Parallel Computing for Musicians

Invited lecture given at
V Brazilian Symposium on Computer Music, Belo Horizonte, 1998

Eduardo Reck Miranda
SONY CSL - Paris
6 rue Amyot
75005 Paris - France

Abstract

This lecture introduces the fundamentals of parallel computing and discusses its benefits for musicians. It begins by introducing classic parallel architectures and parallel programming strategies. Next, it discusses the benefits of parallel computing for a specific area of computer music research: sound synthesis. The lecture then concludes with a brief discussion on the future of the musician's desktop computer. It is suggested that new paradigms for computer music will emerge from parallel computing technology, but it is up to our community to find the right approaches to them.

1 Introduction

Since computers were invented there have been no major changes in their basic design. The main core of most currently available computers is a central processing unit (CPU), provided with a single memory component, which executes instructions sequentially. This configuration is commonly referred to as *von Neumann architecture* (after its inventor), but for the purposes of this article I use the more generic acronym SISD, for Single Instruction Single Data.

SISD machines call for the notion of sequential programming in which a program consists of a list of instructions to be executed one after the other. Although such machines may now work so fast that they appear to be executing more than one instruction at a time, the sequential programming paradigm still remains for most techniques and tools for software development.

The sequential paradigm works well for most of the ordinary tasks we need the computer for, but scientists are very aware of the limitations of current computer technology. Assuming that computers were programmed to display some sort of intelligent or customary behaviour, the sequential paradigm would fail to model many aspects of

human intelligence and natural systems. For example, particular mental processes seem to be better modelled as a system distributed over thousands of processing units, as an idealised model of brain tissue. It is possible to simulate this distributed type of behaviour on fast SISD machines by modelling a limited number of processing units, but no single processor nowadays can support the number of units required to study the behaviour of the more sophisticated distributed machine models.

Although the speed of SISD machines has increased substantially over the last few years, these improvements are beginning to touch the limits of physics, beyond which any further development would be impossible. In particular, the speed of light is a constant upper boundary for the speed of signals within a computer, and this speed is already proving too slow to allow further increases in the speed of processors. A plausible alternative to meet the demand for increasing performance is to abandon dependence upon single processor hardware and look towards parallel processing techniques.

Parallel computing is commonly associated with the possibility of breaking the speed limits imposed by SISD machines. However much faster computers become, parallel computing also offers interesting programming paradigms for the development of innovative computer music software; see, for example, Holm's paper on the applications of the Communicating Sequential Processes (CSP) technique in music composition and analysis (Holm, 1992). From a musician's perspective, it is this latter aspect of parallel computing that excites interest here.

2 Architectures for parallel computing

Two main approaches are used to build parallel computers: SIMD (for Single Instruction Multiple Data) and MIMD (for Multiple Instructions Multiple Data). Whilst the former employs many processors simultaneously to execute the same program on different data, the latter employs several processors to execute different instructions on different data. Both approaches underpin different programming paradigms and have their own merits and inadequacies.

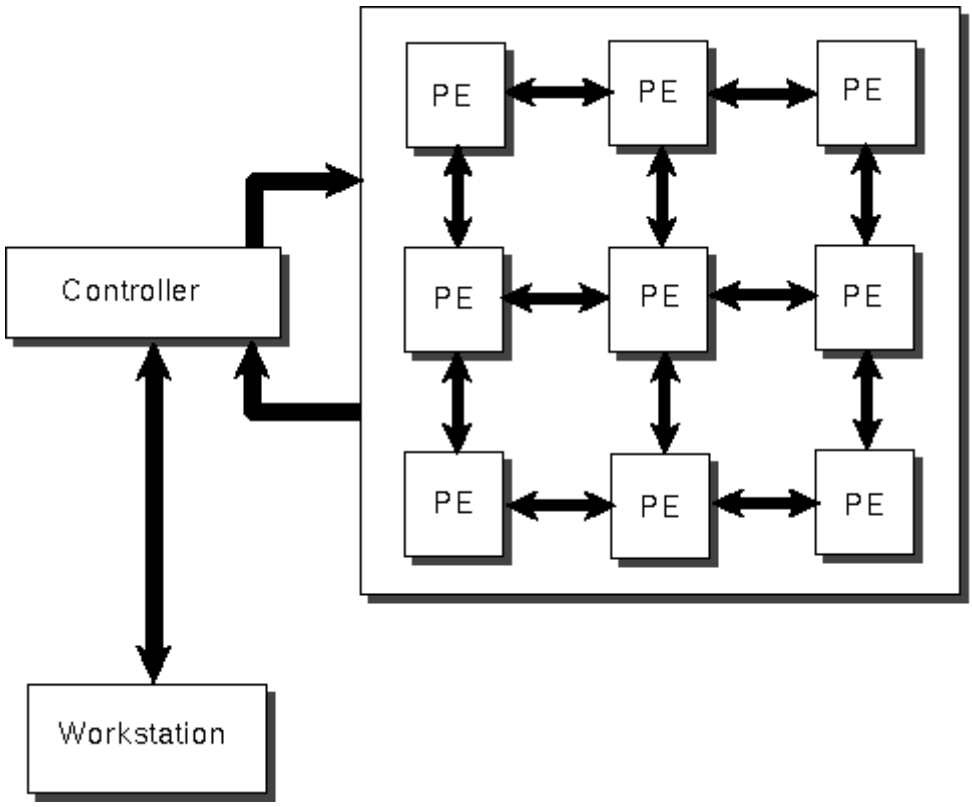
2.1 The SIMD architecture

SIMD-based computers employ a large amount of interconnected *processing elements* (PE) running the same programming instructions concurrently, but working with different data (Figure 1). Each PE has its own local processing memory, but the nature of the PEs and the mode of communication between them varies for different implementations. PEs are usually simpler than conventional processing units used on SISD machines because PEs do not usually require individual instruction fetch mechanisms; this is managed by a master control unit (MCU). Communication between PEs commonly involves an

orthogonal grid of data pathways and each PE can often communicate directly with its four or eight nearest neighbours.

SIMD computers are essentially synchronous models because all PEs execute the same instruction at the same time. The only control the programmer has over an individual PE is to allow or prevent it from executing the instruction. This makes SIMD computers easier to program than MIMD computers (see below) because the control of different programs running on each processor of a MIMD machine becomes very demanding as the number of processors increase. SIMD architectures therefore suit the provision of very large arrays of PEs; current SIMD computers may have as many as 65,536 PEs.

Figure 1:
The SIMD architecture employs a large amount of interconnected processing elements running the same instructions concurrently.



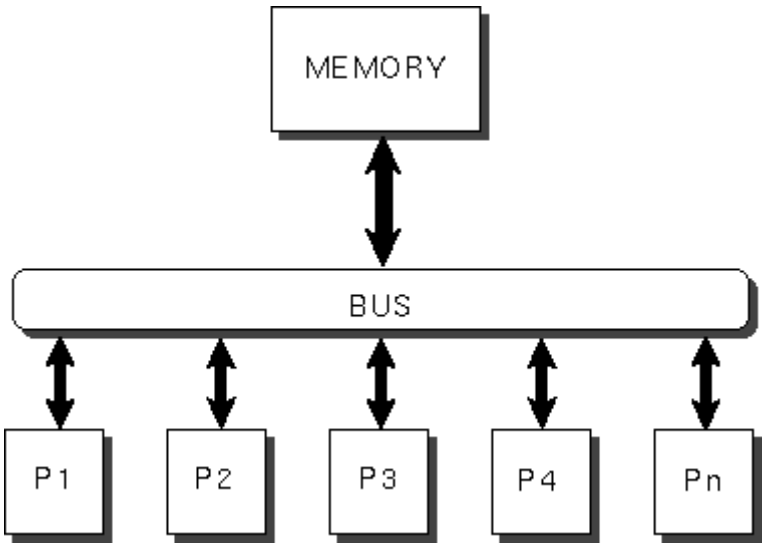
2.2 The MIMD architecture

MIMD-based computers employ independent processors running different programming instructions concurrently and working on different data. There are a number of different

ways to construct MIMD machines, according to the relationship of processors to memory and to the topology of the processors.

Regarding the relationship between processor and memory, we identify two fundamental types: *shared global memory* and *private local memory*. In shared global memory, each processor has a direct link to a single global memory via a common pathway, or *bus* in computer science jargon (Figure 2). In this case, processors only communicate to each other via the global memory. This type of MIMD architecture is sometimes preferable because they are relatively straightforward to program. The major drawback is that the number of processors should be limited to accommodate the capacity of the bus. Otherwise, too many processors competing to access the memory would need onerous mechanisms to prevent data traffic jams.

Figure 2:
In MIMD's shared global memory each processor has a direct link to a single global memory.



In order to employ large amounts of MIMD processors it is necessary to furnish each processor with its own local memory. In this case, each unit is normally encapsulated in a microchip called a *transputer*.

Since MIMD transputers execute different instructions and work on different data, the interconnection between them must be carefully planned. It is utterly unthinkable to connect each transputer to all other transputers. The number of connections rises as the square of the number of units. From a number of possibilities, most customarily used topologies are *binary trees*, *two dimensional grids* and *hypercubes* (3). Sophisticated MIMD implementations allow for user-configuration of topologies to suit particular

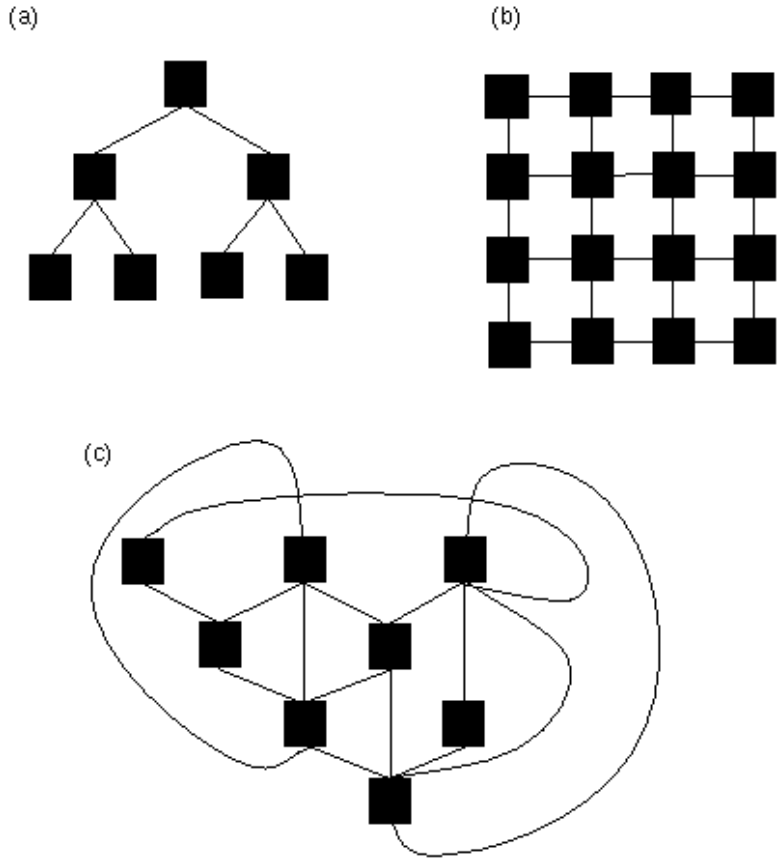
needs.

MIMD computers are essentially asynchronous because each processor or transputer may proceed at their own rate, independently of the behaviour of the others. They may, of course, perform in synchronous mode if required.

3 Parallel programming strategies

It seems likely that in a few years time computers will be based upon a parallel architecture of some kind. Some of these machines will probably make their parallel architecture transparent to high-level programmers and will somehow allow for the processing of sequential programs as well. In this case, sequential programs must to be adapted either manually by a programmer or automatically by the machine itself (e.g. by the operational system).

Figure 3:
Most customarily used MIMD topologies: (a) binary trees, (b) two dimensional grids and (c) hypercubes.



Cutting edge technology and best achievable performance will certainly require software explicitly designed for parallelism. There are still many intrinsically sequential problems that pose difficulties for the design of a parallel processing solution. For example, a melody implies sequences of sounds, which in turn imply sequences of samples; these are best processed sequentially. A program that has been specifically designed for a parallel architecture, will perform unquestionably better than a converted sequential program. This is to say that there certainly are some problems that suit parallelisation better than others - it is up to us software developers to make the right choices.

There are two different but closely related parallel programming strategies: *decomposition* and *parallelisation*. The former involves methods to decompose a sequential program for parallel processing and the latter involves methods to design parallel programs from scratch.

All these strategies are inter-related and very often the solution to a problem is best achieved by applying a combination of them.

3.1 Decomposition

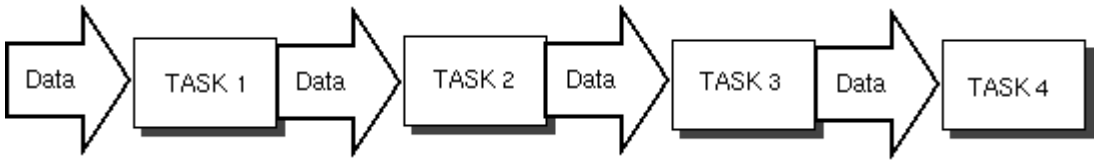
The main objective of decomposition is to reduce the execution time of a sequential program. In general, it is possible to split a sequential program into various parts, some of which would have good potential for concurrent processing. By allocating these parts to various parallel processors, the runtime of the overall program could be drastically reduced.

Decomposition requires caution because the splitting of a program into a large number of processors incurs considerable costs. For example, it should be estimated that individually processed parts might require demanding communication mechanisms to exchange data between them. That is, a large amount of processors does not necessarily lead to good performance.

I identify three main strategies for decomposition: *trivial*, *pipeline* and *geometric*. A trivial decomposition takes place when the original sequential program processes a great amount of data with no interdependencies. In this case, there are no major technical problems to split the task, since parallelism may be achieved simply by running copies of the entire program on various processors concurrently. Conversely, a pipeline decomposition strategy is applied when a program can be split into various modules and allocated to different processors. All input data still passes through each of the modules sequentially but as they move through the pipeline, the modules work simultaneously (Figure 4). The first input element passes through the first stage of the pipeline and after being processed, it then passes on to the next stage of the pipeline. While this first

element is being processed at the second stage, the second input element is fed into the first stage for processing, and so on. All stages work simultaneously because each has its own exclusive processor.

Figure 4:
In pipeline decomposition the input data passes through each of the modules but as they move through the line the modules work simultaneously.



Finally, geometric decomposition splits the data rather than the program. This strategy is suited for those problems where identical operations are applied to different parts of a large data set. As with the case of trivial decomposition, each processor has a copy of the whole program. The difference between trivial and geometric decomposition is that the latter is designed to cope with data dependencies; that is, the result of the operation of one data subset may require information about the operation of other data subsets. Geometric decomposition works best when the data subsets need to interact with their neighbouring subsets. In this case the data is distributed to a certain number of processors arranged in a lattice and the program will have to be furnished with mechanisms for passing messages across the processors.

3.2 Parallelisation

There are two main different approaches to the design of a parallel program: *specialist processing* and *task farm processing*. These approaches have many links to the decomposition categories introduced above and both concepts certainly complement each other.

3.2.1 Specialist processing

Specialist processing achieves parallelism by dividing the problem into specific types of tasks. For example, if the problem is to build a piano, then there will be a variety of specialised tasks such as making the keys, making the string, making the case, etc. Once all pieces are complete, the piano is then assembled. Specialist workers are assigned for each task in order to assure the quality of the components and foster the speed of construction. However, if the goal is to build only one piano, workers may not be able to start working simultaneously because some will have to wait for the outcome of other colleagues in order to start their job. Conversely, if the goal is to manufacture several

pianos, then it may be possible to keep all workers occupied for most of the time by 'pipelining'the tasks (see pipeline decomposition above). In any case, it is essential to establish efficient channels of communication between the workers so that they can co-ordinate their work.

3.2.2 Task farm processing

In task farm processing, the problem is divided into several tasks but not necessarily targeted for specialist processors. To continue the aforementioned metaphor of the piano, there are no specialist workers in the team here, because it is assumed that all of them are capable of carrying out any of the tasks for building a piano. In this case, there is a master who holds a list of tasks to be performed. Each worker takes a task and when it is accomplished he or she comes back and picks another one. Communication occurs mostly between a worker and the master; for the sake of maximum performance, workers are not encouraged to talk to each other because their attention must not be diverted from main task.

4 Benefits for music: sound synthesis

The benefits of parallel computation for music can be enormous but here I will illustrate only its application in sound synthesis systems. In several ways, parallel computing concepts already have started making their way into synthesis technology, from the design of dedicated VLSI (Very Large Scale Integration) chips to the implementation of new paradigms for sound synthesis.

4.1 Signal processing level

Dedicated digital signal processors (called as DSP) with multiple functional units and pipeline methods have been encapsulated into VLSI chips and today are commonly found in a variety of hardware for sound synthesis; e.g. the Betel Orionis system developed at the University of Rome in Italy [Nottoli and Costantini, 1998].

The functioning of a DSP is driven by a set of instructions that is loaded into its own memory from a host computer. The DSP then cycles continuously through these instructions until it receives a halting message; a sample is produced at each cycle of the DSP. Special DSP arrangements may form arrays of DSP for processing blocks of samples at once. In this case, the output for each cycle is an array of samples.

Also, parallel configurations of specially designed general purpose microprocessors based on RISC technology (Reduced Instruction Set Computer) have been used on several

occasions; e.g. the IRCAM Musical Workstation board developed in France [Lindermann et al., 1991].

4.2 Software synthesis programming level

On the software synthesis programming front, there have been a few attempts to decompose existing synthesis programming languages for concurrent processing.

Remarkable results have been reported by composer Peter Manning and his collaborators at Durham University, in England, who have managed to run Csound concurrently on a parallel machine [Bailey et al., 1990]. Csound is a synthesis programming language in which instruments are designed by connecting many different synthesis units. One or more instruments are saved in a file called the *orchestra*. Then the parameters to control the instruments of the orchestra are specified on an associated file called the *score* [Miranda, 1998]. This score file is organised in such a way that each line corresponds to a stream of parameters to produce a sound event (or note) on one of the instruments of the orchestra. For instance:

```
i1  0  1.0  440  90
i2  1  0.5  220  80
...
```

In the example above, each line specifies five parameters for two different instruments, including the 'names' of the instruments (i.e., *i*1 and *i*2, respectively), start time (in seconds), duration of the note (in seconds), frequency (in Hz) and amplitude (in dB). Each line produces a note.

The parallel implementation of the Durham team is inspired by the *task farm* paradigm discussed earlier. A copy of the entire orchestra is distributed to each processor and the score is considered as a list of tasks for the processors; that is, each processor picks a line from the score for processing.

4.3 Synthesis modelling level

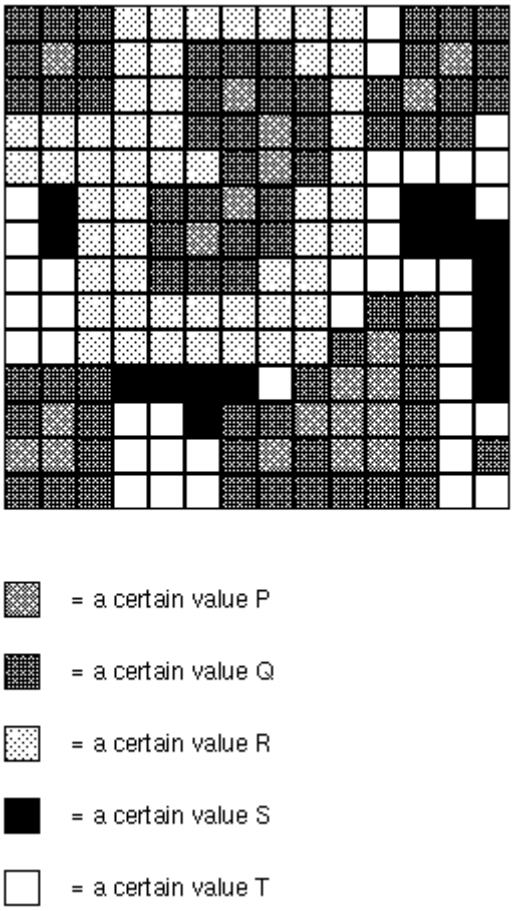
At this level a parallel computing paradigm is normally imbedded in the synthesis model itself. As an example I cite *Chaosynth*, a synthesis system developed by myself in collaboration with the engineers of the Edinburgh Parallel Computing Centre (EPCC), in Scotland [Miranda, 1994] [Miranda, 1995].

In short, *Chaosynth* uses cellular automata (CA) to control the parameters of a granular synthesis instrument. The granular synthesis of sounds involves the production of sequences of thousands of short sonic particles (for example, 35 milliseconds) in order to

form larger sound events.

CA are mathematical models of dynamic systems in which space and time are discrete and quantities take on a finite set of discrete values. CA are often represented as a regular array with a discrete variable at each site, referred to as a cell. The state of the CA is specified by the values of the variables at each cell. It evolves in synchronisation with the tick of an imaginary clock, according to a global function that determines the value of a cell based upon the value of its neighbourhood. As implemented on a computer, the cells are represented as a grid of tiny rectangles, whose states are indicated by different colours (Figure 5).

Figure 5:
The CA cells are represented as a grid of tiny rectangles, whose states are indicated by different colours.

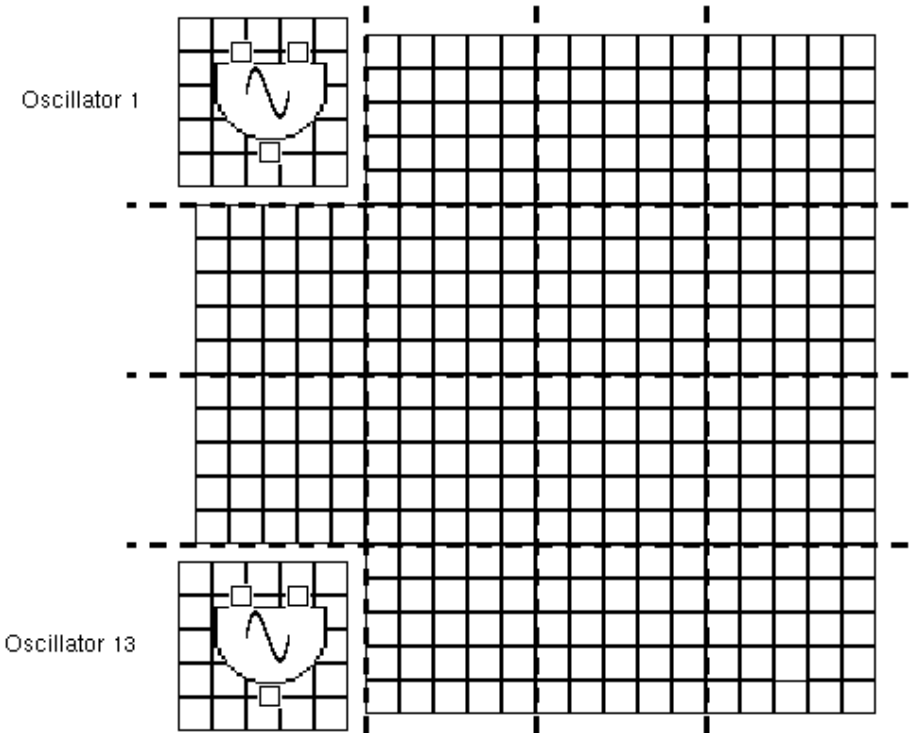


The algorithm of the CA used in *Chaosynth* is fully explained in [Miranda 1995]. Each sound granule produced by *Chaosynth* contains a number of partials produced by a bank of oscillators, as with additive synthesis. The frequencies of the partials are determined

according to the evolution of the CA. The states of the CA cells represent frequency values rather than colours. The oscillators are associated to a group of cells, and in this case the frequency for each oscillator is established by the arithmetic mean of the frequencies of the cells associated to the respective oscillator. At each cycle of the CA, *Chaosynth* produces one sound granule and as the CA evolves, the components of each granule change therein. The size of the grid, the number of oscillators and other parameters for the CA are all configurable by the user.

CA are intrinsically suited for parallel processing; they are arrangements of cells, each of which is updated every cycle, according to a specific global rule that takes into account the state of the neighbouring cells. This is a typical problem for *geometrical decomposition* with message passing. In this case, the grid is subdivided into a number of contiguous rectangular portions in such a way that all of them are handled concurrently by different processors. The oscillators of *Chaosynth* are therefore computed in parallel (Figure 6).

Figure 6:
CA are suited for geometrical decomposition.



5 Conclusion

Since the invention of the microprocessor, computers have become twice as fast almost every two years; the microprocessor of your average personal computer nowadays encapsulates more than 7 millions of transistors in a single silicon chip. Because of this explosive progress, current computers are millions of times more powerful than their crude ancestors, but integrated circuit technology is running up against its limits. Advanced lithographic techniques would certainly be able handle circuitry designs 1/100 the size of what is currently available, but at this scale - at the atom level - integrated circuits cannot function properly. If computers are to become more powerful and yet smaller, new technology must either replace or supplement current trends.

Research in quantum-mechanical computer technology is very promising but until now scientists did not manage to actually build a suitable quantum processor for industrialisation. In meantime, the industry is supplementing the power of current microchip technology with parallel computing strategies; e.g. pipelining is becoming increasingly popular in microchip design.

In the future, musicians may never want to know whether his or her computer employs parallel processing concepts. All the same, parallel software engineering techniques are becoming more popular and new parallel paradigms for computer music will certainly continue to emerge. It is up to our community to develop them.

References

[Bailey et al. 1990] Bailey, N., Purvis, A., Bowler, I. and Manning, P., "Concurrent Csound: Parallel Execution for High Speed Direct Synthesis", *Proceedings of the International Computer Music Conference*, Glasgow: ICMA, UK.
[Holm, 1992] Holm, F., "Machine tongues XIV: CSP-Communicating Sequential Processes", *Computer Music Journal*, 16(1):25-33, MIT Press, USA.
[Lindermann et al. 1991] Lindermann, E., Dechelle, F., Smith, B. and Starkier, M., "The architecture of the IRCAM musical workstation", *Computer Music Journal*, 15(3):41-49, MIT Press, USA.
[Miranda, 1994] Miranda, E. R., "Chaosynth - computer music meets high-performance computing", *Supercomputer*, 11(1):16-23, ASFRA, The Netherlands.
[Miranda, 1995] Miranda, E. R, "Chaosynth: Um sistema que utiliza um aut mato cellular para sintetizar part culas s nicas", *Anais do II Simp sio Brasileiro de Computa o e M sica*, Porto Alegre: Instituto de Inform tica da UFRGS, pp. 205-212, Brasil.
[Miranda, 1998] Miranda, E. R., *Computer Sound Synthesis for the Electronic Musician*, Oxford (UK): Focal Press.

[Nottoli and Costantini, 1998], Nottoli, G. and Costantini, G., "Betel Orionis: a real-time, multiprocessing sound synthesis system", *Actes des Journ es d'Informatique Musicale 98*, Marseille: Publications du LMA no. 148, pp. E3-1-E3-5, France.