Trace2au
Audio Monitoring Tools for Parallel Programs

Jean-Yves Peterschmitt *
LIP
Unité de Recherche Associée 1398 du CNRS
Ecole Normale Supérieure de Lyon
46, allée d'Italie
69364 Lyon Cedex 07
France

Bernard Tourancheau †‡
University of Tennessee
Computer Science Department
Knoxville, TN 37996-1301
USA
e-mail: btouranc@lip.ens-lyon.fr

August 1993

Abstract

It is not easy to reach the best performances you can expect of a parallel computer. We therefore have to use monitoring programs to study the performances of parallel programs. We introduce here a way to generate sound in real-time on a workstation, with no additional hardware, and we apply it to such monitoring programs.

Keywords: Monitoring, Parallelism, Sound on a workstation, Sonification

1 Introduction

Monitoring the behavior and performances of massively parallel programs has proved to be quite difficult. We have to deal with two major problems: gathering the monitoring data and using it to understand the behavior of the studied parallel program to, hopefully, increase its performances.

This paper will focus on the latter problem. More precisely, we will see how sound can be generated in real-time[1] on a workstation, and used to extract relevant information from the monitoring trace files. We therefore suppose that the monitoring data is already available, and that we have one way or another to access it.

The idea of using sound in a program is not new. We have been used to hearing our computers beep to catch our attention when we were getting new mail, or committing some kind of error.

*This work was supported by Archipel SA and MRE under grant 974, DRET and the PRC C^3.

†On leave from LIP, CNRS URA 1398, ENS Lyon, 46 allée d'Italie, 69364 Lyon Cedex 07, France.

‡This work was supported in part by CNRS-NSF grant 950.223/07, Archipel SA and MRE under grant 974, the National Science Foundation under grant ASC-8715728, the National Science Foundation Science and Technology Center Cooperative Agreement CCR-8809615, the DARPA and ARO under contract DAAL03-91-C-0047, PRC C^3, and DRET.

[1]We mean by *real-time* that the sound can be played as soon as it is generated, without having to store it in a temporary file.

Yet, the use of sound is new in the sense that sound chips have become standard equipment on workstations only very recently. Using fancy sounds is no more the privilege of personal computers. We are only beginning to use sound, and we will probably benefit from the steady breakthroughs coming from the multimedia field.

Very few programs use the newly available sound capabilities of workstations. We believe that the advances made in this field are slowed down by the lack of users actually using audio enhanced programs in their daily work. Potential users are still quite skeptical about the advantages that these kinds of programs could bring them.

We present in this paper our attempt to make small, stand-alone programs, that use sound to convey monitoring information. These programs can generate sound in real-time, and can be easily modified to suit the needs of a given user (e.g. doing on-line monitoring). All we need is a processor that can compute sound samples at a rate higher than the frequency supported by the sound chip.

After the introduction, we study in section two how to convey data with sound. In the third section, we present what can be done with the built-in sound chip of Sun's SPARCstations. In the next section, we show the kinds of sound waves we generate to analyze the monitoring data. The fifth section presents a prototyping tool that allows the user to determine how well sounds relate to each others. The last section gives more details about the actual audio monitoring programs we have implemented. We eventually conclude by explaining that using our basic tools, it is straightforward to build any kind of dedicated tool and study any kind of traced event. These tools should be able to fit the needs of anybody who tries to improve a massively parallel program.

We hope that this paper will also help programmers add sound generating procedures to their programs. Workstations such as Sun's SPARCstations have become widely available, and we do believe that not using their sound chips would be a waste of resources. Moreover, it will be easy for all the people having access to Sun's SPARCstations to use the sound programs released for these stations.

2 Conveying data with sound

Scientific visualization is the science (maybe we should still say *the art*) of using visual displays to extract information from huge raw data files, and to get a better insight into whatever simulation or experiment produced the data. In this field, it is common to use the term *dimension*, when we speak about the number of independent variables that we can display at the same time. People always look for new ways of displaying more dimensions at the same time. Yet, you must be careful to keep the resulting display clear enough (a display might quickly get confusing). Using a 3D-plot gives you three dimensions. Color is an additional dimension, time (animation) is yet another dimension, and so on.

Therefore, using sound in a scientific visualization application will add another (set of) dimension(s) to the existing ones. Several papers have already studied this subject (see for example [BH92, FJA91, Mad92, RAM+92, ZT92]). Using sound in such a context has been coined *sonification* or *auralization*. Concerning monitoring, [FJA91] focuses on the mapping of events to the MIDI format[2], and uses the resulting sounds in parallel with ParaGraph ([HE91]). [Mad92] introduces a more general purpose sonification tool, and uses it in the Pablo monitoring environment (see also [RAM+92]). This tool allows the user to switch easily between using MIDI or sound on a Sun's SPARCstation.

[2] *Musical Instrument Digital Interface*, a communication protocol that allows sound synthesizers to be interconnected and computer controlled [Ass88].

What is maybe most important is the fact that the first set of dimensions described above relies on seeing, whereas the sound related dimensions rely on hearing. These two sets of dimensions are radically *orthogonal* because they use two different senses, and can therefore convey information to our brain in parallel. Moreover, one of the advantages of sound is that we can process part of the information in a passive manner (i.e. without intently listening to it). This advantage has been detailed in [ZT92].

To convey information, using sound, we can play with the *basic parameters* of sound, and have them change over time:

The *pitch* is related to the frequency, and can be used to convey a numerical value. The human ear can theoretically hear frequencies in the range [20...20,000] Hz. In practise, we cannot generate such a wide range on a computer (only [100...3,500] Hz on a Sun's SPARCstation).

The *timbre* depends on the waveform of a sound. Two sounds of the same pitch, having a different timbre, will sound different. This is what allows us to make the difference between two musical instruments.

The *amplitude* of the waveform. A feature of the human ear is that it is actually sensitive to the *intensity* of a sound rather than to its amplitude. The intensity is a linear function of the square of both the amplitude and the frequency. Therefore, a sound with a high pitch will seem louder than a sound with the same amplitude but with a lower pitch.

We can adjust the variation over time of the intensity of a sound. This is called the *envelope* of the sound, and is divided into three parts: attack, sustain and decay.

The *duration* of a sound determines how long a given sound is played. It is natural to use this parameter to represent the length of an event. By using this parameter in the right way, we can often get an idea of a specific rhythm related to the studied data set.

We can also use *secondary characteristics* of sound:

- a given sound mixes well with another sound. The resulting sound is at the same time different from the two original sounds, and still retains enough characteristics of them to allow us to recognize them.
- a sound can be located in space. Using two or more speakers, we can make the placement of a sound in space change over time.

Note that for obvious technical reasons, we can not achieve all these sound effects on a standard workstation. We will talk about this in the next section.

As it is emphasized in [BH92], sound can be used for four different reasons in a scientific application: reinforcing existing visual displays, conveying distinctive patterns or signatures (that are not obvious with mere displays), replacing displays or signaling exceptional conditions.

Unfortunately, there are still some drawbacks in the use of sound! A few people can recognize the absolute pitch of a tone, but most people can only assess pitch intervals[3]. There is the same problem with the intensity: people can tell whether a sound is loud, or louder than another one, but that is about all they can say. Nobody can determine precisely the numerical value of a sound

[3]The human ear is in fact sensitive to logarithmic changes in frequency. This makes it hard, if not impossible, to make the difference between two sounds that differ only by a small change in pitch. If you only change the pitch slightly, you need to change the timbre to be able to make the difference between two sounds.

parameter. We have the same problems with the perception of colors, but in this case, we can at least display a color scale on the side of a graphical display. Unfortunately, there is no such thing as a *sound scale* that could be used in the same way as a color scale. We can use *xplayer*, the prototyping tool described in section 5, to determine what a given pitch sounds like, but we are not able yet to play at the same time this reference pitch and the studied sound. To be able to do this in the best way, we would need to have access to two different channels at the same time. This could be done on a single workstation offering stereo audio support, or on two different workstations[4]. Remember that if you work with a single workstation, the channels are not completely independent, in the sense that only one program can access the audio device at a time. Therefore, the same sound program would have to compute and supply the reference pitch, as well as the studied sound. What we can already do, with the currently available programs, is to create a trace sound file, load it into xplayer, then create a sound having a given pitch with xplayer and mix those two sounds. We are then able to listen to these two sounds at the same time, on a single channel.

We believe that the users will be able to understand increasingly complex parallel programs, thanks to the use of sound, with some appropriate training. The more dimensions you can *use/display* at the same time, the more processors and parameters you can study. This will be crucial when we will have to study truly massively parallel programs.

3 Sound on Sun's SPARCstations

Sound can be generated in real-time on any workstation having a processor that can compute sound samples at a rate higher than the frequency supported by the sound chip. According to [VR93], the following manufacturers sell workstations with built-in sound capabilities: DEC, Hewlett-Packard, NeXT, SGI, Sony and Sun. We have worked with the Sun's SPARCstations that were available in our laboratory, but our programs could be easily modified to run on other workstations.

Sound programs on a Sun's SPARCstation take advantage of the built-in digital to analog converter. With this, they can play a sound of 8,000 samples per second (8 KHz), on a single channel. This provides audio data quality equivalent to standard telephone quality[5].

The data supplied to the sound chip is compressed with μ-law encoding. In this encoding algorithm, the spacing of sample intervals is close to linear at low amplitudes, but is closer to logarithmic at high amplitudes. Therefore, instead of supplying the chip with 14-bit samples, we just send it 8-bit samples. Thus, a 1 minute sound will use around 470 Kbytes instead of 820 Kbytes. We don't have access to the source of the format converting functions provided in the sound handling library, but we can assume that compressing the sound (going from the actual amplitude values to the μ-law encoded values) will take longer than the reverse operation.

Now that we have the technical details (see also [Sun91, Sun92a, VR93]), let us see what kind of sound we can have on Sun's SPARCstations, with regard to the parameters discussed in the previous section. We also have to keep in mind that we want to make programs that will have low memory requirements, and be fast enough to produce sound in real-time[6].

- according to the Nyquist theorem[7], the highest pitch we will be able to get will be of 4 KHz.

[4] With the sound program running on the first one, and xplayer running on the other one.

[5] Sun has announced in [Sun92b] that the SPARCstation 10 would be capable of simultaneous input and output of 16-bit stereo audio at rates up to 48 KHz. It would, among other things, support the standard CD sampling rates (16-bit, 44.1 KHz), and Digital Audio Tape rates (16-bit 48 KHz).

[6] Producing sound in real time means being able to create at least 8,000 samples per second.

[7] This theorem states that to reproduce a signal, the sampling rate must be greater than twice the bandwidth of the input signal.

- if we want to use sounds with several different timbre, we will need a lot of storage space: 8,000 bytes per second of sound, a few milliseconds of sound for each note of a given instrument, all this multiplied by the number of notes and the number of instruments...

  Of course, we could try to *create* sounds having different timbre, but this would be computation intensive, and we could not do it in real-time.

- we can mix several sounds together but, assuming that the sounds to be mixed are available in uncompressed form, we will still have to perform a lot of μ-law encoding operations (one call for each sample, with the current converting function). This may cost a lot of CPU power.

- we can't locate a sound in space, because we have only one output channel. Of course, it could be possible to have two nearby stations work together to produce a stereo sound, but this is beyond the scope of this project. We will probably have to consider this in our future work, if we want to be able to study massively parallel computers more accurately.

4 Implementation

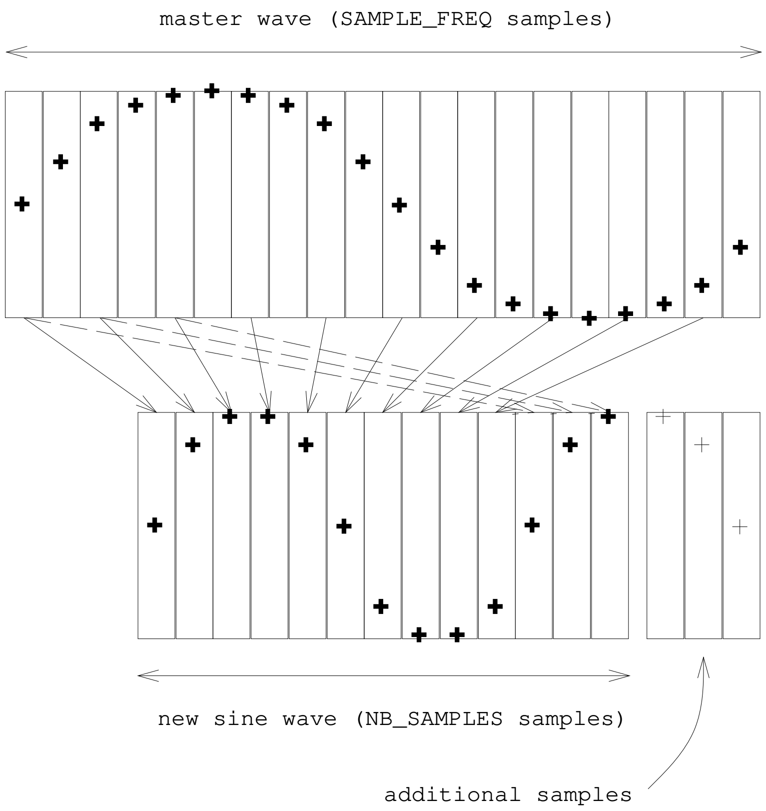

Figure 1: Computing a new wave

Our goal in this project was to create small stand-alone programs. We could therefore not afford to rely on a large library of recorded sounds, digitized off-line, to produce the final monitoring sounds. Moreover, we also wanted to be able to produce the sounds in real-time, to avoid having to store them in a huge temporary file, thus allowing a very scalable implementation. Our programs needed to be fast and have at the same time low memory and disk-space requirements.

This approach was a bit different from the usual one, where you choose to optimize either the speed or the the memory usage. We were nevertheless lucky, in that we got interesting enough results with seemingly very simple sound waves: basic sine waves.

4.1 Creating and using the *master sine wave*

We start by computing the *master sine wave*. It is an array of `SAMPLE_FREQ`[8] samples, where we store the samples corresponding to a sine wave of exactly one hertz (i.e. the first element of the array is $sin(0)$, and the last is $sin(2\pi * (1 - \frac{1}{SAMPLE\_FREQ}))$). Depending on what kind of sound we plan to create, we either store the floating point values of the samples, or directly their μ-law encoded values. If we don't plan to perform additional computations on the samples, it is best to store them in the μ-law form so that we don't have to lose time doing the compression when we start generating the sound.

Suppose now that we want to create a sine wave of frequency `f`, lasting `T` seconds. We will have to generate `NB_SAMPLES = SAMPLE_FREQ * T` samples.

We deduce the new sine wave from the master sine wave by copying selected samples from the master wave to the array where we store the new wave. Since the master wave has a frequency of 1 Hz, we just have to copy every `f`th sample to the new array to get a sine wave of frequency `f`. We copy `NB_SAMPLES` samples this way. The pointer to the sample to be copied from the master sine wave is computed modulo `SAMPLE_FREQ`, so that we never get out of range[9].

Note that, even if we play the sound as soon as the samples are computed, we usually store 1 second worth of samples (SAMPLE_FREQ samples) in a buffer array before sending them to the audio device. Otherwise, writing one sample at a time to the audio device would be too slow.

Figure 1 shows how we get from the master sine wave to a higher frequency wave. On this example, we retain every other sample of the master wave, thus doubling the frequency of the original wave. Two things are worth noticing :

- when we reach the end of the master wave array (plain lines), we start over near the beginning of the array (dashed lines), and we keep on doing this until we have enough samples.

- if we plan on storing the resulting wave in a new array, and cycling through this array to play a continuous sound of the specified frequency, we are likely to get sharp *clicks* in the speaker. These clicks come from the discontinuity between the value of the last sample, and the value of the first sample (in the new wave).

  That is why, in such case, we compute some *additional samples* (bottom left of the figure), so that the value of the last sample is as close as possible to the one of the first sample. This way, we will ensure the continuity of the wave. Note that, to get even better results, we should try to ensure the continuity of the first derivative of the wave. As shown on figure 1, we have only ensured the the continuity of the function, in our program: $f(0) = f(n)$, but $f^{(1)}(0) \neq f^{(1)}(n)$.

4.2 Mapping events to frequencies

Now that we are able to get sounds of different frequencies, it is time to decide how we should map them to the events or values we want to study. The *events* we study are details about the parallel program execution that we can deduce from the monitoring data: for instance the time when the

[8] In our case `SAMPLE_FREQ = 8,000`.

[9] We can cycle through this array this way because the master array holds a whole period of a sine function.

`SEND`s and the `RECEIVE`s take place. We can also choose to study the change of numerical values over time (e.g. how many messages have been sent but not received, at a given time). We assume that we have N different events or values, and that the range of frequencies we can actually get on Sun's SPARCstations is $[f_{min} ... f_{max}]$[10].

It would seem natural to use the usual linear mapping between events and frequencies. Yet, we will rather use a logarithmic mapping, because the human ear is sensitive to logarithmic changes in frequency. We will therefore have the following mapping :

$f_n = \alpha \ * \ f_{n-1}$, where α is a constant and $f_1 = f_{min}$, $f_N = f_{max}$

or $f_n = \alpha^{n-1} \ * \ f_{min}$, with $\alpha = \sqrt[N-1]{\frac{f_{max}}{f_{min}}}$

4.3 Changing the amplitude

When we create a sound, we also have to choose its amplitude, keeping in mind that the absolute value of the samples must be smaller than `1.0`. It is straightforward to imagine that a given sine wave will sound louder, when played, if its amplitude is closer to `1.0`. Unfortunately, the sound also seem louder if, for a given amplitude, we increase the frequency. That is why we rather have to adjust the *intensity* of the sine wave : two sounds having the same intensity will seem to be of equal loudness.

We have $I = \lambda \ * \ a^2 \ * \ f^2$

	I	:	intensity of the sound
With	λ	:	constant
	a	:	amplitude of the sine wave
	f	:	frequency

Therefore, if we want to generate several waves of equal intensity, we have to make sure that the product $a \ * \ f$ remains constant. On the other hand, we have to increase the value of this product if we want the sound to grow louder.

4.4 Mixing sounds

It is not easy to quickly find the best way to mix sounds and get the kind of sound effect you are looking for. Of course, it always involves the serial addition of the samples[11] of all the sounds you want to mix. But then, you have to take several problems into account :

- how should the individual sounds be normalized? We must make sure that the samples resulting from the addition still have an absolute value smaller than `1.0`.
- what should we do if we want all the mixed sine waves appear to have the same intensity?
- what should we do if we want one wave to appear louder than the other ones?

The correct *mixing formula* is usually found by trial and error, and looks like the weighted sum of the different sounds:

$$new\_sample[t] = \frac{\sum_{first}^{lastsound} \lambda_i \ * \ sample_i[t]}{\sum_i \lambda_i}$$

The formula can get more complex, depending on what we want to get. For example, in one of our programs, we wanted to mix several sine waves, $wave_i[t]$, of amplitude `1.0`. All the waves being

[10] We have determined that the best results were obtained with $f_{min} = 100$ Hz and $f_{max} = 3500$ Hz.

[11] It is important to add the actual numerical values of the samples, and not their μ-law encoded form!

played with the same intensity, and have the generated sound seem louder when more processors were sending. We eventually used the following formula:

$$new\_sample[t] = \frac{\sum_{sending\ procs} a_i^2 \ * \ wave_i[t]}{(\sum_{sending\ procs} a_i) \ * \ (1 + 7 \ * \ \frac{nb\_proc \ - \ nb\_sending}{nb\_proc})}$$

where a_i was such that all the waves had the same intensity. Note that we chose to use the number 7 in the previous formula, so that we would get 8 different loudness levels, depending on the number of sending processors. We could have chosen another number of levels, but this was what yielded the best results.

There is another problem that should not be overlooked when you mix sounds: it takes a lot of CPU power to mix sounds. Even if you assume that you already have the samples to be mixed in the correct uncompressed form, you will still have to perform many computations before you get the final result. You need two nested loops, one loop for the time (samples), and a computation intensive inner loop. The inner loop will perform as many floating-point additions as there are sounds to mix. Besides, you also need several floating-point operations to normalize the sound. At last, when you have the final sample, you have to convert it to the μ-law form before you can send it to the sound chip. It is therefore quite important to take all this into account, if you still want your program to generate a sound in real-time.

5 The prototyping tool

In the course of the project, we developed an audio prototyping tool, *xplayer*. We wanted to be able to determine how well different sounds related to each other.

xplayer is a kind of interactive *audio workbench* that allows the user to experiment with sounds. Figure 2 shows some typical windows of xplayer. The user can work with several different sounds at a time. The sounds are placed in five *slots*, that can be selected/deselected to determine what sounds are to be played. The tool allows you to:

- load/save a sound into/from one of the five available slots.
- create a sound of a given frequency and amplitude in one of the slots.
- mix sounds from two different slots, and store the resulting sound in a third slot. Notice that you have to specify how much of each sound you want.
- select the sounds you want to play sequentially (boxes with a tick mark in the upper left window), and play them. When you are dealing with very small sounds (lasting less than a second), you can also adjust the speed at which they will be played.

This tool was the first sound program we developed. Our aim was to learn how to use sound on a Sun's SPARCstation, and solve the lack of available sound programs for this kind of workstation. We have therefore programmed it in such a way that it could easily be given additional features (*Special* menu in the main window), but we did not spend time working on features that we did not really needed for our project.

6 The AudioTrace programs

We will now talk about the actual monitoring programs we have worked on.

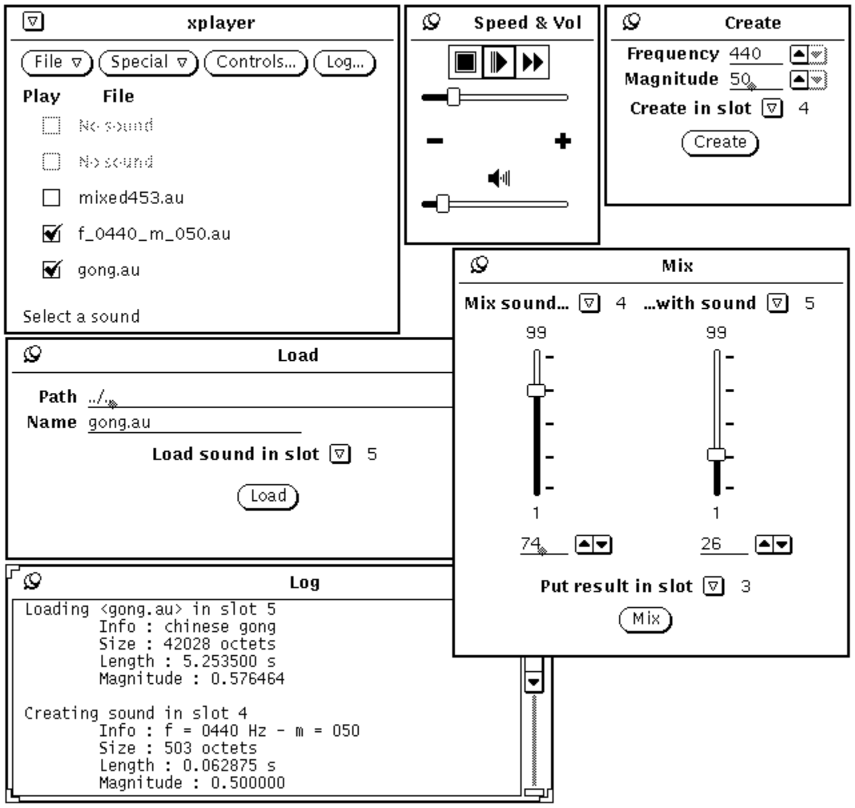

Figure 2: The prototyping tool

6.1 Common points

All of our audio monitoring programs have the same structure, and share therefore several common features:

- the source code is small, and the resulting executable is small as well (less than 50 Kbytes)[12]. This shows that adding the same kind of sounds to existing programs will not make these programs much bigger.
- the input is a trace file or a trace stream. The content of the supplied trace data is sorted according to increasing timestamps. We decided to use the same kind of trace file we were using with vol_tmspg (our customized version of ParaGraph for the ARCHIPEL Volvox machine [HE91, Env92, vRT92]): ASCII "`.trf`" files.

  The kind of trace file used can be easily modified. All we need is a way to know when the interesting events (`SEND`s and `RECEIVE`s in our current tools) take place.
- the output is a "`.au`" sound[13]. The sound is created with a valid audio header, and can be either played directly, or stored for future use.

[12] We don't use a graphical user interface, just command line arguments. We will however soon add a user interface to our programs, to make their use more intuitive.

[13] Audio files that can be played on a Sun's SPARCstation usually have the "`.au`" extension. For more details about the file structure and the file header, see [VR93].

- the programs are fast. This allows us to generate and play the created sound on the fly. Thus, we only need a way to access trace data (no storage problem, with our scalable implementation), and have no need to store the (usually) huge resulting sound file.

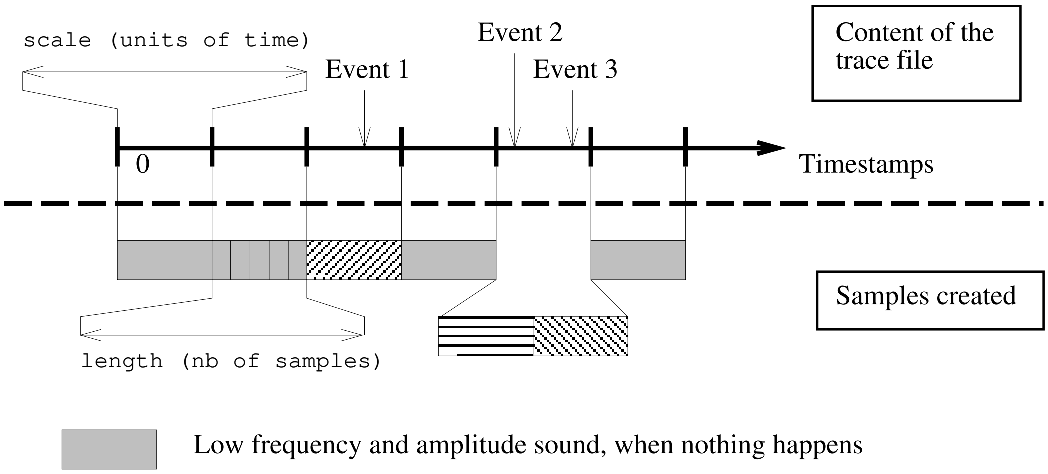

Figure 3: Relation between the execution time and the sound duration

- the duration of the created sound is proportional to the actual execution time of the studied parallel program. Therefore, the relative places of the sound events in the generated sound will be the same as in the actual execution of the parallel program.

  The duration of the generated sound depends on two parameters, `length` and `scale`, as shown in figure 3. At the beginning of the program, the time is set to `0`. It is then incremented by `scale` units of time at each step. This is called the *replay time*. At the same time, the trace file is read sequentially, in search of *interesting* events[14].

  At a given replay time, we are always in one of these two cases:

  - no interesting event took place between the previous and the current replay time, and we generate `length` samples of a sound having a low frequency and amplitude (i.e. a sound that won't be heard, unless the loudness of the speaker is set to a high value).
  - one or more interesting events took place, and we generate as many consecutive sounds of `length` samples as there were interesting events.

6.2 Using the programs

The programs all work the same way, and have a name in the form `tr_xxx`, where `xxx` specifies the type of the program ("`tr_`" means that we work with `TR`ace files). They have four common parameters, specified on the command line:

file is a trace file ("`.trf`" ASCII file).

nb is the number of events we want to map to frequencies. *nb* can be, for instance, the total number of processors involved in the parallel program. When possible, `nb` should be given the smallest

[14] What we mean by *interesting* depends on what we are studying.

possible value (i.e. the exact number of events), so that we have as much difference as possible between two consecutive frequencies[15].

scale is the *replay time* unit (as it was explained above). In our case, scale is a time in microseconds, because the timestamps in the trace files we use are given in microseconds.

length is the number of samples created for each event. As shown on figure 3, we may sometimes have several events taking place during `scale` microseconds, if `scale` is too big. In this case, we create an integer number of `length` samples for `scale` units of time. Note that a sound composed of `length` samples will last `length / SAMPLE_FREQ`[16] seconds.

In other words, if the trace file contains `n` events that took place less than `scale` units of time apart, `n * length` samples will be generated for just *one* `scale` interval. When this happens, the total length of the created sound is no longer exactly proportional to the execution time of the parallel program. On the other hand, trace files can be quite long, and we don't lose the relative order of the events (increasing timestamps), even if we are no more exactly proportional. Therefore, it can often be useful to start with big values of `scale` to quickly get a rough idea of what happened in the program. Common starting values of these parameters are: `scale > 100 milliseconds` and `length < 200 samples`.

If we want to play the sound at the same time it is created, we use:

`cat`[17] `file.trf | tr_xxx nb length scale | play`[18]

Otherwise, to store the generated sound in a sound file, we rather type:

```
cat file.trf  |  tr_xxx nb length scale > file.au
```

We have three programs available. Others could be easily and quickly deduced from the available ones.

tr_send : when a processor sends a message, `tr_send` plays a *beep* at the frequency associated with this processor.

This allows the user to determine how many messages were sent during the execution, and when. This quickly gives an idea of the different phases of the algorithm. It also shows the iterative communication patterns, or the lack of such patterns. From a sound frequency (or *pitch*) point of view, the user can easily determine if processors or groups of processors communicate more than others. This knowledge can then be used to find a better communication balance. If known groups of processors act the same way during the execution, it may be interesting to change the mapping of their associated frequencies, and get this way an even better insight of what happened during the execution. It is easy to determine an interesting mapping with *xplayer*, the prototyping tool described in section 5.

tr_sendmix : at a given time, `tr_sendmix` mixes the frequencies associated to all the processors that have sent one or more messages, but whose messages have not all been received yet.

This tool draws the user's attention towards the pending messages in the machine: the more messages there are, the louder the sound is. It gives not only a good idea of the phases, by

[15] Remember that the low frequencies are quite close on a logarithmic scale.

[16] `SAMPLE_FREQ = 8,000` samples/s on a Sun's SPARCstation.

[17] Note that the output of cat is used as the input stream of our program. Any stream of traces could be used, and there is therefore no limitation to the size of the trace file studied. This allows us to avoid storage problems, and makes our approach scalable.

[18] `play` is the standard on-line sound playing program supplied with Sun's SPARCstations (usually located in the `/usr/demo/SOUND` directory)

pointing out their starting time (like tr_send), but also an idea of their duration. This makes it easy to find out when the communication bottlenecks take place, even if they don't originate in the same phase, because the corresponding sounds are sustained until the reception of the messages. The mapping of the frequencies may also be changed, to emphasize groups of processors.

tr_sendnum : the pitch of the sound generated by `tr_sendnum` at a given time is proportional to the number of messages sent by all the processors, but not received yet.

This tool, based on tr_sendmix, gives a simple and efficient way to determine the communication bottlenecks of an execution. The user does not need to focus his attention on the sound produced by the tool. The high pitch corresponding to (too) many pending messages can be used as an attention catching signal. The user can then use more precise tools to study in details what happened during the execution.

These three programs complement each other. Using them, you can easily determine when the communications take place. It is also easy to hear several processors sending data on a regular basis, and others being *out of phase*. By listening carefully to the rhythm, you can also determine if the programs go regularly through the same communication patterns. Unfortunately, the programs have not yet been used extensively enough, and by enough different users to tell more about the help they can bring to the user.

7 Conclusion

This paper has shown how easy it is to use sound on a Sun's SPARCstation with our approach, and how sound can be used to convey data. The trouble is that it is still quite hard to tell whether using sound in a monitoring program will help the user or not, outside of the research community. We believe however that the use of sound will prove increasingly useful, as the users get more and more processors to work together, and have therefore more trouble understanding the behavior of their programs. We hope that the availability of our programs, and their ease of use will help more users to use sound regularly, or at least give it a try. We will then be able to get enough feed-back about what the users think about sound, and how it could be used to meet their needs in the best possible way.

In the future, we will try to add our sound procedures to existing *mute* programs[19], such as HeNCE [BDG+92] or ParaGraph [HE91]. We will also add a graphical user interface to our sound programs and start using them on top of PIMSY [PTV92, vRTV92], our scalable monitoring system. We also plan to study the sound capabilities of workstations other than Sun's SPARCstations, and have our programs run on these stations as well.

8 Obtaining the sound tools

The tools discussed in this paper, and some example trace and sound files, are available from `netlib`. For more information about this package, send the following message to `netlib@ornl.gov`:

`send index from trace2au`

This research report is available by anonymous ftp in:

[19] We will have to deal with two problems: identifying where to add the calls to the sound procedures, and synchronizing the sounds with the data displayed on the screen.

- lip.ens-lyon.fr, in /pub/LIP/RR/RR93/RR93-24.ps.Z
- netlib2.cs.utk.edu, in tennessee/ut-cs-93-208.ps

References

[Ass88] MIDI Manufacturers Association. *MIDI - Musical Instrument Digital Interface, Specification 1.0.* International MIDI Association, Los Angeles, CA, 1988.

[BDG+92] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, K. Moore, and V. Sunderman. PVM and HeNCE : Tools for heterogeneous network computing. In J. Dongarra and B. Tourancheau, editors, *Environments and tools for parallel scientific Computing*, volume 6 of *Advances in parallel computing*, pages 139–154, Saint Hilaire du Touvet - France, September 1992. CNRS - NSF, Elsevier Sciences Publisher.

[BH92] Marc H. Brown and John Hershberger. Color and sound in algorithm animation. *Computer*, December 1992.

[Env92] Volvox Machines Programming Environment. *VolTms User's Guide*. ARCHIPEL SA, 74940 Annecy-le-vieux, France, 1992.

[FJA91] Joan M. Francioni, Jay Alan Jackson, and Larry Albright. The sounds of parallel programs. In IEEE Computer Society Press, editor, *The Sixth Distributed Memory Computing Conference Proceedings*, 1991.

[HE91] M. Heath and J. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8:29–39, September 1991.

[Hor85] Bill Horne. The sound of music. *Computing Today*, April 1985.

[Mad92] Tara Maja Madhyastha. A portable system for data sonification. Master's thesis, University of Illinois at Urbana-Champaign, 1992.

[PTV92] S. Poinson, B. Tourancheau, and X. Vigouroux. Distributed monitoring for scalable massively parallel machines. In J. Dongarra and B. Tourancheau, editors, *Environment and Tools for Parallel Scientific Computing*, volume 6 of *Advances in parallel computing*, pages 85–101, Saint Hilaire du Touvet - France, September 1992. CNRS - NSF, Elsevier Sciences Publisher.

[RAM+92] Daniel A. Reed, Ruth A. Aydt, Tara M. Madhyastha, Roger J Noe, Keith A. Shields, and Schwartz Bradley W. An overview of the pablo performance analysis environment. Technical report, University of Illinois at Urbana-Champaign, November 1992.

[Rem86] Claire Remy. Le compositeur et l'ordinateur. *Micro-Systemes*, June 1986.

[Sun91] Sun Microsystems. *SPARCstation Audio Programming*, July 1991. Part No : FE318-0.

[Sun92a] Sun Microsystems. *Multimedia Primer*, February 1992. Part No : FE328-0.

[Sun92b] Sun Microsystems. *SPARCstation 10 System Architecture*, May 1992. Part No : 4/92 FE-0/30K.

[VR93] Guido Van Rossum. Faq: Audio file formats. Usenet News, May 1993.

[vRT92] M. van Riek and B. Tourancheau. The design of the general parallel monitoring system. In N. Topham, R. Ibbett, and T. Bemmerl, editors, *Programming Environments for Parallel Computing*, volume A-11 of *IFIP*, pages 127–137, Edinburgh, Scotland, April 1992. IFIP, North-Holland.

[vRTV92] M. van Riek, B. Tourancheau, and X. Vigouroux. The massively parallel monitoring system (a truly scalable approach to parallel monitoring). In G. Haring, editor, *Performance Measurement and Visualization of Parallel Systems*, Moravany, CZ, October 1992. Elsevier Sciences Publisher.

[ZT92] Eugenio Zabala and Richard Taylor. Process and processor interaction : Architecture independent visualization schema. In J. J. Dongarra and B. Tourancheau, editors, *Environments and Tools for Parallel Scientific Computing*, volume 6 of *Advances in parallel computing*, Saint Hilaire du Touvet - France, September 1992. CNRS - NSF, Elsevier Sciences Publisher.