circuit under consideration; this paper is particularly interesting amongst hardware-oriented ones because it describes some of the implementations as pseudo-code fragments. The efficiency of such a code structure might be called into question when being executed as an array of communicating processes since intuition would suggest that an array processor operating in lock-step would have superior performance. However, Bronson, Cansavant & Jamieson[108] have constructed a machine capable of selecting between M.I.M.D.-polled-network, S.I.M.D. and M.I.M.D.-barrier-synchronised (this last being broadly equivalent to the communication mechanism in the architecture demonstrated by the Durham University Music Technology Group). Experiments with this machine show the M.I.M.D. *modus operandi* employing barrier synchronisation to be marginally faster than S.I.M.D. operation, and significantly faster than an M.I.M.D. machine communicating by means of a polled network. Of the implementations to be considered suited to a parallel array, pipelined solutions such as those put forward by Swartzlander & Halnor,[109] Yuhang Wu,[110] or Groginsky & Works[111] might be directly applicable. The algorithm suggested by Yuhang Wu is a pipeline-parallel version of Bruun's algorithm,[112] which computes the D.F.T. of a time sequence by evaluating its Z-transform at points equally spaced around the unit circle.

Alternatively, the Arithmetic Fourier Transform (A.F.T.) arising from number theory, uses the Möbius Inversion Series to effect a frequency-space transformation. This has been extended by Reed *et al.*[113] to encompass odd and even periodic functions. The A.F.T. is more desirable than the F.F.T. in that it lends itself to parallel computation, but reconstruction error is worse than the F.F.T. for a given word length. Where accuracy is a prime concern, this may be improved at the expense of calculation time by increasing the order of interpolators used at a particular point in the analysis procedure.

Conceptually similar processes are the Number-Theoretical Transformations

(N.T.T.s) based upon the Fermat Number Transform; these have been identified as an alternative to the F.F.T. in digital signal processing, and a hardware architecture to support them is proposed by Truong *et al.*[114] If a Fermat number is chosen such that two is a root of unity, multiplications by powers of the Fermat number are achieved easily by bit-shifting. The advantages over the conventional F.F.T. algorithm are that it is possible to construct transform systems without the use of multiplications, using only integer arithmetic and introducing no rounding errors. However, a modulus operator is required, which can be just as slow as a multiplier in practice, and very large dynamic ranges are required in the intermediate stages of longer transforms.

Successful transform procedures for tree-structured arrays like the one shortly to be described include the Prime Factor Algorithms (P.F.A.) of Arambepola[115] and Kolba & Parks.[116] Whilst Aloisio, Fox, Kim & Veneziani[117] have shown them to be in some respects inferior to the standard F.F.T. when executed on a hypercube topology, it is better suited to tree-based architectures where the address calculation method is much simplified. Compare, for example, the network topology chosen by Lo, Siu, Lun & Purvis[118] with that of the University of Durham machine.[90]

9.3 Tracking Partials

After conversion to polar representation, the result of the chosen n-point Fourier transformation algorithm is a vector of couplets

$$((a_0, \phi_0), (a_1, \phi_1), \ldots, (a_{(n/2)-1}, \phi_{(n/2)-1}))$$

denoting the amplitude and phase of partials in each of $n/2$ equally spaced frequency bands. It is now required to separate the most significant partials from this vector, and to track them as they move from band to band. The list of partials so formed constitutes the input data for the look-up table converter of our instrument.

A candidate partial exists where there is a local maximum in the frequency data, which is to say $a_k > a_{k-1}$ and $a_k > a_{k+1}$ where $0 < k < (n-1)/2$. The partials thus defined are assembled into a list and passed to a precedence sorter. The dimensionality of the look-up map places an upper limit on the number of partials which can be considered in choosing the control parameters for the synthesizer. The precedence sorter implemented here retains the `partials` largest amplitude couplets, along with information on the bin number in which they appeared.

Recalling that the target algorithm demands a fine-grain, systolic concurrency, the traditional sorting algorithms seem less suitable. Such sorters tend to require that the whole data set be maintained in the processor's local memory, which leaves little space for other code and data in a machine with limited RAM per node. Taking advantage of Occam's facility of expression for concurrent programs, it is possible to write code which is easily distributed across many processors using only small local data space. The model precedence sorter shown in the Occam code below is worthy of note because it requires that only `partials` data tuples be retained in main memory, and also that these data may be distributed across processors to cut the local memory demand still further. For example, in a working system, 512 amplitudes and phases may be produced for each analysis frame, and the number of partials which may be used as control parameters may be limited to approximately twenty. This program is very similar to the sort program example given in the Occam Programmers' Manual;[119] it defines `partials` concurrent tasks connected in a pipeline. Data arriving at a process is compared with the data already stored; the smaller of the two is passed to the next process in the chain, while the larger is retained for future comparison. After the entire list of results has been sent into the pipeline, a `Flush` token is sent. This causes the process which receives it to emit its stored value followed by the `Flush` token, and then to reset to its initial state. Hence the data stored in the pipeline appears at its end, the smallest value first.

```
--{{{  Sort partials
```

```
[partials][2]REAL32 biggest :          -- Array of partials to be considered
[partials][2]REAL32 results :          -- Final set of largest amplitude ptls
[partials]INT16 result.bins :          -- Bin number of above partials
[partials+1]CHAN OF Sort.Data sieve :-- Communications pipeline for sieve
[partials]BOOL run.sieve :             -- Termination flags for sieve
BOOL running :
SEQ
  ... initialise

  [2]REAL32 new.partial :
  INT16 bin :
  PAR

    WHILE running
      ALT
        to.sorter ? CASE
          Partial ; new.partial ; bin
            BOOL scratch :
            [2]REAL32 reject :
            INT16 rejbin :
            PAR
              sieve[0] ! Partial ; new.partial ; bin
              ... read and discard rejected partial
          Flush ; running
            PAR
              sieve[0] ! Flush ; running
              SEQ
                ... Read result from sieve sort
                from.sorter ! results ; result.bins ; running

    --{{{ seive sort partials
    [partials][2]REAL32 new.partial :
    [partials]INT16 bin, new.bin :
    PAR i = 0 FOR partials
      WHILE (run.sieve[i])
        ALT
          sieve[i] ? CASE
            Partial ; new.partial[i] ; new.bin[i]
              IF
                (new.partial[i][0] > biggest[i][0])
                  SEQ
                    sieve[i+1] ! Partial ; biggest[i] ; bin[i]
                    biggest[i] := new.partial[i]
                    bin[i] := new.bin[i]
                TRUE
                  sieve[i+1] ! Partial ; new.partial[i] ; new.bin[i]
            Flush ; run.sieve[i]
              SEQ
                sieve[i+1] ! Partial ; biggest[i] ; bin[i]
                biggest[i] := [ 0.0(REAL32), 0.0(REAL32) ]
                bin[i] := 0(INT16)
                sieve[i+1] ! Flush ; run.sieve[i]
    --}}}
```

Candidate partials are received down the channel `to.sorter`. An additional boolean variable is also read; this indicates whether the master process has further frames to analyse, or whether the processes should terminate. This parallel program is shown schematically in figure 16.

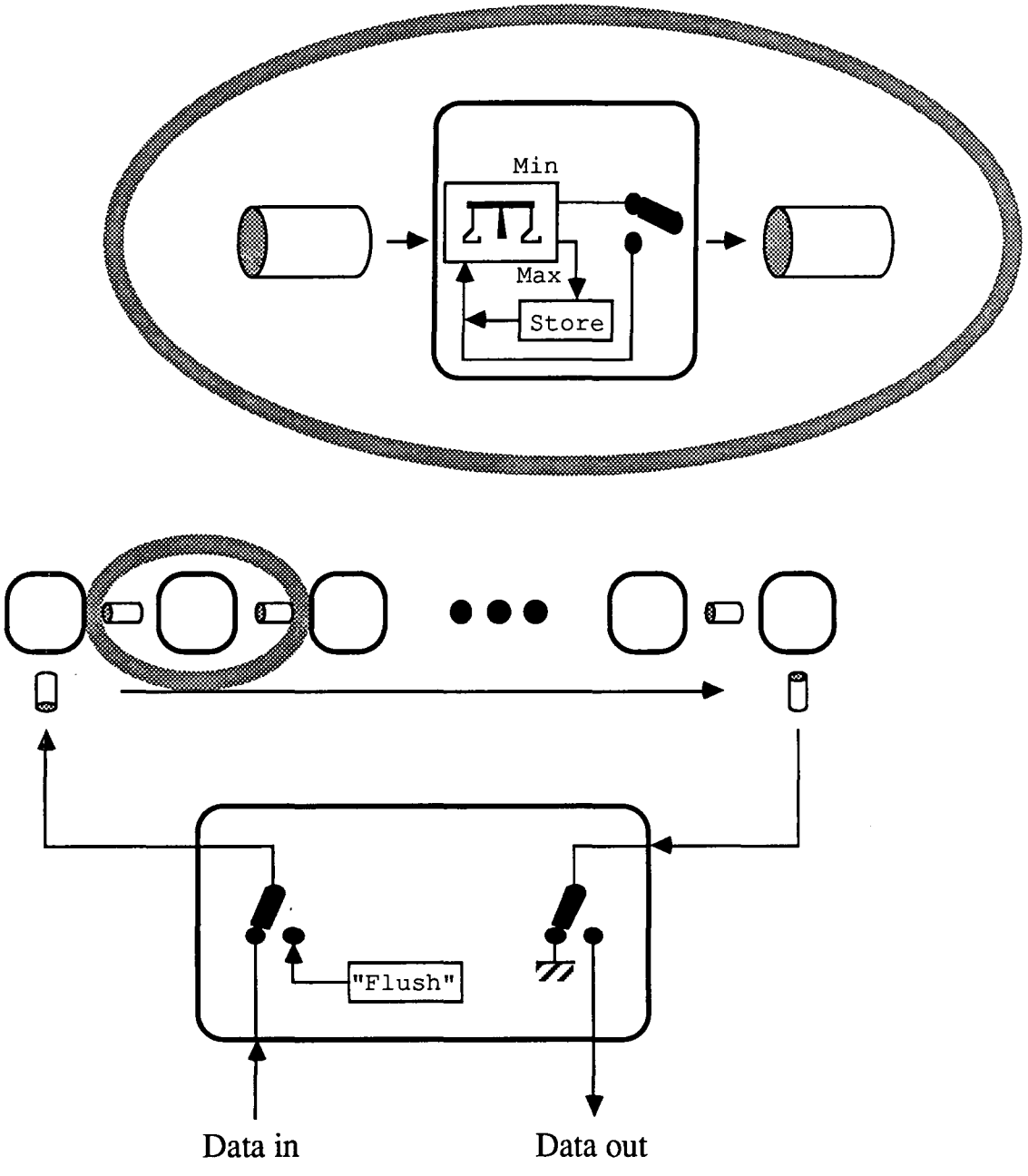


Figure 16: Schematic Diagram of Parallel Sort Program

9.4 Adjacency Testing

Having constructed a list containing the partials of greatest amplitude, an algorithm is required which will retain the continuity between analysis frames. It is necessary that the ordering of the partials in a control block is retained between frames, ignoring their amplitudes, as it may be the case that the result from the partial tracking software subsequently undergoes interpolation at a rate higher that the analysis frame rate. A simple resynthesis stage might perform linear interpolation between successive amplitude and frequency results; if the partials did not retain their position in the output vector from the analyser, but instead changed position within that vector according to their relative magnitudes, an attempt to perform element-wise interpolation between the successive analysis frames would give invalid results.

Let the output vector $\vec{a}$ contain elements describing the (at most `partials`) most significant partials at frame Φ. Frame $\Phi + 1$ yields a new vector of partials $\vec{a}'$ ordered by amplitude, to be reordered so that $\vec{a}'$ represents the partial-by-partial evolution of the elements of $\vec{a}$ up until the time of analysis window $\Phi + 1$. A new partial represents an evolution of an old partial if the new partial occupies a bin which is the same as or adjacent to the bin occupied by the old partial. Since the presence of a partial is detected by a local maximum in the frequency spectrum of the signal undergoing the Fourier transformation, it is impossible that there can be two partials occupying adjacent bins in the same analysis frame. It is possible, however, that an element of $\vec{a}$ occupying bin n disappears, and that $\vec{a}'$ has elements at bins $n + 1$ and $n - 1$. In this case, the element of $\vec{a}$ is deemed to have moved up or down one bin, according to whether the vanishing element is above or below the centre frequency for bin n. A better algorithm for this eventuality would be to take account of the difference in phase between the old partial and the new candidate partials, making a choice based upon the smallest change of phase. However this is an heuristic which may be difficult to justify without reference to the movement of the surrounding partials, which would lead

to a program of unacceptable complexity.

$\vec{a}'$ is decomposed into two lists: `create`, being a list of partials which do not match any of the elements of $\vec{a}$ and are *aspirants* awaiting entry in the current list should room become available; and `maintain`, which contains those partials which do match with an element of $\vec{a}$. A third list, `delete`, is also generated: this contains all of those partials which appear as elements of $\vec{a}$ which are no longer represented in $\vec{a}'$. If the number of partials thus nominated exceeds the globally defined maximum (i.e. the number now under consideration exceeds the number of dimensions of the subsequent mapping process), those of smaller amplitude from the `create` and `maintain` lists are removed and marked for deletion by addition to the `delete` list.

The data structures used in the partial tracking algorithm are shown in figure 17. Pointers are maintained into alternative register sets for the current and next data sets, and these are alternated for each iteration of an analysis window. The three lists described previously are stored as simple array/counter structures. The only potential problem arising is to find a method by which the adjacency test can be applied efficiently. A solution is found by considering a partials set rather than a list. We define:

$$\begin{aligned} \mathbf{A} &= \{p : p \in \vec{a}\} \\ \mathbf{P} &= \{q : q \in \vec{a}'\}. \end{aligned}$$

This enables the re-specification of the tracking algorithm in the following terms:

$$\begin{aligned} \mathbf{M} &= \mathbf{P} \cap \mathbf{A} \\ \mathbf{C} &= \mathbf{P} - \mathbf{A} \\ \mathbf{D} &= (\mathbf{A} - \mathbf{P}) \cup \mathbf{S} \end{aligned}$$

where $\mathbf{S}$ is the set formed by removing the element of smallest magnitude from $\mathbf{M} \cup \mathbf{C}$ while $n(\mathbf{M} \cup \mathbf{C}) > d$, the order of the control vector for the mapping and resynthesis unit.

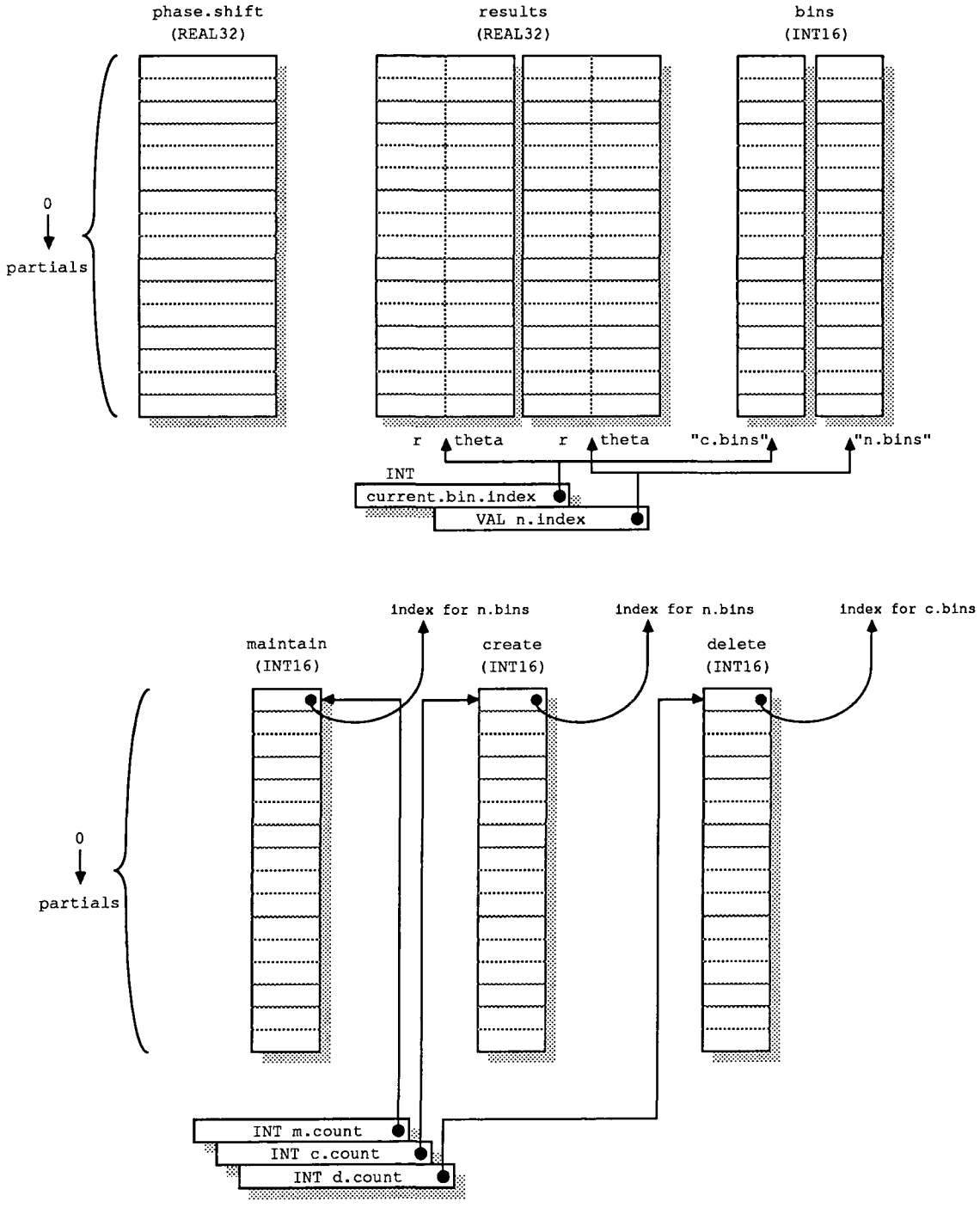


Figure 17: Data Structures in the Occam Partials Tracker

Set representation as a data structure within a computer system normally relies upon the use of bit-fields. Sufficient bits are reserved for the maximum number of elements of the set: a set bit represents the presence of the corresponding element; a reset bit represents its absence. Although the data type 'set' is not supported directly by the Occam programming language, the use of this convention enables efficient storage of sets and also enables the adjacency criteria to be applied easily. Consider the following code fragment from the partial tracking program:

```
--{{{  Sort partials into three lists

-- Each of the partials under consideration is entered into the
-- appropriate list (stored as an array):
--
--   maintain - the partial is tracked from the previous P.V.
--              frame;
--
--   delete   - no partial exists in this bin where it previously
--              existed in this bin or an adjacent bin;
--
--   create   - no partial previously existed in this bin or an
--              adjacent bin, but a new partial of significant
--              amplitude has been born.
--
-- It is important that len(maintain) + len(create) - len(delete)
--                                        <= partials !

BOOL present :
SEQ
  write.char(screen, '{')
  sc.tab IS score.tab[current.bin.index] :
  SEQ i = 0 FOR partials
    SEQ
      --{{{  Check for adjacency - result in BOOL present
      --
      --  Test the score table of the previous frame for the presence of
      --  of a partial in this or adjacent bins.  If one is identified
      --  then remove it, and set the BOOL variable to true.  In the
      --  event of a partial being between two existing partials, the
      --  lower partial is matched if the phase lags, and the upper if
      --  it leads.
      --
      IF
        n.bins[i] = 0(INT16)
          present := FALSE
        TRUE
          INT word, displacement :
          SEQ
            word := ((INT n.bins[i]) >> 5)
            displacement := (((INT n.bins[i]) - 1) /\ 31)
            IF
              displacement = 31
                word := word - 1
              TRUE
                SKIP
            VAL bit IS INT32 (7(INT32) << displacement) :
```

```
INT32 result :
SEQ
  IF
    (bit = #80000000(INT32))
      result := (((sc.tab[word] /\ bit) >> 31) \/
                 ((sc.tab[word+1] /\ 3(INT32)) << 1))
    (bit = #C0000000(INT32))
      result := (((sc.tab[word] /\ bit) >> 30) \/
                 ((sc.tab[word+1] /\ 1(INT32)) << 2))
    TRUE
      result := ((sc.tab[word] /\ bit) >> displacement)
  present := result <> 0(INT32)
```

An array `score.tab` of thirty-two-bit integers is dimensioned so that it provides sufficient contiguous memory to hold a bit-field representing the result-bins of the Fourier transformation. The prototype program uses a 1024-point F.F.T., which yields 512 amplitude and frequency values. 512 bits occupies 16 long words of memory. The test for adjacency to a partial in bin n is performed by constructing a bit-mask ("`bit`") consisting of binary 111 shifted left $n$mod32 times. When a bitwise logical-and operation is performed between the mask and appropriate word of the `score.tab` array, the result is non-zero if a partial is already identified as occupying the same or an adjacent bin in the previous analysis frame. 'The appropriate word' means that word which is indexed by the integer result of dividing the bin number by the word length in bits; this ensures that the array is treated as a continuous 512-bit field. The final conditional statements ensure that an overflow from the left-shift of the three-bit mask is trapped, and the necessary test made on the next word of the array.

The Boolean variable `present` is now set if the partial currently being considered existed in the previous analysis frame. The generation of new control block requires that the flag is now reset and the appropriate partial added to the `maintain` list. A subsequent operation will test build the list of partials due for deletion from the flags remaining in the score table. Identified partials which are not represented in the previous frame are added to the `create` list. At the end of the loop, the new score table (`score.tab[n.index]`) is built, which will provide the comparison data for the next iteration of the main program.

```
--{{{  Delete partial (if matched) from score table
INT matched :
```

```
        SEQ
          IF
            NOT present
              SKIP
            result = 4(INT32)
              matched := ((INT n.bins[i]) + 1)
            result = 2(INT32)
              matched := INT n.bins[i]
            result = 1(INT32)
              matched := ((INT n.bins[i]) - 1)
            result = 5(INT32)
              IF
                results[n.index][i][1] > 0.0(REAL32)
                  matched := ((INT n.bins[i]) + 1)
                TRUE
                  matched := ((INT n.bins[i]) - 1)
          VAL word IS INT (matched >> 5) :
          VAL bit IS INT32 (1(INT32) << (matched /\bsl\ 31)) :
          sc.tab[word] := (sc.tab[word] /\bsl\ (~bit))
        --}}}
--}}}
IF
  present
    SEQ  -- Partial carried over from last frame
      maintain[m.count] := INT16 i
      m.count := m.count + 1
  TRUE
    SEQ  -- If not, must be a new partial
      create[c.count] := INT16 i
      c.count := c.count + 1
-- {{{  Insert this partial into the bit-table of new partials
-- One of the arrays '[partials]INT16 bins' contains the bin numbers
-- corresponding to the oscillator frequencies required at the
-- beginning of this analysis frame.  This routine bilds a bit-wise
-- set of the incoming bins, which will enable a rapid test of adjacency
-- to the bins in the other frame.  The result is written in the
-- word array 'score.tab[n.index]'
SEQ
  IF
    n.bins[i] > 0(INT16)
      VAL bit IS INT32 (1(INT32) << ((INT n.bins[i]) /\bsl\ 31)) :
      word IS score.tab[n.index][(INT n.bins[i]) >> 5] :
      word := word \bsl/ bit
    TRUE
      SKIP
--}}}
```

It can be seen that care has been taken to reduce memory requirement rather than to achieve absolutely optimal speed of execution. The final sections, describing the target machine, will demonstrate why this is the case.

9.5 A Multi-processor Architecture

Real-time digital signal processing and musical applications place unconventional demands on a computing system, by requiring very high computational speed in

preference to large storage. A novel architecture based upon a *minimum memory* principle is used for this machine. It will be shown that more general-purpose applications can still be run, but that these require a substantially modified approach from that taken with von-Neumann machines; computation becomes relatively inexpensive, while memory intensive structures such as look-up tables become less feasible.

The non-real-time programmer is free to use large storage structures and introduce algorithms with unpredictable and largely varying execution times; provided the average computation time required to produce a sample remains reasonable, the mass storage device used as an intermediate storage for the output will absorb local variations. A characteristic of real-time machines is that they absorb a very large quantity of information *via* video or audio input, and reduce this information to a much more limited bandwidth which may be used directly for the control of the synthesis programs already described. They must also perform this operation at the greatest possible speed, since if the delays between applying the input stimulus and obtaining the result become too long, the control loop is no longer closed, and the machine can no longer be considered 'real-time'.

The virtuoso pianist employs many subtleties of technique called variously 'fingering', 'touch' etc.[120] to produce the desired effect. The mechanical mechanism of the piano is such that the hammer is moving ballistically when it strikes the string; it therefore reduces all of this high-level performance data by a process of mechanical filtering to five parameters: strike time, strike rate, terminal key velocity, release time, and release rate. This information is transmitted to the sounding part of the instrument, where it is used to stimulate a complex network of subtly inter-connected resonators and produce the piano sound. This whole process of high-level, high-bandwidth control, reduced to a lower bandwidth internal representation, and finally used to produce a complex output has been evolved within the electronic instrument domain. The problem arises of how to devise a computer which is naturally suited to such processes.

Hierarchical structures have been popular in the literature, as it is observed

that the event and control rates, being different from the audio data rate by orders of magnitude in conventional patched note-list event systems, might usefully be handled by a few processors controlling a larger number.[121] More recently, the emphasis seems to have shifted towards distributed real-time hardware without hierarchical control, preferring instead to delegate specific parts of an algorithm to digital signal processing microprocessors;[122] this is often referred to as the 'hardware subroutine' approach. Much material has been published concerning the best 'general purpose topology' for a parallel computer,[123] but results so far appear to be highly application dependent and largely heuristically based.

The strict data-movement discipline of real-time electroacoustic synthesis may mean that the architectures which are chosen for it are also relevant in neural network simulation. Some work has been undertaken in the application of neural networks to topics in music technology,[124, 125, 126] but this field is still insufficiently understood to tell if it is capable of the type of higher dimensional operations which are required. Work in non-linear system identification using neural networks[127] suggests that this sort of mapping is indeed possible, but that the quantity of training data required becomes prohibitive as the number of dimensions increases.

The suggestion that a systolic mode of operation is inefficient when data availability determines the flow of program control[128] relies on the assumption that the overhead in context switching on an event is high; this is not the case with a transputer. This does not suggest that the choice of topology is unimportant or non-application-specific, however. One solution might be to build a computer which is totally configurable, rather along the lines of the configurable, highly parallel (CHiP) computer due to Snyder,[129] for which code placement and memory usage heuristics are already beginning to be understood.[130] If only a small quantity of memory is provided at each node, however, the routeing algorithm for a machine which is abitrarily reconfigurable might represent a considerable overhead as a proportion of total system resource. Since storage implies

delay, which is certainly undesirable in a real-time system, and may be indicative of a poorly conceived algorithm in a systolic environment, a fixed topology was preferred. Tanimoto[131, 132] has already demonstrated that a fixed, hierarchical topology is appropriate in the processing of two-dimensional data (such as video images); this pyrimidal structure was eventually modified to be more suitable for the one-dimensional data manipulated by the music technologist. A competing architecture, the 'perfect shuffle',[133] where the connexion pattern resembles the order of cards in a perfectly shuffled deck, is attractive in certain applications; Fourier transformation[134] is particularly well suited to this configuration. However, a significant part of the implementation of an electronic musical instrument consists of the control algorithm, and it is not clear that the perfect shuffle structure would be particularly suited to this.

The first part of this thesis has shown that an efficient architecture for DSP and synthesis is a pipeline of processors with additional control data routes being made available. This structure has the property that audio samples pass down the pipeline at a constant rate regardless of the number of processors, and that the addition of further processors increases the control bandwidth linearly with the number of processors.

Using Inmos transputers, each processor has available four hardware communications links which permit the construction of a ternary tree, providing hierarchical control. Whilst the tree structure provides short path lengths between arbitrary nodes, the algorithms for which the machine is designed tend to use hierarchical data structures. This gives rise to a requirement for horizontal communication across the tree, between siblings at the same level. The fundamental element actually employed by the machine is therefore a modified version of the basic ternary tree; atoms of its structure is shown in figure 18 by darker shading. This configuration combines the advantages of a tree structure (hierarchical control; increasing control bandwidth with additional processors) with those of a pipeline (a lack of communications traffic 'hot-spots'; trivial routeing algorithm). Figure 18 illustrates the inter-connection of such modules to provide

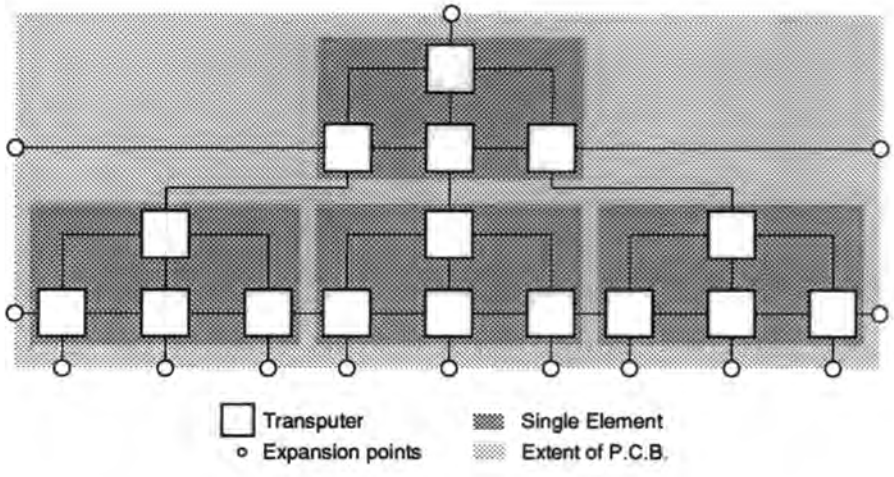


Figure 18: Basic Topology of the Transputer Tree

a tree/pipeline network. Four of these modules, a total of 16 processors, can be accommodated on each 3U Printed Circuit Board in the prototype machine; this represents 140MIPs processing, and 560Mbs^{-1} (peak) off-board communications capacity, using the less expensive 17 · 5MHz parts. The reason that such a high processing density can be achieved is that no external RAM is fitted; each Transputer uses only its internal 4KB of RAM for data and program storage. Our initial prototype comprises ten such boards, making a 1400MIP system. This quantity of memory may be considered impractically small, but the approach was encouraged by the results of van Renterghem,[135] who shows that such an array is useful in finite element analysis.

9.6 Installing Highly Parallel Algorithms

The absence of large quantities of RAM attached to each processor radically affects the approach to programming, but need not necessarily reduce the machine's functionality. If the full parallel potential were to be realised, then each PCB could be regarded as a single 140MIP processor with 64KB of zero wait-state memory. The restriction falls upon the quantity of memory available locally (a

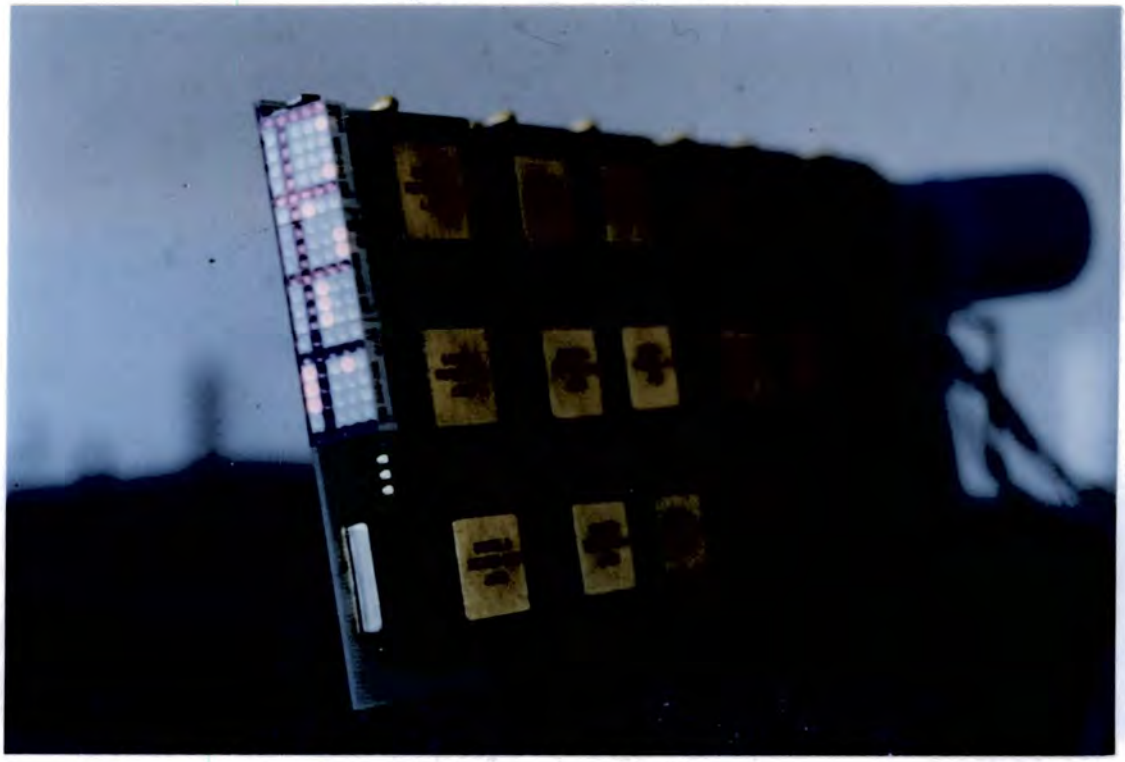
relatively small 4KB), which may force the partitioning of tasks which demand larger quantities across several processors.

A class of algorithm which tends to require a large quantity of memory is those using look-up tables. These usually submit to one of three approaches. If the look-up table is providing values for a mathematically determined function (as would be the case in a digital oscillator), an evaluation may be considered as an alternative, with several processors being used to provide for the greater computational demand. A second possibility is to distribute the lookup over a pipeline, with only a small portion of the table stored at each processor. Data to be converted are passed down the pipeline, tagged as 'Domain' data. If the current processor has available locally the necessary information to perform a transformation, this is done and the data tagged as 'Range'. By the time a block has emerged from the pipeline, all of the data should have been converted, and any items still tagged 'Domain' must have unknown or illegal values. The third approach, and one which has been used in the construction of digital oscillators,[136, 137] is to omit a large quantity of the data in the look-up table, using instead an interpolation function to provide intermediate points. Similarly, programs with larger data structures are often decomposable into smaller subproblems; consider digital interpolator and filter decomposition[138] or the partitioning of larger matricies for processing by a smaller systolic array. Other special algorithms for highly parallel machines are also documented; Akl & Meijer[139] have published a parallel binary search algorithm for a machine with exclusive-read-exclusive-write memory, such as the design presented here, which actually outperforms a concurrent-read-exclusive-write system, the most usual shared memory paradigm.

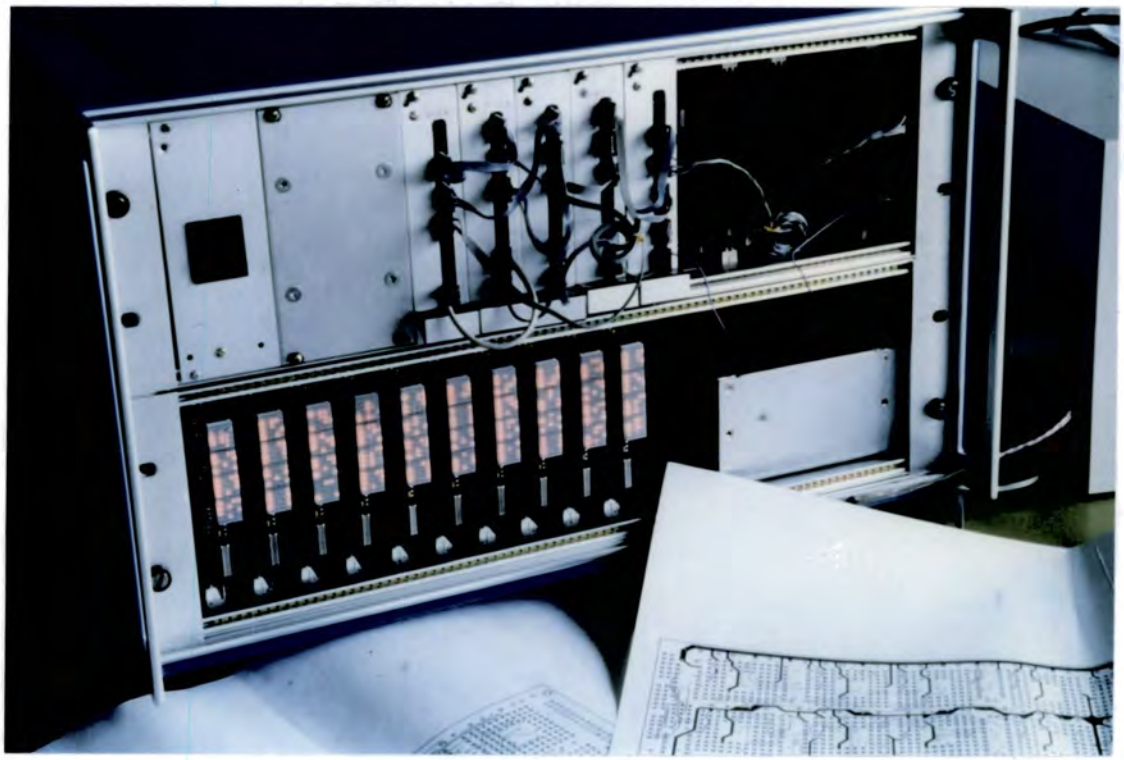
The construction and initial testing of this computer are now complete, and work is required to distribute the algorithms already described amongst its processors. The plate shows an early prototype and the final machine running a network test program which saturates the communication links with pseudo-random data, and ensures that acceptable bit error rates are achieved.

The Transputer-tree Machine

Prototype board with 16 transputers
running a network testing program



Lower half of case occupied by ten
professionally manufactured circuit boards
running the same network test program



Chapter 10

Conclusion

COMPUTER MUSIC is a discipline which demands very high performance machines. Even with the highest-speed processors of 1991, the synthesis and manipulation of high fidelity sounds in real-time requires more processing power than is available from a single, general-purpose microprocessor.

Three different problems have been examined with a view to solving them through the use of an array of transputers. The transputers used were developed by Inmos, as these are the only monolithic ones currently available. Each solution is representative of a different class of parallel programs. The first, the general direct synthesis of audio signals, runs efficiently using a isonomic software architecture; each processor uses an identical copy of the program, with partitioned input data sets describing the required result. The final result of this system is gained by combining the results of each of the programs. The second algorithm to be recoded for concurrent operation runs most efficiently on a dianomic architecture; the processing 'hot-spot' of the phase vocoder program, the forward and reverse Fourier transformations, are delegated to a task force of 'worker' processors by a 'master' process residing on the root node of the network. Thus all transfers to and from the host file system are naturally concentrated at one node, avoiding the potential bottleneck of multiple processor disk accessing, which was one of the restricting attributes in the first case.

Having demonstrated two different approaches to parallel software configuration as working parallel systems which are in day-to-day use 'in the field', attention was turned to developing the hardware and software required to improve the

utility of computers in electroacoustic music. Present systems suffer from the requirement of a certain level of technical knowledge. This prohibits some musicians from using them, as traditional musical expertise demands no such ability. The construction of a real-time performing instrument which is capable of transforming gestural inputs into arbitrary sound outputs benefits from the partitioning of the problem into three discrete areas: instrument definition; musical composition; and live performance. So that a new and therefore unknown human-computer interface is not introduced, a machine which is potentially capable of capturing the gestural information from a conventional musical instrument is specified. This machine, capable of sustaining almost one and a half thousand millions of instructions per second, has been built and tested. The hardware architecture used is based heuristically upon the structural requirements of the gestural capture algorithms, and although conventional, general-purpose programs cannot be run on this configuration immediately, many can be modified or rewritten to take advantage of this new hardware configuration. The finished machine provides exciting opportunities for further work in real-time algorithms for electroacoustic music.

Bibliography

[1] **Vercoe, B** CSOUND *Reference Manual* MIT Press, 1986.

[2] **Chamberlain, H** *Musical Applications of Microcomputers* Hayden Books, Howard W Sams &Co. 4300 West 62nd St., Indianapolis, Indiana 64268 USA ISBN 0–8104–5768–7 (1987 2nd Edition)

[3] **Comerford, P J, Eaglestone, B M** *Bradford Musical Instrument Simulator and Workstation* Proceedings of Euromicro '88 conference on Supercomputer Technology and Applications, Zurich. Microprocessing and Microprogramming 24 (1988), pp73–78.

[4] **Lowe, W, Currie, R** *Digidesign's Sound Accelerator: Lessons Lived and Learned* Computer Music Journal 13(1) pp36–46 (1989)

[5] **Grey, J M** *Doctoral Thesis* Psychology Department of Stanford University, USA (Feb. 1975)

[6] **Serra, Marie-Helene, Rubine, Dean, & Dannenberg, Roger D** *The Analysis and Resynthesis of Tones via Spectral Interpolation* Proceedings of the International Computer Music Conference 1988, pp322–332

[7] **Sasaki, Lawrence H, & Smith, Kenneth C** *A Simple Data Reduction Scheme for Additive Synthesis* Computer Music Journal 4(1) pp22–24.

[8] **Cook, Perry R** *Synthesis of the Singing Voice Using a Physically Parameterised Model of the Human Vocal Tract* Proceedings of the International Computer Music Conference, Ohio, 1989. pp69–72

[9] **Roads, C** *Granular Synthesis of Sounds* Computer Music Journal 2(2) pp61–62 (1978)

[10] **Truax, B** *Real-Time Granular Synthesis with a Digital Signal Processor* CMJ 12(2) Summer 1988.

[11] **Rodet, X** *Time-Domain Formant-Wave-Function Synthesis* Computer Music Journal, 8(3), 1984 pp 9–14

[12] **Rodet, X, Potard, Y, Barriere, J** *The Chant Project* Computer Music Journal, 8(3), 1984.

[13] **Rivas, D, Watkins, S, Chau, P M** *VLSI for a Physical Model of Musical Instrument Oscillations* Proceedings of the International Computer Music Conference, Ohio, 1989. pp253–256

[14] **Emmerson, Simon (Editor)** *The Language of Electroacoustic Music* Macmillan Press Ltd 1986. ISBN 0–333–39760–6.

[15] **Manning, P D** *Electronic and Computer Music* Oxford University Clarendon Press (1985) ISBN 0–19–311981–8

[16] **Roads, C** *Composers and the Computer* William Kaufmann Inc., LA, California. 1985

[17] **Zicarelli, D** *M and Jam Factory* Computer Music Journal, 11(4) (Winter 1987) pp13–29

[18] **Haus, Goffredo** *Music Processing at L.I.M.* Proceedings of Euromicro '88 conference on Supercomputer Technology and Applications, Zurich. Microprocessing and Microprogramming 24 (1988), pp 435–441.

[19] **Langston, P S** *Six Techniques for Algorithmic Music Composition* Proceedings of the International Computer Music Conference, Ohio, 1989. pp164–167

[20] **Katayose, H, Takami, K, Fukuoka, T, & Inokuchi, S** *Music Interpreter in the Kansei Music System* Proceedings of the International Computer Music Conference, Ohio, 1989. pp147–150

[21] **Katayose, H, Kato, H, Imai, M, & Inokuchi, S** *An Approach to an Artificial Music Expert* Proceedings of the International Computer Music Conference, Ohio, 1989. pp139–146

[22] **Strasburger, Hans, Kohler, Stefan, & Radauer, Irmfried** *Score Input to* CSOUND *via the MIDI Keyboard* Proceedings of the International Computer Music Conference, Glasgow, 1990, p208

[23] **McAdams, Stephen, & Bregman, Albert** *Hearing Musical Streams* Computer Music Journal 3(4) (1979) pp26–43,60,63

[24] **Haynes, Stanley** *The Musician-Machine Interface in Digital Sound Synthesis* Ph.D. Thesis, Department of Music, University of Durham

[25] **Buxton, W** *Design Issues in the Foundation of a Computer-Based Tool for Music Composition* Structured Sound Synthesis Project, Computer Systems Research Group, University of Toronto, Toronto, Ontario, Canada M5S 1A4.

[26] **Buxton, William, Sniderman, Richard, Reeves, William, Patel, Sanad, & Baeker, Ronald** *The Evolution of the SSSP Score-Editing Tools* Computer Music Journal 3(4) (1979) pp14–25

[27] **Clarke, M, Manning, P D, Berry, R, Purvis, A** *Vocel: An FOF Unit-Generator for* MUSIC11 International Computer Music Conference, Cologne, 1988.

[28] **Endrich, T** *C.D.P. Reference Manual* C.D.P. Limited, 1987.

[29] **Almasi & Gottlieb** *Highly Parallel Computing* Benjamin/Cummings Publishing, ISBN 0–8053–0177–1

[30] **Lee, Edward Ashford, & Messerschmitt, David G** *Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing* IEE Transactions on Computers Vol C-36, No 1. January 1987.

[31] **Laurson, M, & Duthen, J** *Patchwork: A Graphic Language in Preform* Proceedings of the International Computer Music Conference, Ohio, 1989. pp172–175.

[32] **Decker, S L, Kendall, G S, Schmidt, B L, Ludwig, M D, & Freed, D J** *A Modular Environment for Sound Synthesis and Composition* Computer Music Journal 10(4) pp28–40 (1986)

[33] **Tarabella, L, Bertini, G** *A Digital Signal Processing System and a Graphic Editor for Synthesis Algorithms.* Proceedings of the International Computer Music Conference, Ohio, 1989. pp312–315

[34] **Sedgewick, Robert** *Algorithms* Addison-Wesley 1983. ISBN 0–201–06672–6

[35] **Walker, W F** *KIWI: A Parallel System for Software Sound Synthesis* Proceedings of the International Computer Music Conference, Ohio, 1989. pp328–331

[36] **Flynn, Michael J.** *Some Computer Organisations and their Effectiveness* IEEE Transactions on Computers C21 pp948–960 (1972)

[37] **Bowler, I W, Manning, P D, Purvis, A, Bailey, N J** *A Transputer-Based Additive Synthesis Implementation* Proceedings of the International Computer Music Conference, Ohio 1989. pp58–61.

[38] **Gould, L, Bowler, I W, Purvis, A** *Real-Time Multi-Channel Digital Filtering on the Transputer* Submitted to Computer Architecture and Digital Signal Processing IEE Conference, Hong Kong, 1989.

[39] **Tannenbaum, A S** *Operating Systems* Prentice Hall, 1987

[40] **Dijkstra, E W** *Cooperating Sequential Processes* in 'Programming Languages' ed. Genuys, F New York Academic Press (1968) pp34–112

[41] **Dijkstra, E W** *Solution of a Problem in Concurrent Program Control* Communications of the Association of Computing Machinery, Vol. 8(5) (Sept 1965) p569.

[42] **Tayli, Murat, & Benmaiza, Mohamed** *Transputer Implementations of General Semaphores* Occam User Group Newsletter, No. 14 (January 1991) pp50–60

[43] **Knuth, D E** *The Art of Computer Programming, vol. 1, sec 2.2 pp234 et seq.* Addison-Wesley (1968)

[44] **Finkel, A, Choquet, A** *FIFO Nets Without Order Deadlock* Acta Informatica 25(1) (1986) pp15–36

[45] **Fischer, Michael J, Lynch, Nancy A, Burns, James E, & Borodin, Alan** *Distributed FIFO Allocation of Identical Resources Using Small Shared Space* ACM Transactions on Programming Languages and Systems, 11(1) pp90–114

[46] **Peterson, James L** *Petri-net Theory and the Modeling of Systems* Prentice-Hall (1981) ISBN 0–13–661983–5

[47] **Murata, T** *Petri-nets: Properties, Analysis and Applications* Proc. IEEE 77(4) (April 1989) pp 541–580

[48] **Lynch, Nancy A, & Fischer, Michael J** *A Technique for Decomposing Algorithms which use a Single Shared Variable* Journal of Computer System Science 27(3) (1983) pp350–377

[49] **Courtois, P, Heymans, F, & Parnas, D** *Concurrent Control with 'Readers' and 'Writers'* Communications of ACM 14(10) (Oct. 1971) pp667–668

[50] **Cooprider, L** *Petri-nets and the Representations of Standard Synchronisations* Dept of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania (Jan. 1976)

[51] **Peacock, J Kent** *Deadlock Avoidance in Loosley-Coupled Multiprocessors with Finite Buffer Pools* ACM Operating Systems Review 23(2) (April 1989) pp20–24

[52] **Holm, Frode** *Frequency Scheduling: Real-Time Scheduling in Multiprocessing Systems* Proceedings of the International Computer Music Conference, Ohio, 1989 pp127–130

[53] **Orlarey, Y, & Lequay, H** *MIDI Share: A Real-Time Multi Tasks Software Module for MIDI Applications* Proceedings of the International Computer Music Conference, Ohio, 1989. pp 234–237

[54] **Helmbold, David P, & McDowell, Charles** *Modeling Speedup(N) Greater Than N* IEEE Transactions on Distributed and Parallel Systems 1(2) (April 1990) pp250–256

[55] **F Richard Moore** *Elements of Computer Music* Prentice Hall, 1990. ISBN 0–13–252552–6

[56] **Allen, J B** *Short Term Spectral Analysis, Synthesis, and Modification by Discrete Fourier Transform* IEEE Transaction on Acoustics, Speech and Signal Processing, ASSP-25(3) (June 1977) pp235–238

[57] **Wishart, T** *C.D.P. Phase Vocoder Reference Manual* C.D.P. Limited, 1989.

[58] **Lent, K** *An Efficient Method of Pitch Shifting Digitally Sampled Sounds* Computer Music Journal 13(4) (Winter 1989) pp65–71

[59] **Schroeder, M R** *Linear Prediction, Entropy and Signal Analysis* IEEE ASSP Magazine, July 1984

[60] **Koikkalainen, Pasi & Sauer, Frank** *Architecture-independent Multi-computing via a self-distributing communication Harness* Information Processing Laboratory Publication, Lappeenranta University of Technology, POBox 20, SF–53851, Lappeentanta, Finland. Abstracted by 1st Nordic Transputer Seminar, Turku, Finland (1990).

[61] **3L Limited** *3L Parallel C Version 2.0, Manual* 3L Limited 1988.

[62] **Kamangar, F A, Duderstadt, R A, Smith, J O** *Implementing the Back-Propagation Algorithm on the Meiko Parallel Computing Surface* Proceedings of the International Conference on the Applications of Transputers, Liverpool, 1989. Published as 'Applications of Transputers, Volume 1', Len Freeman and Chris Phillips (ed.) IOS Press (1990) ISBN 90–5199–025–1.

[63] **3L Technical Note No. 8** *Processor Farms* ref. 02708, January 18, 1990. 3L Ltd., Peel House, Ladywell, Livingston, EH54 6AG, Scotland, U.K.

[64] **Casserley, Lawrence** *Seried Phi Real-Time Digital Signal Processor* Proceedings of the International Computer Music Conference, Glasgow, 1990, pp124–126

[65] **Kirk, R & Orton, R** *MIDAS A Musical Instrument Digital Array Signal Processor* Proceedings of the International Computer Music Conference, Glasgow, 1990, pp127–131

[66] **Foley, J D** *Interfaces for Advanced Computing* Scientific American, 1987 pp83–90

[67] **Schaeffer** *Esquisse d'un Solfège Concret* (1952).

[68] **Boulez, Pierre, & Gerzo, Andrew** *Computers in Music* Scientific American, 258(4) April 1988.

[69] **Morita, H, Ohteru, S, & Hashimoto, S** *Computer Music System which Follows a Human Conductor* Proceedings of the International Computer Music Conference, Ohio, 1989. pp207–210

[70] **Vercoe, B, & Cumming, D** *Connection Machine Tracking of Polyphonic Audio* Proceedings of the International Computer Music Conference, Cologne, 1988.

[71] **Vercoe, Barry, & Ellis, Dan** *Real-Time* CSOUND*: Software Synthesis with Sensing and Control* Proceedings of the International Computer Music Conference, Glasgow, 1990. pp209–211.

[72] **Vercoe, B, & Puckette, M** *Synthetic Rehearsal: Training the Synthetic Performer* Proceedings of the International Computer Music Conference, 1985, pp275–278

[73] **Dannenberg** *Software Techniques for Interactive Performance Systems* Proceedings of the International Workshop on Man-Machine Interaction in Live Performance, Computer Music Department CNUCE/CNR, Pisa, June 1991.

[74] **Yavelow** *Music & Microprocessors: MIDI and the State of the Art* in 'The Music Machine', ed. Roads, pp199-241, M.I.T. press, 1989. ISBN 0–262–18131–2

[75] **Baird, Bridget, Blevins, Donald, & Zahler, Noel** *The Artificially Intelligent Computer Performer on the Macintosh II and a Pattern Matching Algorithm for Real-Time Interactive Performance* Proceedings of the International Computer Music Conference, Ohio, 1989 pp13–16

[76] **Roads, C** *The Second STEIM Symposium on Interactive Composition in Live Electronic Music* Computer Music Journal, 10(1) (Summer 1986) pp44–50.

[77] **Helm, E** *Composer, Performer, Public: A Study in Communication* International Music Council; Music & Communication; 1. Florence, 1970

[78] **Boie, Bob, Mathews, Max, & Schloss, Andy** *The Radio Drum as a Synthesiser Controller* Proceedings of the International Computer Music Conference, Ohio, 1989. pp42–45.

[79] **Keane, D, & Gross, P** *The MIDI Baton* Proceedings of the International Computer Music Conference, Ohio, 1989.

[80] **Flanagan, J L, Golden, R M** *The Phase Vocoder* Bell System Technical Journal 45 pp1493–1509 (1966)

[81] **Portnoff, M R** *Implementation of the Digital Phase Vocoder using the Fast Fourier Transform* IEEE Transactions on Acoustics, Speech and Signal Processing, ASSP-24(3) (June 1976)

[82] **Dolson, M** *The Phase Vocoder: A Tutorial* Computer Music Journal 10(4) pp14–27 (1986)

[83] **Gordon, J W, Strawn, J** *An Introduction to the Phase Vocoder* in 'Digital Audio Signal Processing — an Anthology' The Computer Music and Digital Audio Series ed. Strawn J: William Kaufmann, Inc., 95 First Street, California 94022 ISBN 0–86576–082–9.

[84] **Moorer, J A** *The Use of the Phase Vocoder in Computer Music Applications* Journal of the Audio Engineering Society 26(1) pp42–45 (January/February 1978)

[85] **McGee, W F, Merkley, Paul** *A Real-Time Logarithmic-Frequency Phase Vocoder* Computer Music Journal 15(1) pp20–27 (Spring 1991)

[86] **Zwicker, E, & Terhardt, E** *Analytical Expressions for Critical-Band Rate and Critical Bandwidth as a Function of Frequency* Journal of the Accoustical Society of America, 65(5) (Nov 1980) pp1523–1525

[87] **Terhardt, E** *Fourier Transformation of Time Signals: Conceptual Revision* Acustica 57 (1985). pp242–256

[88] **Heinbach, W** *Aurally Adequate Signal Representation: The Part Tone Time Pattern* Acustica vol. 67 (1988) pp113–121

[89] **Schlang, Martin F, & Mummert, Markus** *Die Bedeutung der Fensterfunktion Für die Fourier-T-Transformation als Gehörgerechte Spektralanalyse* Lehrstuhl für Elektroakustik, TU, München

[90] **Bailey, N.J., Purvis, A., Bowler, I.W., & Manning, P.D.** *An Highly Parallel Architecture for Real-time Music Synthesis and Digital Signal Processing Application* Proceedings of the International Computer Music Conference, Glasgow, 1990. pp169–171.

[91] **Despain, Alvin M, & Patterson, David A** *X-Tree: A Tree Structured Multi-Processor Computer Architecture* Proceedings of the 5th Annual Symposium on Computer Architectures, April 1978, pp144–151

[92] **Wessel, David W** *Timbre Space as a Musical Control Structure* Computer Music Journal 3(2) (1979) pp45–52

[93] **Carbonneau, Gerard R** *Timbre and the Perceptual Effects of Three Types of Data Reduction* Computer Music Journal 5(2) pp10–19

[94] **Serra, X, Smith, J O** *Spectral Modeling Synthesis* Proceedings of the International Computer Music Conference, Ohio, 1989. pp281–284

[95] **Bowler, I W, Manning, P D, Purvis, A, Bailey, N J** *On Mapping N Articulation- onto M Synthesiser-Control Parameters* Proceedings of the International Computer Music Conference 1990, Glasgow. pp181–184

[96] **Bowler, I W, Manning, P D, Purvis, A, Bailey, N J** *New Techniques for a Real-Time Phase Vocoder* Proceedings of the International Computer Music Conference 1990, Glasgow. pp178–180

[97] **Parkinson, D** *Parallel Efficiency can be greater than Unity* Parallel Computing 3 pp261–262 (1986)

[98] **Cooley, James W, Tukey, John W** *An Algorithm for the Machine Calculation of Complex Fourier Series* Mathematics of Computation 19 (April 1965), pp297–301

[99] **Winograd, S** *On Computing the Discrete Fourier Transform* Mathematics of Computation, 32(141) (Jan 1978) pp175–199

[100] **Macnaghten, A M, Hoare, C A R** *Fast Fourier Transform free from Tears* The Computer Journal 20(1) (1977) pp78–83

[101] **Fay D Q M** *An Implementation of the Fast Fourier Transform in Occam* Computer Science and Informatics 14(2) pp3–12

[102] **Tsay, Jong-Chuang, & Yuan, Sy** *Systolic Flow* Journal of Parallel and Distributed Computing 8 (1990) pp 286–291)

[103] **Bergland, Glen D** *Fast Fourier Transform Hardware Implementations — An Overview* IEEE Transactions on Audio and Electroacoustics AU-17(2) (June 1969) pp104–108

[104] **Snyder, L** *The Role of the CHiP Computer in Signal Processing* In "VLSI and Modern Signal Processing" ed. Kung, Whitehouse and Kailath, Prentice Hall, 1985. ISBN 0–13–942699–X

[105] **Snyder, L** *Configurable, Highly Parallel (CHiP) Approach to Signal Processing Applications* SPIE Technical Symposium East 1982 Proceeding

[106] **Kailath, T** *Signal Processing Applications of Concurrent Array Processors* In 'VLSI and Modern Signal Processing' ed. Kung, Whitehouse and Kailath, Prentice Hall, 1985. ISBN 0–13–942699–X

[107] **Thompson, Clark D** *Fourier Transforms In VLSI* IEEE Transactions on Computers C-32(11) (Nov 1983) pp1047–1057

[108] **Bronson, E C, Casavant, T L, Jamieson, L H** *Experimental Application-Driven Architecture Analysis of an SIMD/MIMD Parallel Processing System* IEEE Transactions on Parallel and Distributed Systems, Vol. 1, No. 2, April 1990, pp195–205.

[109] **Swartzlander, E E, Jnr, & Halnor, G** *Frequency Domain Digital Filtering with VLSI* In 'VLSI and Modern Signal Processing" ed. Kung, Whitehouse and Kailath, Prentice Hall, 1985. ISBN 0-13-942699-X

[110] **Yuhang Wu** *New FFT Structures Based on the Bruun Algorithm* IEEE Transactions on Acoustics, Speech and Signal Processing 38(1) (Jan 1990)

[111] **Groginsky, Herbert L, & Works, George A** *A Pipeline Fast Fourier Transform* IEEE Transactions on Computers, Nov 1970, pp1015–1019

[112] **Bruun, George** *Z-Transform DFT Filters and FFTs* IEEE Transactions on Acoustics, Speech and Signal Processing ASSP26(1) (Feb 1978) pp56–63

[113] **Reed, I S, Tufts, D W, Yu, X, Truong, T K, Shih, M-T, Yin, X** *Fourier Analysis and Signal Processing by use of the Möbius Inversion Formula* IEEE Transactions on Acoustics, Speech and Signal Processing 38(3) (March 1990)

[114] **Truong, T K, Reed, I S, Yeh, C-S, Chang, J J, Shao, H M** *A Parallel VLSI Architecture for a Digital Filter using a Number-Theoretic Transform* In "VLSI AND MODERN SIGNAL PROCESSING" ed. KUNG, WHITEHOUSE AND KAILATH, Prentice Hall, 1985. ISBN 0–13–942699–X

[115] **Arambepola, B** *Discrete Fourier Transform Processor based on the Prime-Factor Algorithm* IEE Proceedings Vol 130 Part G No 4 (Aug 1983) pp138–144

[116] **Kolba, Dean P, Parks, Thomas W** *A Prime-Factor FFT Algorithm Using High-Speed Convolution* IEEE Transactions on Acoustics, Speech and Signal Processing ASSP25(4) (Aug 1977) pp218–294.

[117] **Aloisio, G., Fox, G.C., Kim, J.S., & Veneziani, N.** *A Concurrent Implementation of the Prime Factor Algorithm* IEEE Transactions on Acoustics, Speech and Signal Processing, vol 30, pp160–170.

[118] **Lo, K C, Siu, W C, Lun, D P K, Purvis, A** *Address Generation of Prime Factor Algorithm in a Multiprocessor System* Presented at IEEE Conference on Circuits and Systems, Singapore, July 1991

[119] **Inmos Ltd.** *Transputer Development System ch. 5–7* Prentice Hall ISBN 0–13–928995–X

[120] **Cadoz, C, Luciani, A, & Florens, J** *Responsive Input Devices and Sound Synthesis by Simulation of Instrumental Mechanisms: The Cordis System* Computer Music Journal, vol. 8 no. 3 (Autumn 1984).

[121] **Snell, J M** *General-Purpose High-Fidelity Affordable Real-Time Computer Music System* 1987 ICMC Proceedings, pp130–137.

[122] **Boynton, L, Cumming, D** *A Real-Time Acoustic Processing Card For The Mac-II* Proceedings of the International Computer Music Conference, 1988, pp349–356

[123] **Baude, Françoise, Carré, Françoise, Cléré, Pascal, & Vidal-Naquet, Guy** *Topologies for Large Transputer Networks: Theoretical Aspects and Experimental Approach* Research Paper: Laboratories de Marcoussis, CGE Research Center, Route de Nozay, 91460 Marcoussis, France.

[124] **Lippmann, R P** *An Introduction to Computing with Neural Nets* IEEE ASSP Magazine April 1987 pp4–22

[125] **Laden, B, Keefe, D H** *The Representation of Pitch in a Neural Net Model of Chord Classification* Computer Music Journal 13(4) (Winter 1989) pp12–26

[126] **Dolson, M** *Machine Tongues XII: Neural Networks* Computer Music Journal 13(3) (Autumn 1989) pp28–40

[127] **Chen, J R, Mars, P** *Artificial Neural Networks and Nonlinear System Identification* Internal report: School of Engineering and Applied Science, Uuniversity of Durham, South Road, Durham, United Kingdom (Jan 1990)

[128] **Loy, Gareth** *On the Scheduling of Multiple Parallel Processors Executing Synchronously* Proceedings of the International Computer Music Conference, 1987. pp117–124

[129] **Snyder, L** *Introduction to the Configurable, Highly Parallel Computer* Computer 15(1) (1982) pp47–56

[130] **Snyder, L** *Overview of the CHiP Computer* In 'VLSI81' ed. J P Gray, Academic Press, New York (1981) ISBN 0–12–296860–3

[131] **Tanimoto, S L** *A Pyramidal Approach to Parallel Processing* Proceedings of the 10th international symposium on computer architecture, Stockholm, June 1983

[132] **Tanimoto, S, Pavlidis, T** *A Hierarchical Data Structure for Picture Processing* Computer Graphics and Image Processing 4 pp104–119 (1975)

[133] **Stone, Harold S** *Parallel Processing with the Perfect Shuffle* IEEE Transactions on Computers C-20(2) (Feb 1971), pp153–161

[134] **Pease, M C** *An Adaptation of the Fast Fourier Transform for Parallel Processing* Journal of the Association of Computing Machinery 15(2) (April 1968) pp252–264

[135] **van Renterghem, P** *Applicability of a 16-Node Transputer Array Without External Memory* in 'Applying Transputer Based Parallel Machines' ed. Bakkers, A IOS Press, Netherlands, 1989. ISBN 90–5199–011–1

[136] **Rabiner, L R, Gold, B** *Theory and Application of Digital Signal Processing* Prentice-Hall International (1975) ISBN 0–13–914101–4. Section 9.12 (pp563-565)

[137] **Oppenheim, A V, Schafer, R W** *Digital Signal Processing* Prentice-Hall International (1975)

[138] **Oetken, Geerd, Parks, Thomas W, & Schuessler, Hans W** *New Results in the Design of Digital Interpolators* IEEE Transactions on Acoustics, Speech and Signal Processing, ASSP-23(3) (June 1975) pp301–309

[139] **Akl, Selim K, & Meijer, H** *Parallel Binary Search* IEEE Transactions on Parallel and Distributed Systems 1(2) (April 1990) pp247–250

Alphabetial List of References

3L Limited *3L Parallel C Version 2.0, Manual* 3L Limited 1988. First cited on page 66

3L Technical Note No. 8 *Processor Farms* ref. 02708, January 18, 1990. 3L Ltd., Peel House, Ladywell, Livingston, EH54 6AG, Scotland, U.K. First cited on page 69

Akl, Selim K, & Meijer, H *Parallel Binary Search* IEEE Transactions on Parallel and Distributed Systems 1(2) (April 1990) pp247–250 First cited on page 127

Allen, J B *Short Term Spectral Analysis, Synthesis, and Modification by Discrete Fourier Transform* IEEE Transaction on Acoustics, Speech and Signal Processing, ASSP-25(3) (June 1977) pp235–238 First cited on page 64

Almasi & Gottlieb *Highly Parallel Computing* Benjamin/Cummings Publishing, ISBN 0–8053–0177–1 First cited on page 15

Aloisio, G., Fox, G.C., Kim, J.S., & Veneziani, N. *A Concurrent Implementation of the Prime Factor Algorithm* IEEE Transactions on Acoustics, Speech and Signal Processing, vol 30, pp160–170. First cited on page 113

Amdahl, G *Validity of the single processor approach to achieving very large scale computing capabilities* Proceedings of AFIPS Computing Conference, v.30 (1967) pp 483–485

Andre-Obrecht, R *A New Statistical Approach for the Automatic Segmentation*

of Continuous Speech Signals IEEE Transactions on Acoustics, Speech and Signal Processing 36(1) (Jan 1988)

Arambepola, B *Discrete Fourier Transform Processor based on the Prime-Factor Algorithm* IEE Proceedings Vol 130 Part G No 4 (Aug 1983) pp138–144 First cited on page 113

Bailey, N.J., Purvis, A., Bowler, I.W., & Manning, P.D. *An Highly Parallel Architecture for Real-time Music Synthesis and Digital Signal Processing Application* Proceedings of the International Computer Music Conference, Glasgow, 1990. pp169–171. First cited on page 103

Baird, Bridget, Blevins, Donald, & Zahler, Noel *The Artificially Intelligent Computer Performer on the Macintosh II and a Pattern Matching Algorithm for Real-Time Interactive Performance* Proceedings of the International Computer Music Conference, Ohio, 1989 pp13–16 First cited on page 88

Baude, Françoise, Carré, Françoise, Cléré, Pascal, & Vidal-Naquet, Guy *Topologies for Large Transputer Networks: Theoretical Aspects and Experimental Approach* Research Paper: Laboratories de Marcoussis, CGE Research Center, Route de Nozay, 91460 Marcoussis, France. First cited on page 124

Bergland, Glen D *Fast Fourier Transform Hardware Implementations — An Overview* IEEE Transactions on Audio and Electroacoustics AU-17(2) (June 1969) pp104–108 First cited on page 111

Biyabani, S R, Stankovic, J A, & Ramamritham, K *The Integration of Deadline and Criticalness in Hard Real-Time Scheduling* Proceedings of the IEEE Real-Time Systems Symposium 1988 pp152–160

Boie, Bob, Mathews, Max, & Schloss, Andy *The Radio Drum as a Synthesiser Controller* Proceedings of the International Computer Music Conference, Ohio, 1989. pp42–45. First cited on page 92

Boulez, Pierre, & Gerzo, Andrew *Computers in Music* Scientific American, 258(4) April 1988. First cited on page 85

Bowler, I W, Manning, P D, Purvis, A, Bailey, N J *A Transputer-Based Additive Synthesis Implementation* Proceedings of the International Computer Music Conference, Ohio 1989. pp58–61. First cited on page 18

Bowler, I W, Manning, P D, Purvis, A, Bailey, N J *New Techniques for a Real-Time Phase Vocoder* Proceedings of the International Computer Music Conference 1990, Glasgow. pp178–180 First cited on page 106

Bowler, I W, Manning, P D, Purvis, A, Bailey, N J *On Mapping N Articulation- onto M Synthesiser-Control Parameters* Proceedings of the International Computer Music Conference 1990, Glasgow. pp181–184 First cited on page 106

Boynton, L, Cumming, D *A Real-Time Acoustic Processing Card For The Mac-II* Proceedings of the International Computer Music Conference, 1988, pp349–356 First cited on page 124

Bronson, E C, Casavant, T L, Jamieson, L H *Experimental Application-Driven Architecture Analysis of an SIMD/MIMD Parallel Processing System* IEEE Transactions on Parallel and Distributed Systems, Vol. 1, No. 2, April 1990, pp195–205. First cited on page 112

Bruun, George *Z-Transform DFT Filters and FFTs* IEEE Transactions on Acoustics, Speech and Signal Processing ASSP26(1) (Feb 1978) pp56–63 First cited on page 112

Buxton, William, Sniderman, Richard, Reeves, William, Patel, Sanad, & Baeker, Ronald *The Evolution of the SSSP Score-Editing Tools* Computer Music Journal 3(4) (1979) pp14–25 First cited on page 6

Buxton, W *Design Issues in the Foundation of a Computer-Based Tool for Music Composition* Structured Sound Synthesis Project, Computer Systems Research

Group, University of Toronto, Toronto, Ontario, Canada M5S 1A4. First cited on page 6

Cadoz, C, Luciani, A, & Florens, J *Responsive Input Devices and Sound Synthesis by Simulation of Instrumental Mechanisms: The Cordis System* Computer Music Journal, vol. 8 no. 3 (Autumn 1984). First cited on page 123

Carbonneau, Gerard R *Timbre and the Perceptual Effects of Three Types of Data Reduction* Computer Music Journal 5(2) pp10–19 First cited on page 105

Casserley, Lawrence *Seried Phi Real-Time Digital Signal Processor* Proceedings of the International Computer Music Conference, Glasgow, 1990, pp124–126 First cited on page 80

Chamberlain, H *Musical Applications of Microcomputers* Hayden Books, Howard W Sams &Co. 4300 West 62nd St., Indianapolis, Indiana 64268 USA ISBN 0–8104–5768–7 (1987 2nd Edition) First cited on page 3

Chen, J R, Mars, P *Artificial Neural Networks and Nonlinear System Identification* Internal report: School of Engineering and Applied Science, Uuniversity of Durham, South Road, Durham, United Kingdom (Jan 1990) First cited on page 124

Clarke, M, Manning, P D, Berry, R, Purvis, A *Vocel: An FOF Unit-Generator for* MUSIC11 International Computer Music Conference, Cologne, 1988. First cited on page 7

Comerford, P J, Eaglestone, B M *Bradford Musical Instrument Simulator and Workstation* Proceedings of Euromicro '88 conference on Supercomputer Technology and Applications, Zurich. Microprocessing and Microprogramming 24 (1988), pp73–78. First cited on page 3

Cook, Perry R *Synthesis of the Singing Voice Using a Physically Parameterised Model of the Human Vocal Tract* Proceedings of the International Computer Music Conference, Ohio, 1989. pp69–72 First cited on page 4

Cooley, James W, Tukey, John W *An Algorithm for the Machine Calculation of Complex Fourier Series* Mathematics of Computation 19 (April 1965), pp297–301 First cited on page 111

Cooprider, L *Petri-nets and the Representations of Standard Synchronisations* Dept of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania (Jan. 1976) First cited on page 38

Courtois, P, Heymans, F, & Parnas, D *Concurrent Control with 'Readers' and 'Writers'* Communications of ACM 14(10) (Oct. 1971) pp667–668 First cited on page 36

Dannenberg *Software Techniques for Interactive Performance Systems* Proceedings of the International Workshop on Man-Machine Interaction in Live Performance, Computer Music Department CNUCE/CNR, Pisa, June 1991. First cited on page 87

Decker, S L, Kendall, G S, Schmidt, B L, Ludwig, M D, & Freed, D J *A Modular Environment for Sound Synthesis and Composition* Computer Music Journal 10(4) pp28–40 (1986) First cited on page 17

Despain, Alvin M, & Patterson, David A *X-Tree: A Tree Structured Multi-Processor Computer Architecture* Proceedings of the 5th Annual Symposium on Computer Architectures, April 1978, pp144–151 First cited on page 103

Dijkstra, E W *Cooperating Sequential Processes* in 'Programming Languages' ed. Genuys, F New York Academic Press (1968) pp34–112 First cited on page 26

Dijkstra, E W *Solution of a Problem in Concurrent Program Control* Communications of the Association of Computing Machinery, Vol. 8(5) (Sept 1965) p569. First cited on page 26

Dolson, M *Machine Tongues XII: Neural Networks* Computer Music Journal 13(3) (Autumn 1989) pp28–40 First cited on page 124

Dolson, M *The Phase Vocoder: A Tutorial* Computer Music Journal 10(4) pp14–27 (1986) First cited on page 94

Duhamel, P, Piron, B, Etcheto, J M *On Computing the Inverse DFT* IEEE Transactions on Acoustics, Speech and Signal Processing, 36(2) (Feb 1988), pp285–286

Emmerson, Simon (Editor) *The Language of Electroacoustic Music* Macmillan Press Ltd 1986. ISBN 0–333–39760–6. First cited on page 5

Endrich, T *C.D.P. Reference Manual* C.D.P. Limited, 1987. First cited on page 7

F Richard Moore *Elements of Computer Music* Prentice Hall, 1990. ISBN 0–13–252552–6 First cited on page 64

Fay D Q M *An Implementation of the Fast Fourier Transform in Occam* Computer Science and Informatics 14(2) pp3–12 First cited on page 111

Finkel, A, Choquet, A *FIFO Nets Without Order Deadlock* Acta Informatica 25(1) (1986) pp15–36 First cited on page 30

Fischer, Michael J, Lynch, Nancy A, Burns, James E, & Borodin, Alan *Distributed FIFO Allocation of Identical Resources Using Small Shared Space* ACM Transactions on Programming Languages and Systems, 11(1) pp90–114 First cited on page 30

Flanagan, J L, Golden, R M *The Phase Vocoder* Bell System Technical Journal 45 pp1493–1509 (1966) First cited on page 94

Flynn, Michael J. *Some Computer Organisations and their Effectiveness* IEEE Transactions on Computers C21 pp948–960 (1972) First cited on page 17

Foley, J D *Interfaces for Advanced Computing* Scientific American, 1987 pp83–90 First cited on page 81

Gordon, J W, Strawn, J *An Introduction to the Phase Vocoder* in 'Digital Audio Signal Processing — an Anthology' The Computer Music and Digital Audio Series ed. Strawn J: William Kaufmann, Inc., 95 First Street, California 94022 ISBN 0–86576–082–9. First cited on page 94

Gould, L, Bowler, I W, Purvis, A *Real-Time Multi-Channel Digital Filtering on the Transputer* Submitted to Computer Architecture and Digital Signal Processing IEE Conference, Hong Kong, 1989. First cited on page 18

Grey, J M *Doctoral Thesis* Psychology Department of Stanford University, USA (Feb. 1975) First cited on page 3

Groginsky, Herbert L, & Works, George A *A Pipeline Fast Fourier Transform* IEEE Transactions on Computers, Nov 1970, pp1015–1019 First cited on page 112

Haus, Goffredo *Music Processing at L.I.M.* Proceedings of Euromicro '88 conference on Supercomputer Technology and Applications, Zurich. Microprocessing and Microprogramming 24 (1988), pp 435–441. First cited on page 5

Haynes, Stanley *The Musician-Machine Interface in Digital Sound Synthesis* Ph.D. Thesis, Department of Music, University of Durham First cited on page 6

Heinbach, W *Aurally Adequate Signal Representation: The Part Tone Time Pattern* Acustica vol. 67 (1988) pp113–121 First cited on page 101

Helm, E *Composer, Performer, Public: A Study in Communication* International Music Council; Music & Communication; 1. Florence, 1970 First cited on page 91

Helmbold, David P, & McDowell, Charles *Modeling Speedup(N) Greater Than N* IEEE Transactions on Distributed and Parallel Systems 1(2) (April 1990) pp250–256 First cited on page 59

Holm, Frode *Frequency Scheduling: Real-Time Scheduling in Multiprocessing Systems* Proceedings of the International Computer Music Conference, Ohio, 1989 pp127–130 First cited on page 56

Inmos Ltd. *Transputer Development System ch. 5-7* Prentice Hall ISBN 0-13-928995-X First cited on page 114

Kailath, T *Signal Processing Applications of Concurrent Array Processors* In 'VLSI and Modern Signal Processing' ed. Kung, Whitehouse and Kailath, Prentice Hall, 1985. ISBN 0-13-942699-X First cited on page 111

Kamangar, F A, Duderstadt, R A, Smith, J O *Implementing the Back-Propagation Algorithm on the Meiko Parallel Computing Surface* Proceedings of the International Conference on the Applications of Transputers, Liverpool, 1989. Published as 'Applications of Transputers, Volume 1', Len Freeman and Chris Phillips (ed.) IOS Press (1990) ISBN 90-5199-025-1. First cited on page 66

Karp, R, Miller, P *Parallel Program Schemata* Journal of Computer and Systems Science 3(4) (April 1968)

Kasparec, F, Doppelbauer, J, Graebner, H, Mandl, T *Advanced Transputer Interconnection Techniques* Proceedings of the International Conference on the Applications of Transputers, Liverpool, 1989. Published as 'Applications of Transputers, Volume 1', Len Freeman and Chris Phillips (ed.) IOS Press (1990) ISBN 90-5199-025-1.

Katayose, H, Kato, H, Imai, M, & Inokuchi, S *An Approach to an Artificial Music Expert* Proceedings of the International Computer Music Conference, Ohio, 1989. pp139-146 First cited on page 6

Katayose, H, Takami, K, Fukuoka, T, & Inokuchi, S *Music Interpreter in the Kansei Music System* Proceedings of the International Computer Music Conference, Ohio, 1989. pp147-150 First cited on page 6

Keane, D, & Gross, P *The MIDI Baton* Proceedings of the International Computer Music Conference, Ohio, 1989. First cited on page 92

Kirk, R & Orton, R *MIDAS A Musical Instrument Digital Array Signal Processor* Proceedings of the International Computer Music Conference, Glasgow, 1990, pp127–131 First cited on page 80

Knuth, D E *The Art of Computer Programming, vol. 1, sec 2.2 pp234 et seq.* Addison-Wesley (1968) First cited on page 30

Koikkalainen, Pasi & Sauer, Frank *Architecture-independent Multicomputing via a self-distributing communication Harness* Information Processing Laboratory Publication, Lappeenranta University of Technology, POBox 20, SF–53851, Lappeentanta, Finland. Abstracted by 1st Nordic Transputer Seminar, Turku, Finland (1990). First cited on page 66

Kolba, Dean P, Parks, Thomas W *A Prime-Factor FFT Algorithm Using High-Speed Convolution* IEEE Transactions on Acoustics, Speech and Signal Processing ASSP25(4) (Aug 1977) pp218–294. First cited on page 113

Laden, B, Keefe, D H *The Representation of Pitch in a Neural Net Model of Chord Classification* Computer Music Journal 13(4) (Winter 1989) pp12–26 First cited on page 124

Langston, P S *Six Techniques for Algorithmic Music Composition* Proceedings of the International Computer Music Conference, Ohio, 1989. pp164–167 First cited on page 5

Laurson, M, & Duthen, J *Patchwork: A Graphic Language in Preform* Proceedings of the International Computer Music Conference, Ohio, 1989. pp172–175. First cited on page 17

Lee, Edward Ashford, & Messerschmitt, David G *Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing* IEE Transactions on Computers Vol C-36, No 1. January 1987. First cited on page 16

Lent, K *An Efficient Method of Pitch Shifting Digitally Sampled Sounds* Computer Music Journal 13(4) (Winter 1989) pp65–71 First cited on page 64

Lippmann, R P *An Introduction to Computing with Neural Nets* IEEE ASSP Magazine April 1987 pp4–22 First cited on page 124

Lo, K C, Siu, W C, Lun, D P K, Purvis, A *Address Generation of Prime Factor Algorithm in a Multiprocessor System* Presented at IEEE Conference on Circuits and Systems, Singapore, July 1991 First cited on page 113

Lowe, W, Currie, R *Digidesign's Sound Accelerator: Lessons Lived and Learned* Computer Music Journal 13(1) pp36–46 (1989) First cited on page 3

Loy, Gareth *On the Scheduling of Multiple Parallel Processors Executing Synchronously* Proceedings of the International Computer Music Conference, 1987. pp117–124 First cited on page 124

Lynch, Nancy A, & Fischer, Michael J *A Technique for Decomposing Algorithms which use a Single Shared Variable* Journal of Computer System Science 27(3) (1983) pp350–377 First cited on page 31

Macnaghten, A M, Hoare, C A R *Fast Fourier Transform free from Tears* The Computer Journal 20(1) (1977) pp78–83 First cited on page 111

Manning, P D *Electronic and Computer Music* Oxford University Clarendon Press (1985) ISBN 0–19–311981–8 First cited on page 5

McAdams, Stephen, & Bregman, Albert *Hearing Musical Streams* Computer Music Journal 3(4) (1979) pp26–43,60,63 First cited on page 6

McGee, W F, Merkley, Paul *A Real-Time Logarithmic-Frequency Phase Vocoder* Computer Music Journal 15(1) pp20–27 (Spring 1991) First cited on page 96

Meller, D *Partitioning Big Matrices for Small Systolic Arrays* In 'VLSI and Modern Signal Processing' ed. Kung, Whitehouse and Kailath, Prentice Hall, 1985. ISBN 0–13–942699–X

Moorer, J A *The Use of the Phase Vocoder in Computer Music Applications* Journal of the Audio Engineering Society 26(1) pp42–45 (January/February 1978) First cited on page 94

Morita, H, Ohteru, S, & Hashimoto, S *Computer Music System which Follows a Human Conductor* Proceedings of the International Computer Music Conference, Ohio, 1989. pp207–210 First cited on page 87

Murata, T *Petri-nets: Properties, Analysis and Applications* Proc. IEEE 77(4) (April 1989) pp 541–580 First cited on page 31

Oetken, Geerd, Parks, Thomas W, & Schuessler, Hans W *New Results in the Design of Digital Interpolators* IEEE Transactions on Acoustics, Speech and Signal Processing, ASSP-23(3) (June 1975) pp301–309 First cited on page 127

Oppenheim, A V, Schafer, R W *Digital Signal Processing* Prentice-Hall International (1975) First cited on page 127

Orlarey, Y, & Lequay, H *MIDI Share: A Real-Time Multi Tasks Software Module for MIDI Applications* Proceedings of the International Computer Music Conference, Ohio, 1989. pp 234–237 First cited on page 56

Parkinson, D *Parallel Efficiency can be greater than Unity* Parallel Computing 3, pp261–262 (1986) First cited on page 110

Peacock, J Kent *Deadlock Avoidance in Loosley-Coupled Multiprocessors with Finite Buffer Pools* ACM Operating Systems Review 23(2) (April 1989) pp20–24 First cited on page 38

Pease, M C *An Adaptation of the Fast Fourier Transform for Parallel Processing* Journal of the Association of Computing Machinery 15(2) (April 1968) pp252–264 First cited on page 125

Peterson, James L *Petri-net Theory and the Modeling of Systems* Prentice-Hall (1981) ISBN 0–13–661983–5 First cited on page 31

Portnoff, M R *Implementation of the Digital Phase Vocoder using the Fast Fourier Transform* IEEE Transactions on Acoustics, Speech and Signal Processing, ASSP-24(3) (June 1976) First cited on page 94

Rabiner, L R, Gold, B *Theory and Application of Digital Signal Processing* Prentice-Hall International (1975) ISBN 0–13–914101–4. Section 9.12 (pp563-565) First cited on page 127

Reed, I S, Tufts, D W, Yu, X, Truong, T K, Shih, M-T, Yin, X *Fourier Analysis and Signal Processing by use of the Möbius Inversion Formula* IEEE Transactions on Acoustics, Speech and Signal Processing 38(3) (March 1990) First cited on page 112

Rivas, D, Watkins, S, Chau, P M *VLSI for a Physical Model of Musical Instrument Oscillations* Proceedings of the International Computer Music Conference, Ohio, 1989. pp253–256 First cited on page 4

Roads, C *A Tutorial on Non-Linear Distortion or Waveshaping Synthesis* Computer Music Journal 3(2) pp29–34 (1979)

Roads, C *Composers and the Computer* William Kaufmann Inc., LA, California. 1985 First cited on page 5

Roads, C *Granular Synthesis of Sounds* Computer Music Journal 2(2) pp61–62 (1978) First cited on page 4

Roads, C *The Second STEIM Symposium on Interactive Composition in Live Electronic Music* Computer Music Journal, 10(1) (Summer 1986) pp44–50. First cited on page 88

Rodet, X, Potard, Y, Barriere, J *The Chant Project* Computer Music Journal, 8(3), 1984. First cited on page 4

Rodet, X *Time-Domain Formant-Wave-Function Synthesis* Computer Music Journal, 8(3), 1984 pp 9–14 First cited on page 4

Sasaki, Lawrence H, & Smith, Kenneth C *A Simple Data Reduction Scheme for Additive Synthesis* Computer Music Journal 4(1) pp22–24. First cited on page 3

Schaeffer *Esquisse d'un Solfège Concret* (1952). First cited on page 83

Schindler, K W *Dynamic Timbre Control for Real-Time Digital Synthesis* Computer Music Journal, 8(1) (Spring 1984)

Schlang, Martin F, & Mummert, Markus *Die Bedeutung der Fensterfunktion Für die Fourier-T-Transformation als Gehörgerechte Spektralanalyse* Lehrstuhl für Elektroakustik, TU, München First cited on page 101

Schroeder, M R *Linear Prediction, Entropy and Signal Analysis* IEEE ASSP Magazine, July 1984 First cited on page 65

Sedgewick, Robert *Algorithms* Addison-Wesley 1983. ISBN 0–201–06672–6 First cited on page 17

Serra, Marie-Helene, Rubine, Dean, & Dannenberg, Roger D *The Analysis and Resynthesis of Tones via Spectral Interpolation* Proceedings of the International Computer Music Conference 1988, pp322–332 First cited on page 3

Serra, X, Smith, J O *Spectral Modeling Synthesis* Proceedings of the International Computer Music Conference, Ohio, 1989. pp281–284 First cited on page 105

Snell, J M *General-Purpose High-Fidelity Affordable Real-Time Computer Music System* 1987 ICMC Proceedings, pp130–137. First cited on page 124

Snyder, L *Configurable, Highly Parallel (CHiP) Approach to Signal Processing Applications* SPIE Technical Symposium East 1982 Proceeding First cited on page 111

Snyder, L *Introduction to the Configurable, Highly Parallel Computer* Computer 15(1) (1982) pp47–56 First cited on page 124

Snyder, L *Overview of the CHiP Computer* In 'VLSI81' ed. J P Gray, Academic Press, New York (1981) ISBN 0–12–296860–3 First cited on page 124

Snyder, L *The Role of the CHiP Computer in Signal Processing* In "VLSI and Modern Signal Processing" ed. Kung, Whitehouse and Kailath, Prentice Hall, 1985. ISBN 0–13–942699–X First cited on page 111

Stapleton, J C, Bass, S C *Synthesis of Musical Tones Based on the Karhunen-Loève Transform* IEEE Transactions on Acoustics, Speech and Signal Processing, 36(3) (March 1988) pp305–319

Stone, Harold S *Parallel Processing with the Perfect Shuffle* IEEE Transactions on Computers C-20(2) (Feb 1971), pp153–161 First cited on page 125

Strasburger, Hans, Kohler, Stefan, & Radauer, Irmfried *Score Input to* CSOUND *via the MIDI Keyboard* Proceedings of the International Computer Music Conference, Glasgow, 1990, p208 First cited on page 6

Swartzlander, E E, Jnr, & Halnor, G *Frequency Domain Digital Filtering with VLSI* In 'VLSI and Modern Signal Processing" ed. Kung, Whitehouse and Kailath, Prentice Hall, 1985. ISBN 0-13-942699-X First cited on page 112

Tanimoto, S L *A Pyramidal Approach to Parallel Processing* Proceedings of the 10th international symposium on computer architecture, Stockholm, June 1983 First cited on page 125

Tanimoto, S, Pavlidis, T *A Hierarchical Data Structure for Picture Processing* Computer Graphics and Image Processing 4 pp104–119 (1975) First cited on page 125

Tannenbaum, A S *Operating Systems* Prentice Hall, 1987 First cited on page 20

Tarabella, L, Bertini, G *A Digital Signal Processing System and a Graphic Editor for Synthesis Algorithms.* Proceedings of the International Computer Music Conference, Ohio, 1989. pp312–315 First cited on page 17

Tayli, Murat, & Benmaiza, Mohamed *Transputer Implementations of General Semaphores* Occam User Group Newsletter, No. 14 (January 1991) pp50–60 First cited on page 26

Terhardt, E *Fourier Transformation of Time Signals: Conceptual Revision* Acustica 57 (1985). pp242–256 First cited on page 101

Thompson, Clark D *Fourier Transforms In VLSI* IEEE Transactions on Computers C-32(11) (Nov 1983) pp1047–1057 First cited on page 111

Truax, B *Real-Time Granular Synthesis with a Digital Signal Processor* CMJ 12(2) Summer 1988. First cited on page 4

Truong, T K, Reed, I S, Yeh, C-S, Chang, J J, Shao, H M *A Parallel VLSI Architecture for a Digital Filter using a Number-Theoretic Transform* In "VLSI AND MODERN SIGNAL PROCESSING" ed. KUNG, WHITEHOUSE AND KAILATH, Prentice Hall, 1985. ISBN 0–13–942699–X First cited on page 113

Tsay, Jong-Chuang, & Yuan, Sy *Systolic Flow* Journal of Parallel and Distributed Computing 8 (1990) pp 286–291) First cited on page 111

Vercoe, B, & Cumming, D *Connection Machine Tracking of Polyphonic Audio* Proceedings of the International Computer Music Conference, Cologne, 1988. First cited on page 87

Vercoe, B, & Puckette, M *Synthetic Rehearsal: Training the Synthetic Performer* Proceedings of the International Computer Music Conference, 1985, pp275–278 First cited on page 87

Vercoe, Barry, & Ellis, Dan *Real-Time* CSOUND*: Software Synthesis with Sensing and Control* Proceedings of the International Computer Music Conference, Glasgow, 1990. pp209–211. First cited on page 87

Vercoe, B CSOUND *Reference Manual* MIT Press, 1986. First cited on page 2

Walker, W F *KIWI: A Parallel System for Software Sound Synthesis* Proceedings of the International Computer Music Conference, Ohio, 1989. pp328–331 First cited on page 17

Wessel, David W *Timbre Space as a Musical Control Structure* Computer Music Journal 3(2) (1979) pp45–52 First cited on page 105

Winograd, S *On Computing the Discrete Fourier Transform* Mathematics of Computation, 32(141) (Jan 1978) pp175–199 First cited on page 111

Wishart, T *C.D.P. Phase Vocoder Reference Manual* C.D.P. Limited, 1989. First cited on page 64

Yavelow *Music & Microprocessors: MIDI and the State of the Art* in 'The Music Machine', ed. Roads, pp199-241, M.I.T. press, 1989. ISBN 0–262–18131–2 First cited on page 87

Yuhang Wu *New FFT Structures Based on the Bruun Algorithm* IEEE Transactions on Acoustics, Speech and Signal Processing 38(1) (Jan 1990) First cited on page 112

Zicarelli, D *M and Jam Factory* Computer Music Journal, 11(4) (Winter 1987) pp13–29 First cited on page 5

Zwicker, E, & Terhardt, E *Analytical Expressions for Critical-Band Rate and Critical Bandwidth as a Function of Frequency* Journal of the Accoustical Society of America, 65(5) (Nov 1980) pp1523–1525 First cited on page 101

van Renterghem, P *Applicability of a 16-Node Transputer Array Without External Memory* in 'Applying Transputer Based Parallel Machines' ed. Bakkers, A IOS Press, Netherlands, 1989. ISBN 90–5199–011–1 First cited on page 126

Glossary of Terms and Abbreviations

A.D.S.R. "Attack, Decay, Sustain, Release": a simple envelope model used to control the variation in the amplitude of the audio signal in early electronic synthesisers.

A.F.T. "Arithmetic Fourier Transform(ation)": a method of deriving the Fourier transformation of a data set making use of results from number theory.

C.C.D. "Charge-Coupled Device": an electronic device which depends upon the storage and transfer of charge for its operation.

C.D. "Compact Disk": a digital audio playback medium upon which commercially made recordings are distributed. The design specification, 41000 samples per second each of 16 bits accuracy, closely approaches the capability of the human ear.

C.D.P. "Composers' Desktop Project": a limited company established to promote and coordinate research in electroacoustic composers' tools in the U.K. and abroad.

C.S.P. "Communicating Sequential Processes": a calculus for the description of parallel computer programs, which is capable of determining the deadlock-freeness of a system.

CHiP "Configurable, Highly Parallel": the CHiP computer was constructed using many processors connected by communications paths which are reconfigurable at run-time.

D.F.T. "Discrete Fourier Transform(ation)"

Dianomic An attribute of a computer program which obtains its concurrency by distributing discrete work-packets and their associated data across a processor array for asynchronous exection.

D.S.P. Variously "Digital Signal Processing", "Digital Signal Processor" etc.

F.F.T. "Fast Fourier Transform(ation)": a type of D.F.T. *q.v.* which requires a reduced number of arithmetic operations compared with the original algorithm.

FOF *"Formes d'Onde Formatique"* ("Formant Wave-function Synthesis"): a method of audio synthesis capable, amongst other things, of remarkably accurate simulations of the singing voice.

Gesture A physical movement or group of movements. Applied musically, the term refers to the sound arising from a performer's movement or group of movements, and can be regarded as similar to a phrase (although a phrase may be composed of many gestures). Also applied abstractly to computer music.

I.I.R. "Infinite Impulse Response": a class of digital filter having the characteristic that the length of response to an impulse at the input is infinitely long.

IRCAM French government-funded institution researching into computer and electroacoustic music.

Isonomic An attribute of a program which obtains its concurrency by executing the same algorithm on many processors, but with a different set of data in

each case.

MIMD "Multiple Instruction, Multiple Data": a class of parallel computer which executes many instructions on many different data simultaneously.

M.I.S.D "Multiple Instruction, Single Data*(sic)*": a parallel computer which executes many instructions on a single datum. Not an oft-used abbreviation, although Flyn equates it to a pipeline.

M.M.I. "Man–Machine Interface": that part of a system (hardware or software) which is responsible for receiving user instructions and presenting results. Referred to increasingly as H.C.I. (Human–Computer Interface) especially in the U.S.A., because of pressure from the Feminist lobby.

MIDI "Musical Instrument Digital Interface": an interface specification for the control of commercially available synthesisers.

N.T.T. "Number Theoretical Transform(ation)": a class of A.F.T. *q.v.*

Parallelisation The process of converting a sequential computer program into a form suitable for execution by several processors concurrently, for the purpose of achieving a decrease in the time required to complete processing.

P.C.B. "Printed Circuit Board": a perforated composite board, usually fibreglass, with a pattern of copper conductors, into which electronic components are inserted.

SIMD "Single Instruction, Multiple Data": a class of parallel computer which executes a single struction on many data simultaneously.

Systolic An attribute of a computer which obtains its concurrency by using many processors to perform operations on data as soon as the data are presented, whereupon the result is immediately transmitted to the processing element which requires it.

Transputer A digital computer designed in such a way as to permit the simultaneous communication with other devices and processing of data. Supports the C.S.P. *q.v.* model of parallel computation.

Vocoder "Voice Coder": a method of representing an audio signal, originally intended to be a speech signal in telephony, as a set of data corresponding to the amount of energy in several frequency ranges.

Appendix A

Adding a New Unit Generator to CSOUND

A.1 Introduction

CSOUND IS A HIGHLY FLEXIBLE synthesis system: the user is able to build his own synthesis 'instruments' from modules, known as 'unit-generators'. A unit-generator is in fact a subroutine in the program, and the user creates an instrument by compiling a sequence of such subroutine calls. CSOUND calculates blocks of sound, calling the unit-generator routines not at every sample but at a slower control rate. The audio-rate information is calculated by internal loops within the unit-generator and the results stored in buffers. Each unit-generator has its own data structure attached to it so that local data can be preserved from one pass to the next where necessary. A separate copy of this data structure is created for each instance of the unit-generator within the orchestra. A wide variety of unit-generators is already available in the standard CSOUND covering all the normal functions required in sound synthesis. There are, for example, table lookup routines (with or without interpolation), F.M. generators, random generators, filters, breakpoint functions etc. For much of the time it will not therefore be necessary to add further unit-generators to CSOUND: the user will be able to build whatever is required out of the existing building blocks. However as new algorithms are discovered and applied to music synthesis there will be occasions when it is necessary to add a new unit-generator to the existing library.

An example of this is the F.O.F. algorithm used in the CHANT program of IRCAM. Clarke wanted to be able to use this algorithm in the context of CSOUND in order to be able to explore its potential in new ways. It is a tribute to the flexibility of CSOUND that it was possible to recreate this extremely complex algorithm as a CSOUND instrument using the standard unit-generators. However, this led to a very large and cumbersome orchestra file which was also inefficient to run, and so it seemed advisable to write a new unit-generator specifically for this task. The addition of this new F.O.F. unit-generator will be taken as an example of how to add a new module: because of its complexity the algorithm illustrates many of the problems likely to be encountered. The standard library of unit-generators is found in the CSOUND files UGENS1.C, UGENS2.C etc., together with the associated header files (UGENS1.H etc.). It is necessary to add the new routine to one of these files or, preferably, to add the new routine in new files (in the C.D.P. release of the extended CSOUND the F.O.F. routines are in the files UGENSM.C and UGENSM.H). If new files are added an additional '`include "newfile.h"`' statement must be put in the file ENTRY.C referencing the header file for the new unit-generator. The additional statement in ENTRY.C for the F.O.F. addition is "`#include "ugensm.h"`"

A.2 Changing the Source

A.2.1 The Unit-generator Header File

This file may contain any definitions required for the new unit-generator. For example, in the F.O.F. generator header file the following is one of the statements:

```
#define LOCAL_BUFFER_SIZE 11
```

The most important feature of the file, however, is that it contains the definition of a structure for the unit-generator. This structure is the data space required for each instance of the generator. This is an abbreviated summary of the structure for the F.O.F. unit-generator:

```
typedef struct {
   OPDS   h;
   float  *ar, *xamp, *xfund, ... ;
   float  clock_incr, duration, ... ;
   long   fundphs, fund_incr, ... ;
   short  fund_per_flag, fof_count, ... ;
   char   *auxds, **auxpchain;
   FUNC   *ftp1, *ftp2;
} FOFS;
```

OPDS is itself a structure (defined in CS.H) and provides the space necessary for the program in creating an instance of the generator and linking together the modules that comprise the instrument. This is followed by a list of the pointers to the output and input buffers. `*ar` is the output buffer of the F.O.F., `*xamp` is the amplitude input, and so on. After this follows a list of the variables used by the unit-generator. Only those that need to be saved from one k-pass to another need to be declared here, those not requiring storage do not need to be part of this data structure and may be declared locally. (`fund_phs`, for example, is the phase of the fundamental and its value must obviously be saved at the end of a k-rate pass for use the next time around). The two variables `*auxds` and `**auxpchain` appear because the F.O.F. routine makes use of additional reserved memory space (for data about each of the overlapping F.O.F.s). Most unit-generators will not require such space. `*ftp1` and `*ftp2` are the pointers to the stored function tables.

A.2.2 The Unit-generator Code

The file that is to contain the code for the unit-generator will probably need to contain include statements for at least these two files: `<math.h>`, and CS.H (being the main header file for CSOUND). It will, of course, also need to include its own header file (UGENSM.H for the F.O.F.s). Externals which may well be needed

(and therefore declared) are as follows:

Floats:

`esr`	Audio Sample Rate
`ekr`	Control Rate
`sicvt`	2^{24}/`esr`
`kicvt`	2^{24}/`ekr`
`maxlen`*	2^{24}
`PMASK`	$2^{24} - 1$
`dv32768`	1/32768

Ints:

`ksmps`	Number of Samples per k-Period

Chars:

`errmsg[]`	Used for Error Messages

Functions:

`*auxalloc()`	System Memory Allocator
`*spalloc()`	Ditto

In the standard CSOUND table-lookup routines the phase is calculated as a 24 bit integer: the user's floating-point input (between 0 and 1) is multiplied by `maxlen` and converted to type `long`. Later it is scaled to the size of the lookup table by bit-shifting: each function table has stored as a member of its data structure the variable `lobits` which is used for this purpose. The unit-generator code itself is normally in two parts; that used for the initial pass, and that used for subsequent control-rate passes. CSOUND convention is to end the name of the initiation subroutine `set()` (for example: `oscset()`, `linset()`, `fofset()`, etc.). The initial pass makes any necessary preparations in advance of the calculation of the audio signal. It will usually be very short, though the complexity of the F.O.F. algorithm means that `fofset()` is longer than average. A simpler example is the initialisation for the oscillator routines `oscset()` (found in UGENS2.C), which is

only 10 lines of code (3 of which are just brackets). It checks the presence of the lookup table and finds its location, and then calculates the initial phase as a 24-bit integer. The longer F.O.F. initialisation has to locate two lookup tables, allocate extra memory space according to the maximum number of overlapping F.O.F.s, and prepare many flags and variables.

The code for the control rate passes will normally have the following general structure, (illustrated here by reference to one of the oscillator subroutines):

```
oscak(p)
OSC *p; /*OSC is the structure for the unit-generator*/
{
   register FUNC  *ftp; /* the structure for the stored table*/
   register float *ar, *ampp ,*ftbl; /*pointers to i/o buffers*/
   register long  phs, inc, lobits, nsmps = ksmps; /*local data*/

   ...control-rate code...

   do {
       ...audio-rate code...
   } while(--nsmps);

   ...more control-rate code...
}
```

This code will be called once each control period. Calculations at audio rate (if any) are performed in a loop, for which `nsmps` (initialised from the global `ksmps`) is used as a counter. Declarations include that of the pointer to the unit-generator structure `*p`, and local declarations of pointers to the input/output data, and of variables.

A.2.3 Updating ENTRY.C

In order for the unit-generator to be recognised by the control part of the program, information about it must be placed in the file ENTRY.C. If the additional unit-generator has been placed in new files then the file that contains the header information for the new generator (in particular its control structure) must be "`#included`" at the beginning of ENTRY.C (where `#include` statements for the other unit-generator files will be found). A little later in this file all the unit-generator functions are declared. Both the initialisation subroutine and the control-rate subroutine (and any others) must be added to this list, for example:

```
int fofset(), fof();
```

Further into the file will be found a description of each unit-generator, the description added for the F.O.F. generator was as follows:

```
{ "fof", S(FOFS), 5, "a", "xxxxkkkkkiiiiop", fofset, NULL, fof},
```

For every such line added to the list of operators the manifest constant "`OPLSTMAX`", defined near the top of ENTRY.C, must be increased by one. The above description is more complex than for most unit-generators, but its meaning is as follows:

`"fof"` The opcode used to call the generator in the orchestra file.

`S(FOFS)` `S` means `sizeof` (defined earlier in ENTRY.C), this is therefore the size of the unit-generator's data structure (`FOFS` was the name given to this structure in the header file described above).

`5` This is the "thread" value (*not to be confused with the parallel processing concept of a thread*). The threads are defined earlier in ENTRY.C and are a binary coding describing the subroutines used by the unit generator:

bit 0 set if a subroutine exists for initialisation;

bit 1 set if a subroutine exists for k-rate output;

bit 2 set if exactly one subroutine exists for a-rate output;

bit 3 set if a variety of subroutines exist for a-rate output depending on the variable ('x') input-rates.[1] (The routines themselves are listed later in this line of code in the order: i-rate, k-rate, a-rate, input dependent — see below.) In our example bits 0 and 2 are set $(0101)_2$, meaning that this unit-generator has an initialisation subroutine and another subroutine for a-rate output.

`"a"` the output rate (audio-rate in this case), alternatives include:

`"i"` init rate;

`"k"` control rate;

`"s"` control or audio rate;

`""` no output;

`"mmmm"` optional outputs (max = 4);

`"x.."` the rates of the inputs in the order in which they occur:

`"x"` audio or control rate;

`"k"` control rate;

`"i"` init rate;

`"o"` optional input, defaults to 0;

`"p"` optional input, defaults to 1;[2]

`fofset` the init pass routine (bit 0 in coding above);

[1]The most complicated set of alternative subroutines in the standard CSOUND is that for the oscillator (its thread value is 11 (decimal)). This is no doubt in order to maximise the efficiency of a very commonly used generator: `oscset` is the initialisation routine; `koscil()` is the routine called for output at the k-rate; `osckk()`, `oscka()`, `oscak()`, or `oscaa()` are invoked depending upon the data rate of the 'x' inputs, amplitude and frequency

[2]Other defaults are described earlier in ENTRY.C and defined in RDORCH.C. The number of input fields for the F.O.F. generator is unusually large, and the number of "x" inputs made it necessary to revise RDORCH as described

`NULL` there is no k-rate output routine (bit 1);

`fof` the a-rate output routine (bit 2).

A.2.4 Other Necessary Modifications

Apart from the addition of the new code itself, and a change to the make file and LNK files to indicate the presence of a new module, the above modifications to ENTRY.C are all that would usually be necessary for the addition of an extra unit-generator. The complexity of the F.O.F. generator however is such that further special features were needed. Brief notes on these follow in case they are of use in other situations.

RDORCH.C

The standard CSOUND unit-generators use no more than two "x" inputs (variable audio or control rate inputs). In RDORCH.C a binary coding is used to record which type of input is actually in use in any particular instance. The F.O.F. generator has 4 "x" inputs and it was therefore necessary to extend this coding procedure. A simple 4 line patch was all that was needed.

Auxiliary Memory Space

A few unit-generators such as `delay` and `comb` make use additional system memory spaces (one for each instance of the generator), which is outside the normal data structure, because its size is determined at initialisation time by input data. (In `delay` the size of the extra space depends on the delay time). The F.O.F. unit-generator also required additional disk space for storing data local to each overlapping F.O.F. excitation. The extra space is allocated by calling, e.g., "`auxp = auxalloc(xds, &p->auxds);`" during the initialisation pass. (the code may be found in AUXFD.C). `auxalloc()` takes two arguments: the size of the space required in bytes; and the location for the storage of a pointer to the address of the auxiliary space. The routine also returns this same pointer. If

this procedure is to be used the data structure for the unit-generator must contain two declarations to which the routine will write (failure to declare these will be disastrous!): "`char *auxds, **auxpchain;`" The use of `*auxds` has already been seen, `*auxpchain` is used internally to keep track of the memory allocations.

Stored Function-tables

CSOUND provides 'gen routines' to create stored function tables. Each table is numbered and unit-generators may reference them, the user specifying the table to be used as one of the input parameters. To do this the unit-generator must include a pointer to the table in its data structure: e.g. `FUNC *ftp;`

In the initialisation pass of the unit-generator the routine `ftfind()` (the code for which is found in FGENS.C) is used to find the pointer to the function table specified by the user in the "score" file. Typical code for this is as follows:

```
if (((ftp = ftfind(p->ifn)) != NULL)
   p->ftp = ftp;
```

The first line finds the pointer to the table, if it exists, and this is then stored in the data memory of the particular instance of the unit-generator.

Appendix B

Transputer Task Configuration

B.1 The CSOUND Processor Pipeline

B.1.1 One Processor Present

CSOUND IS A FIXED TOPOLOGY PROGRAM with needs a definition of the network hardware as well as placement instructions for its constituent tasks. This information is to be found in the file CSOUND.CFG. It needs to be updated if:

- *the hardware configuration is modified*, for example upon the addition of an extra transputer, or

- *the amount of memory on the transputer network is changed*, as might happen if the transputer cards themselves are upgraded.

Table 10 is a listing of a configuration file for a system with one transputer. The second `task` command instructs the configurer to allocate 640KB of data space (stack and heap) for the CS task. This is an appropriate figure for a transputer with 1MB of memory. Note that if more memory is available, this figure should be increased accordingly, and changes should be made to the constant SNDBUFS (defined in file BUF.H) if it is desired to change the quantity of memory reserved for sound buffering.

```
processor host
processor root
wire jumper host[0] root[0]

task afserver ins=1 outs=1
task cs ins=2 outs=2 data=640K
task filter ins=2 outs=2 data=10k

place afserver host
place cs root
place filter root

connect ? filter[0] afserver[0]
connect ? afserver[0] filter[0]
connect ? filter[1] cs[1]
connect ? cs[1] filter[1]
```

Table 10: Configuration file for a Single-transputer System

B.1.2 Several Processors Present

When more than one processor is fitted, the above comments still apply, but multiple copies of the main module must be configured. The source files to achieve this are supplied in a different directory from the one-processor case, as other modifications have also been made (see section 3.10).

Table 11 shows the configuration file for a three processor system similar to the one in use at the University of Durham School of Music. The Unix-compatible make-file will generate the necessary binary executable files:

CS contains the root supervisor process and therefore resides on the transputer connected to the host;

CSM is designed to load intermediate processors in the pipeline (incorporating the MSUPER module) — this is the binary file which is duplicated when an additional processor is added;

CSEL contains the pipeline end module PSUPER and should therefore be loaded onto the final processor.

```
processor host
processor root
processor pip1
processor pip2

wire jumper host[0] root[0]
wire p1 root[1] pip1[2]
wire p2 pip1[1] pip2[2]

task afserver ins=1 outs=1
task cs ins=3 outs=3
task filter ins=2 outs=2 data=10k

task csel ins=1 outs=1
task csm  ins=2 outs=2

place afserver host
place cs root
place filter root
place csm pip1
place csel pip2

connect ? filter[0] afserver[0]
connect ? afserver[0] filter[0]
connect ? filter[1] cs[1]
connect ? cs[1] filter[1]

connect ? cs[2] csm[0]
connect ? csm[0] cs[2]

connect ? csm[1] csel[0]
connect ? csel[0] csm[1]
```

Table 11: A Configuration File for 3 Transputers

```
Task Master Urgent File=pvmain Data=720K
Task Worker File=pvfft Stack=2K Heap=64K Opt=stack Opt=code
```

Table 12: Configuration File for the Distributed Phase Vocoder

It will also be required to change the task and processor declarations if the network is extended; the C compiler manual[61] described the syntax in detail and includes illustrative examples.

Note that the numeric constants in the `connect` statements in table 11 refer to logical channel numbers and must not be changed; it is the `wire` statements which define the network topology.

B.1.3 The Multi-processor Make-file

The file MAKEFILE contains all the information necessary to generate a working CSOUND, using the files with sf CFG and LNK extensions. Switches are provided to enable the generation of a text-only (windowless)[1] version of the program, and to define what sort of processors are used in the network. These switches are fully commented.

B.2 The Distributed Phase Vocoder

Configuring the distributed phase vocoder is altogether far simpler than configuring the processor pipeline. The configuration file for a flood-filled network is but two lines long — see table 12.

The salient points are:

'`Urgent`' means 'run this task at high priority', thus affording the communications task interrupt service status. In systems where there is a task which

[1] A windowless version must be built unless the specially upgraded version of the server VCSERVER is available

performs inter-process arbitration, it is normal to run it at the higher priority level for reasons of efficiency;

'`File=...`' refers to the binary files generated by the make file;

'`Data=720K`' is an appropriate allocation for a transputer with 1MB — the buffer space associated with this memory is allocated by the NETQ module (section 5.5);

'`Opt=stack Opt=code`' specifies that the fast internal RAM of the transputer should be used for the worker to store the stack, and that what remains should be used to store as much worker code as possible.

Appendix C

Transputer Tree Hardware Manual

The transputer tree machine consists of a cross-linked tree of 160 processors. The transputers are 15MHz INMOS T800 devices, fitted with no external memory. Thus each processor has only 4KB of RAM, making a total of 640KB for the whole system.

The hardware is organised as 10 P.C.B.s, each of which contains 16 processors as shown in figure 19. These processors have their links wired as per figure 20. The root P.C.B. has 9 links appearing at the base of the tree; each of these links is connected to another board via the backplane connexions. All links are configured to run at 20Mb/s.

The error flag, links 1, 2, and 3 of all transputers are monitored by a front-panel display. The L.E.D. dot-matrix packages are arranged so that a high signal level illuminates a lamp in the left column for the error flag, or the 3rd, 4th

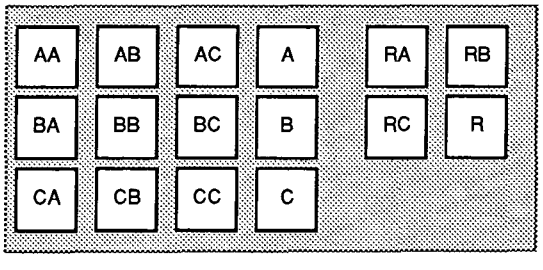


Figure 19: Physical Layout of Processors on each Tree P.C.B.

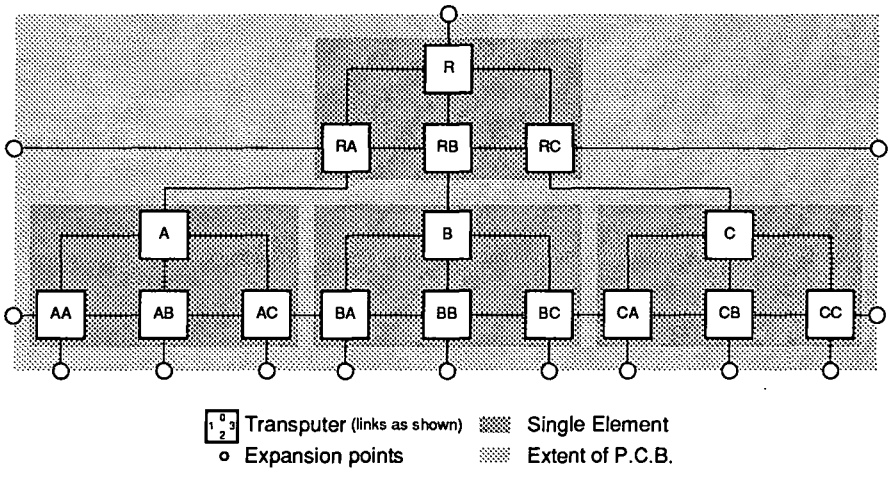


Figure 20: Topology of Transputer Tree with Labelled Processors

or 5th column for the monitored links. The 2nd column is unused. Horizontal organisation is into groups of four lines, separated each by one unused line, with an additional unused line at the very top and bottom of the display. These groups are identified in figure 20 as R (root), A, B and C. Hence reading from the top down, each line monitors processor R, RA, RB, RC, A, AA, ..., CB, CC.

Links shown in the diagram as emerging from the P.C.B. appear at the edge connector for the backplane. The pin designations are given in table 13.

Pin	Label
16a	DOWNNOTANALYSE
16b	UPNOTRESET
17a	UPNOTERROR
17b	UPNOTANALYSE
18a	ROUT0
18b	DOWNNOTERROR
19a	RAOUT1
19b	RIN0
20a	RCOUT3
20b	RAIN1
21a	AAOUT1
21b	RCIN3
22a	AAOUT2
22b	AAIN1
23a	ABOUT2
23b	AAIN2
24a	ACOUT2
24b	ABIN2
25a	BAOUT2
25b	ACIN2
26a	BBOUT2
26b	BAIN2
27a	BCOUT2
27b	BBIN2
28a	CAOUT2
28b	BCIN2
29a	CBOUT2
29b	CAIN2
30a	CCOUT2
30b	CBIN2
31a	CCOUT3
31b	CCIN2
32a	DOWNNOTRESET
32b	CCIN3

Table 13: Tree P.C.B. Edge Connector Pin Designation

Appendix D

Presentations and Publications

Seminars and Lectures

The following presentations arose from the work described in this thesis:

'The Durham Transputer Project' Huddersfield Contemporary Music Festival, November 1989.

'Music Technology for the Composer' Durham University School of Music, December 1989.

'CSOUND *and the Transputer'* University of York Music Department, June 1990.

'Transputers in Non-Real-Time Musical Synthesis' Henndorf am Wallersee, Austria, October 1990.

'Craftsman, Composer, Performer' Queen's University, Belfast, April 1991.

Invited Papers

The following paper was an invited contribution, and was given at the opening of the Computer Teaching Initiative conference:

Bailey, N J, Purvis, A, Manning, P D, & Bowler, I W CSOUND *Inside and Out: Today's Software, Tomorrow's Computers* Proceedings of the Computer Teaching Initiative Conference on Computers in Music and Higher Education, Lancaster, 1990. At press.

Contributed Publications

The following have been published as part of proceedings of the indicated conferences:

As primary author:

Bailey, N J, Purvis, A, Manning, P D, & Bowler, I W *An Implementation of* CSOUND *on the Transputer* Proceedings of the International Conference on the Applications of Transputers, Liverpool, 1989. Published as 'Applications of Transputers 1', Len Freeman and Chris Phillips (ed.) IOS Press (1989) ISBN 90–5199–025–1.

Bailey, N J, Purvis, A, Manning, P D, & Bowler, I W *An Highly Parallel Architecture for Real-time Music Synthesis and Digital Signal Processing Application* Proceedings of the International Computer Music Conference, Glasgow, 1990. pp169–171.

Bailey, N J, Purvis, A, Manning, P D, & Bowler, I W *Concurrent* CSOUND*: Parallel Execution for High-speed Direct Synthesis* Proceedings of the International Computer Music Conference, Glasgow, 1990. pp46–49.

Bailey, N J, Purvis, A, Manning, P D, & Bowler, I W *On the Solution of some Classical Scheduling Problems using Parallel C* Proceedings of the International Conference on the Applications of Transputers, Southampton, 1990. Published as 'Applications of Transputers 2', David J Pritchard and Christopher J Scott (ed.), IOS Press (1990) ISBN 90–5199–035–9.

Bailey, N J, Purvis, A, Manning, P D, & Bowler, I W *Applications of the Phase Vocoder in the Control of Real-time Electronic Musical Instruments* Proceedings of the International Workshop on Man-Machine Interaction in Computer Music, Pisa, June 1991.

Bailey, N J, Purvis, A, Manning, P D, & Bowler, I W *Some Observations on Hierarchical, Multiple-Instruction-Multiple-Data Computers* Proceedings of Euromicro91 conference, Vienna, September 1991. European association for Microprocessing and Microprogramming, PO Box 2346, NL–7301 Apeldoorn, The Netherlands.

As co-author:

Bowler, I W, Manning, P D, Purvis, A, & Bailey, N J *A Transputer-based Additive Synthesis Implementation* Proceedings of the International Computer Music Conference, Ohio 1989. pp58–61.

Bowler, I W, Manning, P D, Purvis, A, & Bailey, N J *On Mapping N Articulation- onto M Synthesiser-Control-Parameters* Proceedings of the International Computer Music Conference, Glasgow, 1990. pp181–184.

Bowler, I W, Manning, P D, Purvis, A, & Bailey, N J *New Techniques for a Real-time Phase Vocoder* Proceedings of the International Computer Music Conference, Glasgow, 1990. pp178–180.