JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **18**, 181–194 (1993)

Breaking the Silence: Auralization of Parallel Program Behavior*

JOAN M. FRANCIONI AND JAY ALAN JACKSON

Computer Science Department, University of Southwestern Louisiana, Lafayette, Louisiana 70504

The run-time behavior of a parallel program is defined by many parameters, for example, the program's communication structure, the processor utilization profile, and the dynamic size of message queues. To understand the execution of a program, it is frequently necessary for a programmer to consider the run-time behavior from a number of different angles. Also, it is sometimes useful to consider one aspect of the behavior in isolation, whereas other times it is necessary to consider different types of behavior together. Sound offers an alternative form of investigation to simply using multiple graphical and textual views for studying the behavior of a program. In this paper, we discuss the properties of parallel programs that are well suited to being mapped to sound and present a number of example mappings. As evidence of the effectiveness of the sound mappings, we present case studies based on a prototype sound tool. In general, sound was found to be effective in depicting certain patterns and timing information related to the programs' behaviors. Also, by listening to sound representations that were sychronized with graphical displays, the speed of recognition and distinction of programs, and parts of programs, was increased. © 1993 Academic Press, Inc.

1. INTRODUCTION

From the early years of computing, there are examples of sound being used to depict the behavior of a program. Fernando Jose Corbató, winner of the 1991 ACM Turing Award, described a feature of the Whirlwind computer (circa mid-1950s) as follows:

> You even had audio output in the sense that you could hear the program because there was an audio amplifier on one of the bits of one of the registers—so each program had a signature. You could hear the tempo of how your program was running. You could sense when it was not running well, or when it was doing something surprising [6].

Later on, people discovered that tuning an AM radio to certain frequencies and placing it close to the computer allowed them to hear how their programs were doing. They were able to detect such things as when a frequently used program was being run and when a specific program was in an infinite loop or was performing poorly in general. (And, of course, programmers spent hours writing programs to *play* particular songs.) Another advantage of this kind of audio output was that it could be set up to be monitored from outside of the relatively cold and noisy machine room.

Programmers in those days made effective use of two basic capabilities of human hearing: (1) detecting aural patterns and anomalies within those patterns; and (2) processing sounds on a passive level, at the same time as doing something else. In addition, the sound was being used to *understand* certain aspects about the run-time behavior of sequential programs. Is it possible, then, that sound can be used effectively to represent the run-time behavior of parallel programs?

It is well known that understanding the behavior of parallel programs is much harder than understanding the behavior of sequential programs. One cannot just study the source code of a parallel program to determine the order in which statements will be executed, as can be done with sequential programs. During debugging and performance tuning of a parallel program, a programmer tries to gain insight into, and develop an intuition about, the concurrent events of the program and their relation to each other. Usually the programmer must consider the run-time events from different perspectives in order to obtain a full appreciation of the program's behavior.

Traditional techniques for representing run-time behavior include tables of statistics, profile information, call-graphs, and graphics of run-time events over time. The latter technique includes both static displays, where time is represented as part of the graph, and dynamic displays, where the actual display changes over time (i.e., an animation). All of these techniques have one thing in common: they are processed via our visual sense. It is possible, however, that other senses can also be used to interpret program behavior data. In this paper, we explore the effectiveness of using sound to represent different aspects of the run-time behavior of parallel programs in order to assist a programmer in understanding how a program is actually working. We define this representation of program data as the *auralization* of a parallel program.

A major reason for considering auralization of parallel programs is to employ the natural ability of sound to depict parallel events in a manner that reveals their tem-

* This work supported by the National Science Foundation under Grant No. CCR-9196108.

poral ordering and duration. In Section 3, we show how a number of program behavioral characteristics can be mapped to properties of sound. In order to study the effectiveness of such mappings related to actual parallel programs, we have developed a prototype auralization system. The system is described in Section 4, and evidence of the effectiveness of auralizations, in general, is presented in Section 5.

2. RELATED WORK

Previous work related to using sound effects and music in order to aid in the comprehension of complex applications has been done in a variety of ways. In the early 1980s, a number of studies were performed which determined that sound could be used effectively in representing multivariate data. The data were being used for such tasks as classifying mineral samples [21], recognizing organic compounds [10], and determining positive correlations among sets of statistical variables [1, 12]. In each of these cases, aural representation of the data was found to significantly aid in the recognition of the desired phenomenon.

Smith *et al.* [17] have also investigated representing multidimensional data via sound. In particular, their approach is designed to aid in the perception of structure in data by associating auditory attributes to visual icons. Each icon's attributes, such as size, position of its parts, pitch, and loudness, are data-driven. The visual attributes are displayed on the screen; the aural attributes are sounded as a mouse-driven cursor is moved over the graphic display. The result is intended to give the user an impression of the overall texture of the iconographic display and thus some insight into the structure of the data.

The use of data-driven sound as an enhancement to scientific animations has recently been investigated by Scaletti and Craig [16]. In their study, four scientific animation videos were "sonified" based on the initial data used to generate the graphics. Although this work deals with aural and visual depictions of the output of programs versus the behavior of the program itself, a number of their conclusions directly support the claims of this paper. For example, based on an informal survey of people who saw both the original and the sound-enhanced videos, they observed that

• users felt it was harder to pick out and remember patterns in the silent animations,

• the most effective mappings were achieved when the sound was coherent with the picture,

• complex timbres were easier to locate in space and were less tiring to the listener than were simple sine waves; i.e., the aesthetics of the sounds used were significant, and

• after hearing the data-driven sound track, users still imagined it when the video was played silently.

As a slightly different example, Fiorella Terenzi, an astrophysics professor at Liceo Scientifico in Milan, Italy, is using the computer to transform radio emissions from outer space into audible form to help her study the nature and composition of galaxy UGC 6697. All of these studies taken together are strong evidence that representing multi-dimensional data with sound can provide insight into the behavior of a system. In addition, it has been demonstrated that presenting multi-dimensional data using sound and graphics together increases the probability that more is understood about the data in a shorter amount of time than if either type of presentation is used alone [1, 12].

Based on a completely different type of human-computer interface, Wenzel *et al.* [19] describe the real-time acoustic display capabilities for the virtual environment project at NASA-Ames. Their system is designed to provide auditory cues related to what the user is seeing in order to help guide movement through the virtual reality. Obviously sound is necessary for any *realistic* virtual environment.

There have been few studies demonstrating the feasibility of mapping parallel program performance data to sound. One approach was studied by Sonnenwald *et al.* in their design of InfoSound [18]. The system allowed developers to create and store musical sequences and special sound effects, and then to associate those sounds to an application program's events. The system was tested on two programs—a telephone network service simulation and a parallel computation simulation—and in both cases was found to help users detect rapid, multiple event sequences that were difficult to visually detect using text and graphical interfaces. InfoSound was not developed as a general purpose tool and thus the sound mappings had to be customized for each program. In addition, the project was discontinued after only these two tests. Nonetheless, it did provide a mapping of parallel computation events to sound, and it did demonstrate that comprehension of the run-time behavior of the program was increased.

The feasibility of mapping actual parallel program performance data to sound was first demonstrated by two separate reports at the same conference: Francioni *et al.* [4] and Reed *et al.* [15]. Reference [4] demonstrated the potential of sound to portray performance data via three separate sound-mappings depicting communication events, processor utilization, and flow-of-control. Reference [15] presented an auditory display that was synchronized with a corresponding graphics display. In both cases, the sounds were directly mapped to run-time events of parallel programs, and the mappings were designed as general purpose tools.

3. MAPPING PARALLEL PROGRAM BEHAVIOR TO SOUND

There are a number of general characteristics and properties related to the run-time behavior of a parallel program that can be defined. Some of these characteristics lend themselves well to visual displays, some to textual representations, and some to auralizations. Our intention is to identify which ones can be effectively represented in the aural domain.

3.1. Characteristics of Parallel Program Behavior

A set of characteristics that can be used to describe much, if not all, of the run-time behavior of parallel programs is defined as follows:

(1) *Relative timing* of events, i.e., when one event occurs in relation to another. Examples of events where timing information is useful include message sends and receives, beginnings and ends of processor busy/idle times, attainment of peak values for some measure, and execution of a marked section of code.

(2) *Duration* of events. As opposed to the timing among events, this characteristic deals with how long an event, or an interval between two events, lasts.

(3) *Patterns* of events. Patterns are found in event occurrences across processors as well as within individual processors. They are recognized based on some predefined structure of the computational model, or because of some repeated or periodic behavior.

(4) *Phases* of behavior over time. Although a particular part of a program may not generate a recognizable pattern of events, it is still possible that phases of a program can be discerned within distinct periods of time.

(5) *Frequency* over time of some behavior. This type of measure is useful in capturing information such as processor utilization, size of message queues, and number of cache hits.

(6) *Balance* of behavior over entire system. This characteristic would be used to describe the distribution of some behavioral aspect among different parts of the system. Examples would include the load balance of the processors' utilizations, and the balance of homogeneous versus heterogeneous communication activity among different processes.

(7) *Specifics* of an event. This includes the detailed information typically stored in the trace-file of a particular run. For instance, the specifics of a message communication would include the sender, receiver, a message id, the length of the message, and the time of the send or receipt.

The reasons for presenting performance information to a programmer are to help the programmer understand what the program is doing and to relate the run-time behavior back to program source code. Given that the above characteristics define the run-time behavior of a parallel program, the relevant question becomes "How can sound effectively represent these characteristics?"

Studies on human perception of sound have shown that the human ear is capable of differentiating among many different sounds [2, 14]. By mapping execution events to sounds, we are afforded the flexibility of using many dimensions of sound and having a wide range of possible values within each dimension. The dimensions of sound that can be used include the timbre of the sound, which can be thought of as the voice of the instrument that generates the sound; the pitch of the sound; the duration of the sound; the intensity of the sound; and the spatial location of the sound source in the stereo field. In addition to these specifics, sound compositions are characterized by patterns of rhythm and meter, melody, harmony, and texture. All of these properties can be used, in varying combinations, to depict characteristic information related to a program's execution behavior.

In answer to the question posed above, then, it is necessary to map specific properties of sound to the characteristics that were enumerated. We present arguments for a number of such mappings in this section and we describe specific examples in Section 5.

3.2. Relation of Program Characteristics to Sound

It is an inherent property of sound that both *relative timings* and absolute *durations* of sounds are implicit in any song or playback. (The word *song* implies a melody, thus we use the term *playback* to refer to a generic sequence of sounds.) When sounds are associated with each of a certain kind of parallel program event, the temporal order of the program's events is implicit in the rhythm and meter of the corresponding auralization. In the same way, the duration of events would be defined implicitly by the duration of their corresponding sounds in an auralization. It has been demonstrated that our ears are sensitive enough to detect time differences of a few milliseconds [14]. This implies that temporal information can be processed by our ears. In addition to the temporal ordering of sounds, more than one sound can be detected by humans at a time. Thus, we have a natural ability to hear parallel events.

As evidenced by our recognition of familiar songs and choruses within a new song, we know that humans have the ability to detect *patterns* in sound. How quickly a pattern is detected is a function of the scale of the piece, the tempo of the piece, the length of the patterns, and the frequency of repetition, among other things. The parameters of a pattern related to parallel program events are based on the actual program's run-time behavior as well as the specific way in which sounds are mapped to each event. For example, the choice of scale used in a mapping of notes to processor-sends may affect the likelihood of recognizing a pattern. In general, however, patterns in the program behavior that can be reflected by the rhythm and/or melody of an auralization are more likely to be aurally detected. *Phases* of a program are not always as well defined in the aural playback of a program. In order to detect a phase aurally, there must be an aural cue such as an obvious break in the sound, a change in the tempo, or even a change in the volume. A phase may also be detected by the return of some recognizable pattern.

In order to map the *frequency* characteristic described above to sound, it is necessary to use a property of sound that naturally depicts quantity. One such property that can be used is the pitch of a sound, i.e., the note. Other properties include the intensity of the sound and the density of notes being heard at a time. Each of these aspects of sound naturally depict increases and decreases of an associated quantity and, thus, they can represent frequency on a relative scale. They do not, however, reflect values on an absolute scale very well. For example, one can hear if the pitch of a sound is getting higher or lower even if the change is very small, but not many people have "perfect pitch" to be able to exactly identify a particular note being heard.

Balance within a parallel program can be mapped to sound in a number of ways. Among simultaneously sounding voices, humans can detect balance in the volume as well as in the rhythmic or melodic activity of the sounds. Humans can also detect the spatial source of a sound from any direction with high acuity [2]. Moreover, it is possible to simulate 3D sound using stereo headphones [19, 20]. Thus, the arrangement in space of sound sources is another way of aurally depicting balance.

The ability of sound to represent *specific* event information depends on the kind of specifics intended. Basically any event can be mapped to sound where an event is either defined in, or computed from, information in a trace file. (It is irrelevant to this discussion whether the trace file is processed postmortem or in real-time.) This includes control events, state transitions, resource and data accesses, and communication events. When there are many events being represented simultaneously, the overall sound of the auralization will tend to dominate the sounds of individual events. For instance, if an auralization is depicting the flow-of-control behavior of a 1000 processors, the behavior of one specific processor will probably not be aurally evident. Conversely, if the sounds relevant to a small number of processors are played at a slow tempo, detailed information can be deduced from a playback.

3.3. Complementing Visual Representation

Visual displays can be grouped into one of two categories: static or dynamic. In a static display, previous history is available but temporal information must be abstracted from the graph. Conversely, in a dynamic display, the temporal information is intrinsic in the animation, whereas the history of past events is available only as that which can be remembered. An aural depiction of program events has similar properties to an animation, although different kinds of patterns can be remembered and recognized in one compared to the other. The best situation for the programmer is to be able to perceive accurate temporal information that is associated with a historical depiction. In that way, the historical depiction can provide cues to help the programmer remember the temporal relationships between events more accurately. This will only work, however, if the programmer can exactly correlate the temporal depiction with the historical depiction. Since people cannot watch two different areas of a screen at the same time, this is very hard to do with only static and animation visual displays. However, people can look and listen at the same time. Correlating an auralization with a visual static display provides a mechanism for accomplishing the above-stated goal.

3.4. Scalability of Sound Mappings

When considering the properties of sound that will map to the characteristics of parallel program behavior, it is important to consider the mappings from the extended point of view of large-scale parallel systems. With respect to scalability, sound has definite limitations in terms of how much sound can be generated and/or assimilated at once. If program events are mapped to the notes of a major scale, for instance, there is a limit of 128 unique notes that can be played within the range of human hearing. Using different scales, this limit can be increased somewhat but certainly not to 10,000. On the other hand, sound does offer different possibilities than graphics for dealing with large numbers of parallel events. Examples in nature of auralizations of large numbers of parallel events include rain drops falling in a storm, leaves blowing in the wind, and bees buzzing at a hive. In each of these cases, we can aurally detect information about a large system based on the overall sound.

Consider the method of dividing a large number of processors up into a smaller number of logical groupings or equivalence classes of the processors, and then considering the interactions of the groups as a whole. There are a number of situations in which our aural processing abilities are exceptional at detecting equivalence classes. For example, our ears can detect that the same note is played by two distinct instruments even when the wave forms of their sounds are entirely different. In this case our ear recognizes that the two notes have the same fundamental frequencies. Another property of sound that lends itself to large-scale processing is that rhythmic patterns are invariant of tempo. Hence, it is possible to aurally emphasize the sound corresponding to a few processors and then play the auralization related to the entire program at a relatively fast tempo. Any rhythmic pattern involving the few processors would remain intact in the rapid playback. In this way, sound can be used to draw the listeners attention to specific behavior at the same time that the overall behavior is being considered.

3.5. Limitations of Sound Mappings

Sound is not always an effective medium for representing certain kinds of detailed information that are useful to a programmer. For example, there is no naturally occurring aural relation between specific notes or instruments and specific processors. Without any formal ear training, it would be difficult to remember which processor went with which note or which instrument for more than two or three processors. It is much easier to look at a graph or table to determine the processor that sent a specific message. On the other hand, one can listen for a certain note or instrument to be played and then recognize when that particular processor has generated an event.

Depicting certain maximum and minimum values of a quantity is difficult to do with sound. For example, what is the maximum loudness or the lowest note or the fastest tempo? Including a reference sound that can be heard in the background is one way of handling this, but a continuously sustained note tends to be annoying. The point is that this is not a natural property of sound that can be put to good use.

There is no guarantee that an auralization will always be acceptable sounding. In some cases an auralization may result in bothersome noise; in the worst case the resulting sound may actually offend the listener (e.g., if the sound is very loud or if it has an extremely high pitch). In general, the character of an auralization will be determined by (1) the definition and timing of the program events being depicted, and (2) the voice and scale assignments of the events to sounds. Unless one is writing programs solely to generate pleasing sounds, the first property cannot be varied to change the sound of an auralization. But the second property should be varied. Experimentation with scale choices and voice assignments (such as a choice of instrument or sound sample, short vs sustained sound, sharp vs gradual attack, etc.) is typically necessary to find meaningful auralizations. Therefore, the potential for creating disturbing auralizations also exists. As more research into this topic is carried out, however, it should be possible to determine a set of low-risk default choices and assignment guidelines that would reduce the possibility of bad auralizations.

4. PROTOTYPE OF AN AURALIZATION TOOL

In order to test the validity of the claims made in the previous sections, an auralization tool prototype has been developed. The tool was designed to present auditory representations of performance monitoring information that are synchronized with matching visual displays. The sounds of the auralizations are based on one of several predefined event-to-sound mappings, where program-related events are derived from a trace file generated during program execution.

4.1. System Organization

The organization of our system for generating an audio display of a program's execution behavior consists of three basic tasks: (1) collection of event trace data, (2) mapping of events to MIDI (musical instrument digital interface) format, and (3) presentation of the synchronized aural/visual display. The collection of the event trace data is facilitated by using the PICL subroutine library [7], which has been implemented for a number of distributed-memory machines. PICL provides execution tracing information on events such as basic sends and receives, high-level communication operations, idle/busy time, and user-defined events.

After the trace data has been collected during a program's execution, the data is then *mapped* to specific sounds according to the MIDI protocol [13]. This protocol defines a standard format for representing the components of sound, such as duration, pitch, instrument, volume, and panning. The mappings are done by a set of C programs that are run postmortem on the trace data. In some cases individual PICL-defined events are mapped directly to sound; in other cases the sound mapping is derived from multiple PICL events. An example of a direct event-to-sound mapping would be as follows:

PICL event:
send clock 0 523 node 0 to 1 type 4 lth 4
MIDI event(s):
playnote, time=523, note=C, channel=1, volume=90
playnote, time=533, note=C, channel=1, volume=0
(The second MIDI event turns the previous note off after 10 time units.)

The file of MIDI data conforms to Format 0 for MIDI sequencers. This makes it possible for the file to be accepted by any commercial MIDI device which is capable of reading the standard format. Therefore, converting the data to sound is done simply by sending the data to a MIDI device such as a synthesizer or tone generator.

During presentation, the aural data is synchronized with one of two ParaGraph visual displays [9]: the space-time graph (e.g., Fig. 1) or the Gantt chart (e.g., Fig. 3). Both of these displays are static displays in the sense that time is depicted along the horizontal axis. (ParaGraph provides many other displays than are represented here, some of which are dynamic.) After the display has been drawn, the user defines a region of the display for aural presentation. During playback, a vertical bar, the height of the display, moves across the window from left to right, and the sounds associated with the current position of the bar are generated. Thus, the user always knows what parts of the graph currently are being played.

The timing of aural events in a playback is exactly proportional to the timing of the actual trace events. Hence, all temporal relationships of the program's run-time behavior are preserved in the auralization. The tempo of the playback, however, is at the user's control. Thus the user can listen to a fast version of a playback to obtain an overall impression of the program's behavior as well as slow the tempo down to hear details, similar to zooming in on part of a visual display. The user also controls the channel-to-voice and channel-to-stereo field assignments. When the MIDI file is generated, each sound to be played is assigned to a specific channel. These channels can then be assigned to specific instruments and stereo fields based on the capabilities of the sound-generating device. So, for example, if all send events are assigned to channel 0, and all receive events are assigned to channel 1, then on playback, channel 0 can be assigned to the left stereo field and channel 1 to the right. In addition, the send events can be played by one instrument, while the receive events are played by another. Both channels can also be assigned to the same stereo field and/or instrument. This flexibility in channel assignment is important when experimenting to determine the most effective auralization for a program. Other controls at the user interface include changing the sound mapping, changing the volume, muting out specific channels, and setting the region of display for playback.

4.2. Sample Sound Mappings

In this section, we describe a sampling of sound mappings that have been implemented for experimentation.

Send–receive. The relative ordering and timing of all messages is one of the defining characteristics of a distributed-memory parallel program. Figures 1, 2, 4, and 6 show examples of space-time diagrams which depict the timing and processor information of messages in a distributed-memory parallel program. In each diagram, time is on the horizontal axis and processors are along the vertical axis. The left endpoint of a diagonal line in the graph represents a send by that processor; the right endpoint represents the receiving processor. A basic sound-mapping that captures this type of behavior is to assign a particular timbre (or instrument) to each kind of event—sends and receives in this case—and play a note for each send and receive event that occurs. The particular note that is chosen can be a function of event attributes such as sending or receiving processor number, message length, and message type. Depending on the specific parameters used in the mapping and depending on the speed chosen for the playback, different kinds of behavior can be depicted.

Simple variations of the above mapping result in auralizations that convey different kinds of information. When the note played on a send corresponds to the sending processor's number and the note played when a message is received corresponds to the receiving processor's number, it can be heard how long it takes a processor to send off a message after having received one by listening for the repeat of a given note. When the note played on a send is set up to correspond to the receiving processor's number rather than the sender's number, the duration of time between a send and its matching receive can be heard. Sustaining the sender's note until the receive occurs is another way of depicting this information that is also useful for detecting unreceived or delayed messages.

The basic send–receive mapping assigns a separate note to each processor. A potential problem of this mapping for large numbers of processors is that, for a given instrument, it is unpleasant to hear certain notes played at the same (or similar) time. One solution to this is to use voices whose sounds are not associated with definite musical scales. For instance, drum-like voices can depict relative pitch information without creating a lot of dissonance.

Group-send–receive. A variation of the basic send–receive mapping applicable to large number of processors is to use a grouping strategy for note assignment. (This is similar to a color-binning strategy for visual displays [3].) The processors are separated into a relatively small number of groups and individual notes are assigned to each group rather than each processor. In such a mapping, it is desirable to distinguish between intra- and inter-communication. One strategy is to direct the sound to one channel when a processor sends a message to another processor in its same group, and when a message transcends group boundaries, the sounds for both the send and receive are directed to a different channel. By using two distinct channels, it is possible to listen to the inter-group communication traffic in conjunction with, and separately from, the intra-group traffic. The capacity for scaling this group-send–receive mapping is basically unlimited as any number of processors can be assigned to one group. Also, a user can experiment with different processor-to-group assignments to study the behavior of a program in varying ways. Possibilities would include dividing the processors simply into two groups to find out which group is doing the most communication, or grouping the processors working on boundary data separately from those processors working on the internal regions.

Idle–busy. Another characterization of the performance of a parallel program is the amount of time each processor spends computing as opposed to idle. The

Gantt chart of Fig. 3 depicts this with a dark bar representing when a processor is busy, and a white bar representing idle time. As in the previous figure, time is along the horizontal axis and processors are along the vertical axis. The ratio of idle-to-busy time can be depicted aurally by playing a certain sound for each processor whenever it is idle and playing no sound when the processor is busy (or vice versa). In either case, the sound should be sustained for the duration of the burst. This property dictates the kinds of voices that are suitable. An enhancement to this mapping is to also use the intensity dimension of sound to reflect the length of an idle burst. In the resulting auralization, the beginnings and endings of the idle periods should be evident, as well as the relative number of processors idle at a time. With the enhancement, attention will also be drawn to the longer idle bursts due to their higher volume.

Flow-of-control. This sound mapping is intended to depict the flow of control within each processor. The mapping is straightforward: each processor is assigned a particular timbre which may or may not be unique, and a note is played on each event being monitored. If the events being monitored are "mark" events of different locations in the code, the note played would be a function of which mark has occurred. If the events being monitored are communication events, the note played could be a function of the processor sending or receiving the message and/or the type of message. The difference in this case from the general send–receive mapping is that, presumably, only special communication events would be mapped to sound. Thus, a programmer could follow, for example, the progression of one processor's communication events or the flow of one message sequence.

Meters. It is often useful in performance monitoring to keep track of certain cumulative statistics over time, expressed as percentages. Visually, this can be depicted with meter types of displays where the height of a bar fluctuates dynamically corresponding to the current statistical value. (ParaGraph contains such displays for communication and utilization statistics.) In auralizations, percentages can easily be represented by selecting notes in a range of pitches to represent corresponding values from 0 to 100%—low notes corresponding to low percentages; high notes to high percentages. This mapping can be designed to depict, for example, the relative amount of system utilization or communication that is occurring over time. Since the range of values to be sounded is restricted to be from 0 to 100%, this mapping is independent of the actual number of processors or events in the system. Hence, it is completely scalable. In addition, sound meters correspond well with visual displays, where the sound represents the aggregate statistic and the graphics represent specific values in detail. Such an aural–visual combination serves to save screen space and also relieves the user of simultaneously following two or more graphical displays.

5. EVIDENCE OF AURALIZATION EFFECTIVENESS

The evidence presented in this section is based on reactions of the authors and others who have listened to various examples of program auralizations and on a small survey taken as part of a research demonstration at the "Supercomputing '91" conference. In general, the more one listens to a particular auralization, the more information one can detect and the quicker one can understand a corresponding graphic display. But even based on a single playback, people typically say they hear more than they expected they would, especially those who lack musical training.

5.1. Test Cases

Each test case in this section is discussed in terms of a specific ParaGraph display generated from actual program trace files. The programs were run on either an Intel iPSC/2 or an nCUBE machine, of up to 64 nodes.

Figure 1 represents the communications in a Fast Fourier Transform program. As one test case, an auralization of the program was generated using the *send–receive* mapping. All send events were mapped to one instrument, namely vibes, and all receives were mapped to another, namely marimba. Also, sends and receives were each directed to a separate stereo field. On each event, the note played was based on the processor that initiated the send or receive. As can be seen in the figure, this program has four distinct phases in which every processor sends a message and then every processor receives a message. In addition, the butterfly communication pattern is evident. The aural playback, on the other hand, provided for some different insights. When the auralization is played at a relatively fast tempo, it sounds like all processors do a send at the same time, and that they all do their corresponding receives together as well. When the auralization is slowed down, however, it is heard that the sends become more and more spread out at each subsequent phase and that the order of the receives is an exact match. In other words, a processor that is later than some others to send off its message will be later in receiving its message of the same phase. This is not immediately obvious from looking at even a scaled down version of the graph.

Figure 2 indicates the communication behavior of a matrix system solver run on 16 nodes. The auralization used in this test was a variation of the send–receive mapping such that only send events were played. Based on a fast-tempo playback, the communication activity in region 1 of the graph seemed to involve all the processors.

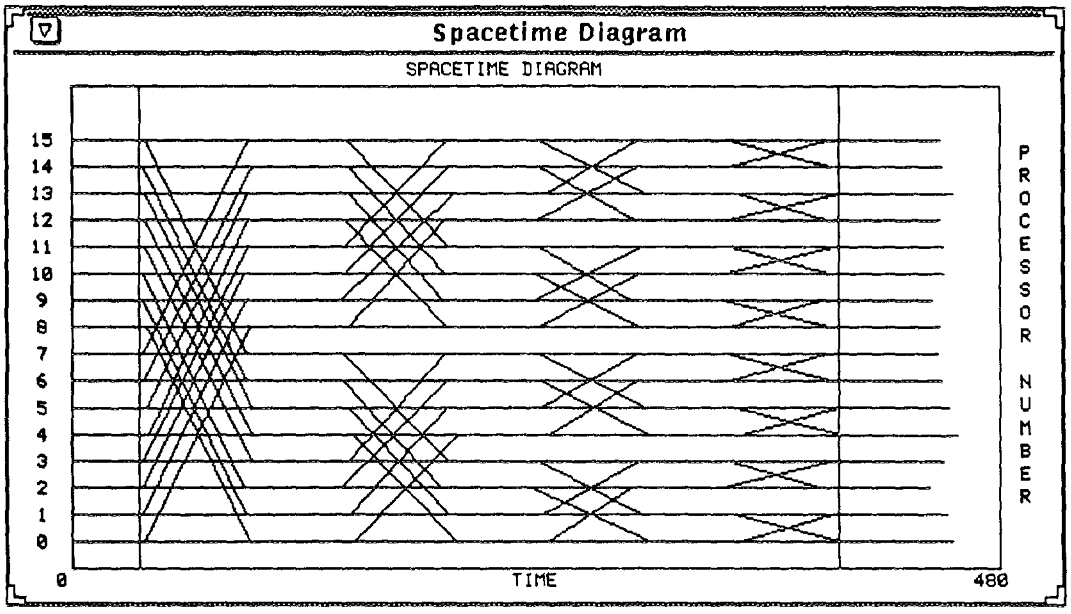

FIG. 1. FFT Program.

A slower playback of the region further indicated a repetition of seven sequences of sends, seemingly involving all 16 processors. Muting out the channels associated with the lower numbered processors (0–7), the seven groups of sends could still be heard. But when the upper processors were muted out, it was heard that the lower numbered processors were, in fact, sending much fewer messages in a very different pattern than that of the upper group. Now, this same investigation could just as well have been carried out using a visual display that had the capability of selectively displaying processors. Unfortunately, the version of ParaGraph we were using did not have such a capability. And although very close scrutiny of a scaled down version of the space-time graph confirmed that the above was indeed true, the behavior was not at all evident when the graph was first studied. In this

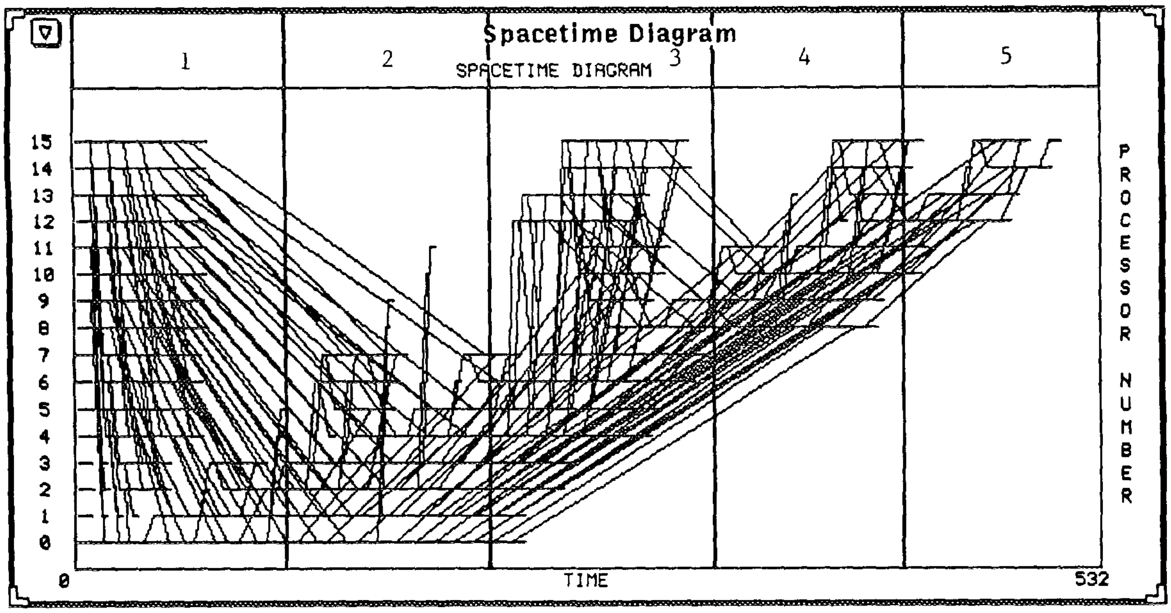

FIG. 2. Matrix computation.

case, the aural information served as the clue to the behavior.

The program of Figure 3 computed a matrix factorization. The auralization generated for this case was based on the *idle–busy* mapping. Specifically, the beginning of each idle burst is signified with a short, bell-like sound played at a note corresponding to a particular processor. This is followed immediately by a sustained string sound, played at the same note and starting at a low volume. The loudness of the string sound then increases as the length of the corresponding idle burst increases. In looking at the graph, one's eyes have a tendency to follow the diagonal bands of either white or dark. This makes it difficult to perceive the relative number of processors that are either idle or busy at different points in time. The auralization forces the programmer to process the graph basically one vertical strip at a time. For example, there are only a few short intervals where all processors are busy. These intervals are aurally recognized as the only completely quiet times, even though the duration of these quiet times is very short.

The flow-of-control auralization generated for another test was intended to follow a particular cause-and-effect sequence of messages: given a starting send, where does that message go and which is the next message sent by the receiving processor, and then where does this next message go, and so on. We term such a sequence of messages to be a *grapevine*. A visual display of a grapevine can be generated by highlighting the appropriate edges of a space-time diagram. Figure 4 shows an example of a grapevine for a Cholesky factorization program using a ring communication topology. For the sound mapping, a note was played for each message in the grapevine corresponding to the sending processor. The note was started when the message was first sent, was sustained until the message was received, and then was ended with the note changing to that of the receiver. Sounds related to the other sending and receiving events of the program were played on different voices at a lower volume according to the basic send–receive mapping. This auralization gives a clear representation of the flow-of-control of a specific message sequence. Since the pitch of the note being sustained is changed from the sender to the receiver, the direction of each message is heard, making the ring communication pattern evident. Also, the relative duration of time between the receipt of a message and the corresponding send of the next message in the sequence is accurately perceived.

Figure 5 shows two views of the storer phase of the SLALOM benchmark program [8]. The Gantt chart shows the utilization of each processor individually, while the Count chart shows the aggregate processor utilization. A meter type of dynamic display for overall utilization basically follows the top of the curve of the Count chart. Coordinating a sound meter of the overall utilization with the visual Gantt chart allows one to process the

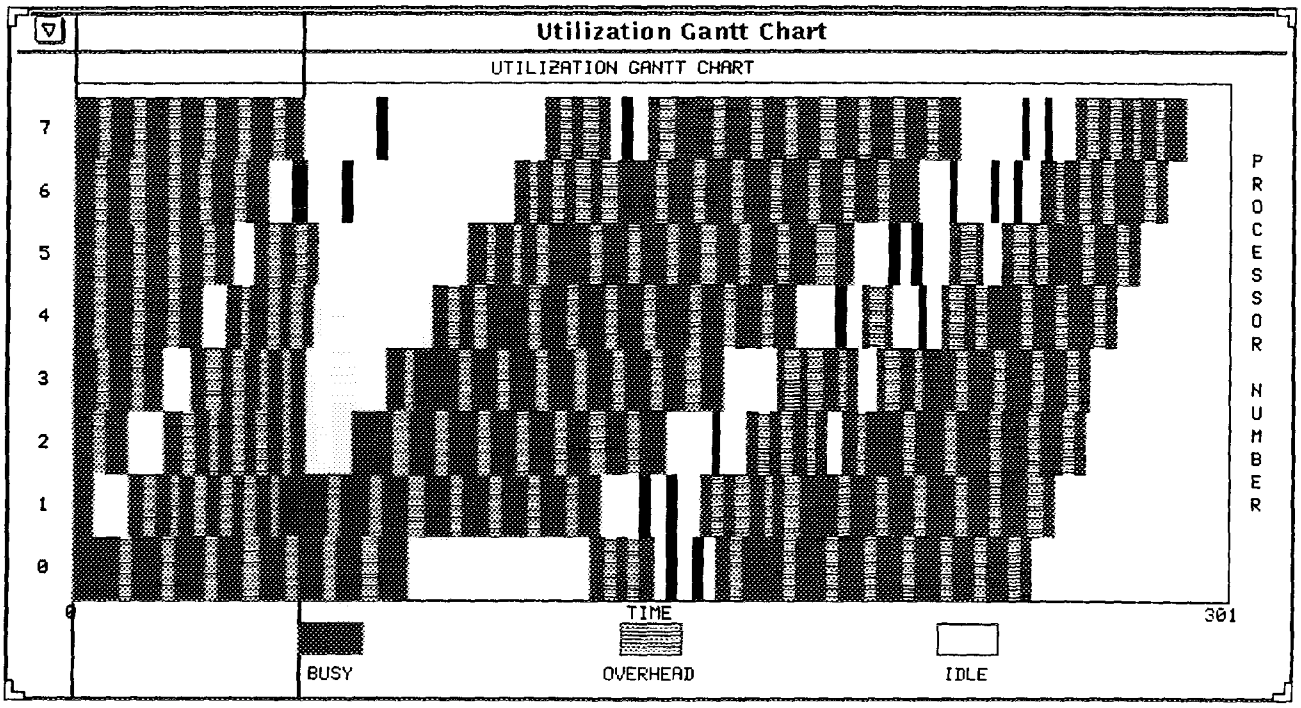

FIG. 3. Factorization program.

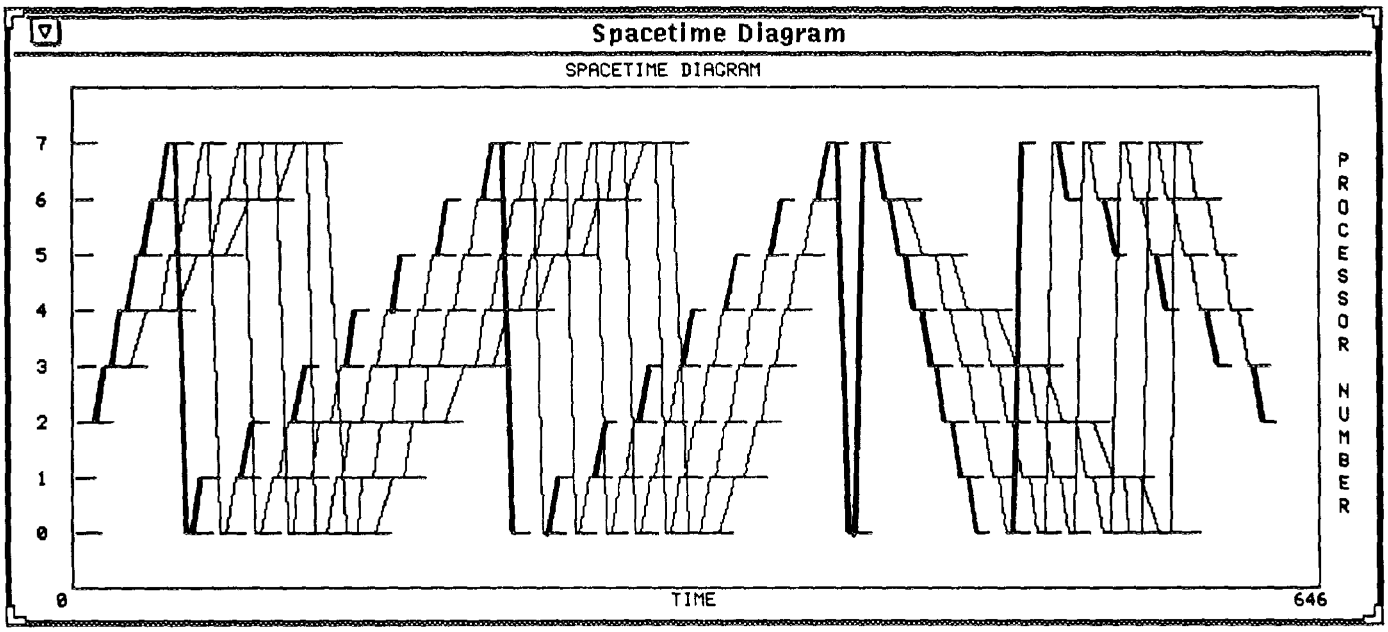

FIG. 4. Cholesky–ring program.

two types of information together. This is much harder to do visually as you try to look at two different displays at the same time. It is evident from the resulting auralization that the overall utilization is higher in the second active region, but also that the pattern of utilization is very similar in both active regions.

Figure 6 was generated from the backsolve phase of SLALOM. The decomposition topology for SLALOM is a two-dimensional mesh, so for 64 processors it is an 8 by 8 mesh. Each row of processors in the mesh is mapped to a row of the matrix. In the backsolve phase, activity "moves" row by row up the mesh and matrix. For this test case, processors were assigned to eight groups, each group included an entire row of processors, and the group-send–receive sound mapping was used [5]. During playback it was aurally detected that inter-group communication was always between the same neighbors (except at the end of the run). It could also be heard that most communication took place within a group and that inter-group communication took place at the end of a phase of intra-group communications. Finally, it was aurally evident that the same basic communication pattern is repeated over and over, and it is also repeated one last time in conjunction with the other "wrap-up" communication. By playing the auralization at a slow tempo, it is possible to hear that the basic communication of each group is the same.

One last example of the effectiveness of program auralizations is of a data recording error found in a trace file that multiple years of visualization had not revealed. The error was detected by Tara Madhyastha while experimenting with auralizations in conjunction with the Pablo performance analysis system [15]. A variation of the basic send–receive sound mapping was being used, whereby the sending note was sustained until the corresponding message was received. Upon playback of this particular trace file, it was evident that one of the messages was not being received, as a constant hum of one note was heard. Since it was not evident which processor was mapped to that particular note, the mapping was modified so that each processor was assigned to a different instrument. On the next playback, it was easy to identify the stuck note as that of the "tuba" processor and consequently the bug was easily traced [11].

5.2. *Survey*

The "Supercomputing '91" conference in Albuquerque provided an opportunity to test the effectiveness of sound-enhanced graphics on a random sampling of programmers. As one of the conference's research demonstrations, we ran a prototype version of our auralization system. Since this was a live demonstration, it was possible to vary parameters of the sound playbacks, such as instrument and tempo, and also to select which processors and which kinds of events to listen to. In general, almost everyone agreed that they were able to hear information that was different from what they could tell by just looking at the graphical displays. In some cases, the aural information simply confirmed something that appeared to be true based on the graph. In other cases, the aural playback portrayed information that was obscured

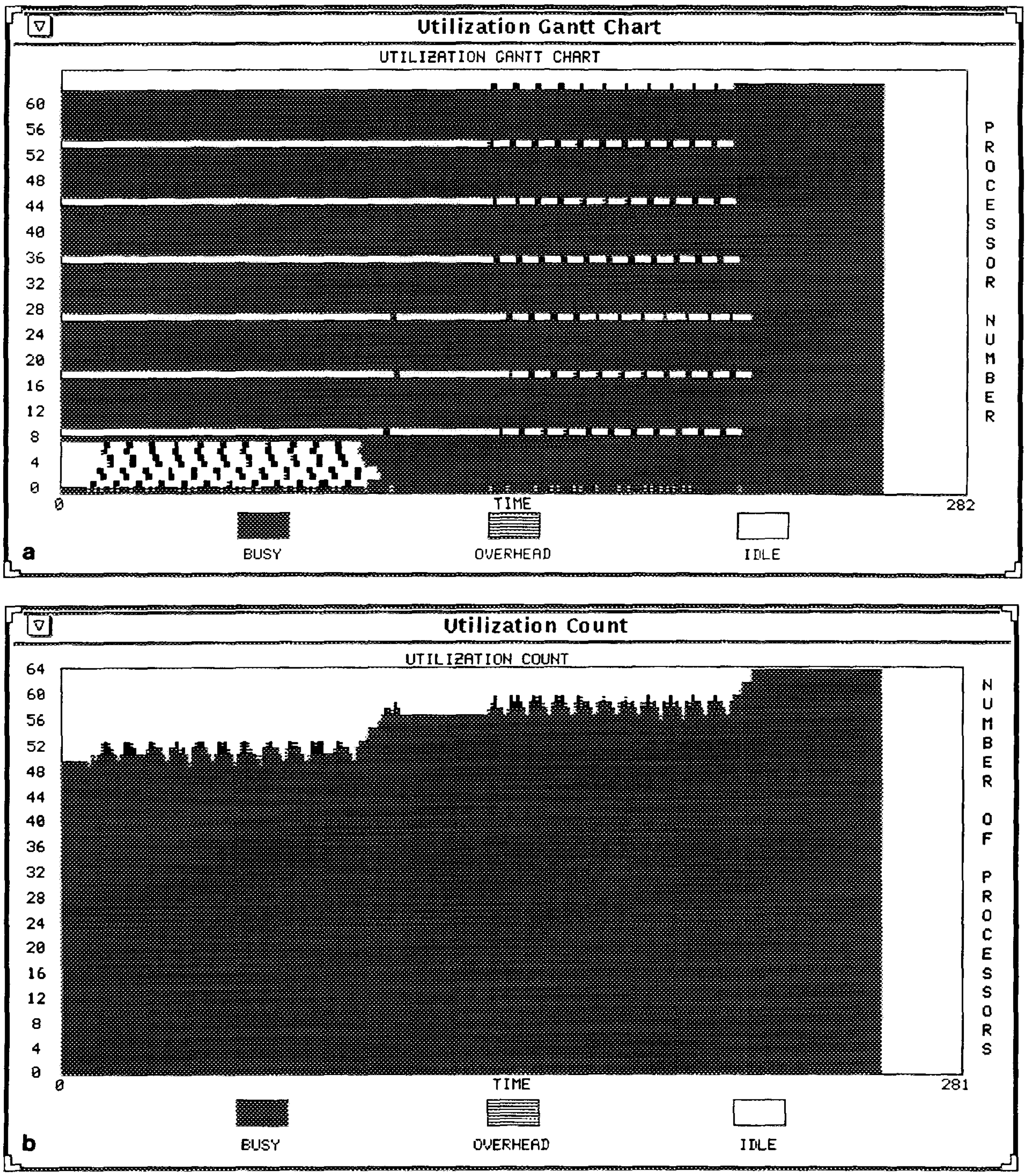

FIG. 5. Processor utilization.

in the graph. In addition to informally observing the reactions of users, a small survey was administered to a random sample of the users. The survey consisted of three questions, where people were asked to answer each question, first based on the graph alone and then based on watching and listening to a synchronized auralization of the graph.

The first question was related to the timing of phases in the FFT program of Fig. 1. Phases were defined to be regions of communication exchange and regions of strict computation involving no communication, either pending or active. The specific question asked was as follows:

(1) In the region indicated, do the lengths of the com-

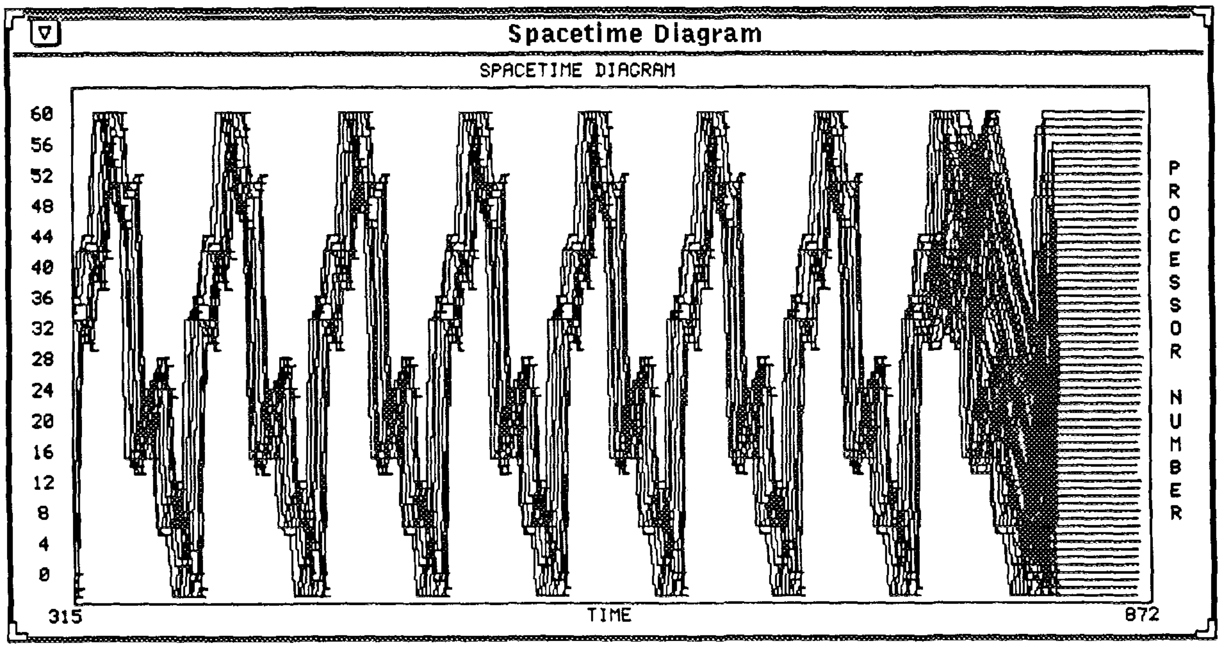

FIG. 6. SLALOM backsolve.

putation phases seem to be less than, equal to, or greater than the lengths of the communication phases?

Based on a send–receive mapping played at a moderately fast tempo, many people found it was easier to hear that the phases are virtually equal in duration than it was to see this. In part, this may be due to the visual differences in the diagonal lines versus the horizontal lines.

The second question was related to the density of "communication activity," defined to be the number of *sends* occurring during any interval. Participants were asked the following question regarding Fig. 2:

(2) Which region represents the period of highest communication activity?

The auralization used in this case generated one sound for each send event. In particular, a synthesizer sound of indefinite pitch, with a very short duration, was used. When played at a relatively fast tempo, this mapping sounded much like static or a Geiger counter. For this program, there are more send events during region 1 than in the other regions. But there are more pending events in region 3 and thus more "black" in the graph. Without the auralization, most people thought region 3 had the highest level of sending events.

The third question dealt with the relative timing of a particular event: the beginning of an idle burst. The graph shown was that of Fig. 3 and the question was as follows:

(3) In the region indicated, do the lengths of time between the starts of each idle period seem to be the same or different?

In this program, the time between the start of each of the first six idle bursts is highly uniform; each one begins within 5% of the average interval. However, the lengths of the idle bursts vary much more—for processor 1 the burst lasts until processor 2's burst begins; for processor 5 the burst is basically half as long. Because of this, it is hard to visually judge when each burst begins even though it is easy to hear the consistent rhythm of the notes. For the auralization, the idle–busy sound mapping was used.

The results of the survey are given in Table I. In each case, a significant number of people changed their answer after hearing the auralization. The most striking point about these results, however, is that people were able to perceive the correct answer 100% of the time based on the sound-enhanced graphics. In addition, many people

TABLE I
Results (in Percentages) of SC'91 Survey[a]

Question	Graph only		Sound and graph		Changed answer
(1)	Less than:	40	Less than:	0	40
	Equal to:	60	Equal to:	100	
	Greater than:	0	Greater than:	0	
(2)	Region 1:	39	Region 1:	100	61
	Region 3:	50	Region 3:	0	
	Region 4:	0	Region 4:	0	
	Unsure:	11			
(3)	Same:	30	Same:	100	70
	Different:	70	Different:	0	

[a] Total number of participants = 21.

said that the sound-enhanced version confirmed what they "thought" was the case based on the graph alone.

6. CONCLUDING REMARKS

It is infeasible to directly observe the execution of a program. Thus, one studies the run-time behavior of programs by first mapping some set of execution events onto a set of observable events and then studying the observable events. The observable events will always be different than the actual execution events since they represent a projection of the actual events onto a different space. Yet, it is known that important information regarding run-time behavior can be deduced by studying both visual and textual representations of program execution.

In this paper, we have defined a mapping of parallel program behavior characteristics to sound properties. But just because program characteristics can be represented with sound does not automatically imply that a programmer will be able to relate the aural information back to the behavior of the program in an effective way. Therefore, we have also presented evidence of how program auralizations can, in fact, effectively portray significant behavior information to the listener.

6.1. Observed Benefits of Program Auralization

Based on our experimentations with sound mappings related to a number of different distributed-memory parallel programs, we believe the auralization of parallel programs offers a number of distinct benefits to the programmer. These benefits include the following:

(1) Because one's ears and eyes detect different patterns and different temporal cues, auralization of program events provides for different insights than visualization into the run-time behavior of a program.

(2) Synchronizing an auralization to a static graphical display results in a form of animation of the graphic display. Thus, the temporal relationships in the graphic display become more obvious.

(3) Designing an auralization for a particular program is useful, in and of itself, because of the analysis required. Making initial choices about voice, scale, and even group, assignments, will be based on what is expected to happen. Hearing the auralization, then, either confirms or dispels those expectations.

(4) Auralization of program events provides a means to filter through a large trace file to determine which parts of the trace warrant further investigation.

In general, offering multiple perspectives of performance data means the user has, in effect, more than one tool to use for gaining insight into a program's behavior. This is why visual tools such as ParaGraph offer more than one type of display. The argument being made here is not that information gathered through auralizations cannot be gathered through visual or textual representations, but rather that auralizations can offer the user another perspective that enhances the graphical display and thus allows the user to gain more insight into the program faster than by studying the graph alone. As long as the auralization does not detract from the other views, it is better to offer both than just one.

A more intangible benefit to auralization is that of capturing the *signature* of a program. The overall impression or signature of a program is a composite of all of the program behavior characteristics, to some degree, that potentially captures the inherent qualities of the program. We look to recognize the signature of a program in order to compare it to other programs—possibly one with a different communication topology—or to the same program run with different data sets, or even to compare program executions across multiple architectures. Aural signatures of parallel program behavior provide a mechanism for this type of comparison. Recognizing similarities or differences in a comparison can potentially stimulate intuition about the reasons behind them, resulting in fruitful discoveries. Our recognition of the spoken word, even when spoken in many different voices with various accents, is evidence of our aural abilities to handle this.

6.2. Practicality of Program Auralization

The practicality of using auralizations for understanding parallel program behavior depends on a number of factors. For one, users will have to learn how to listen to these program representations. This does not mean all users will need professional musical training. Rather, it means users will have to become used to interpreting what they hear in the same way that most are now used to interpreting graphical representations.

The current sound capabilities of the average scientific workstation are not adequate to produce the kinds of auralizations used in this study. Instead, we used external sound equipment and a software device driver. The sound equipment necessary, however, is relatively inexpensive, assuming that the auralizations are defined as MIDI files. An adequate system, consisting of a serial-to-midi interface, a Yamaha TG77 Tone Generator, and a set of stereo headphones, costs less than $1500. Although this type of equipment is unfamiliar to most computational scientists, it is affordable and standardized.

We have found the MIDI protocol to be adequate for our purposes to date. As workstations emerge, possessing more sophisticated sound-generation hardware that is capable of producing high fidelity stereo sound (suitable for use in multi-media applications), alternatives to MIDI-based sound generation will be feasible. This, of course, will also broaden the user base for auralization tools.

Last, but not least, performance tools will need to be designed to include an auralization interface. Trying to add this interface into an existing tool, such that the auralization is synchronized and coordinated with the graphical views, is not a trivial task. The Pablo Performance Analysis System being developed at the University of Illinois [15] is an example of a tool designed to include auditory information that is synchronized with visual graphics. An early prototype of a sonic widget for controlling the auralizations allows the user to specify such parameters as scale, instrument, and volume. This is the type of coordinated environment that will be necessary for an adequate study into the effectiveness of using sound for debugging and tuning parallel programs.

6.3. Future Work

This study has focused on investigating the appropriateness of using sound to depict the kind of information necessary to understand how a parallel program is working. Although the initial research has given promising results, the study is by no means complete. Further investigation into effective sound mappings of parallel program behavior information is necessary. Determining what parameters should be at the user's control for experimenting with auralizations versus which ones should be, and even can be, automatically generated based on trace data should also be studied.

The most effective auralizations will be a function of not only the sound mapping used but also the sound voicing used. Basically, sound voices can be categorized into one of three groups: musical instruments, natural sounds, and synthesized sounds. When trying to create aural analogies for certain program behavioral aspects, different kinds of sounds will be more appropriate than others. Research based both on the theory of sound, as well as experience with auralizations, is needed to identify families of sounds that are most effective for representing particular kinds of program behavior. Work by others investigating the auditory representation of scientific data should be applicable to this problem.

REFERENCES

1. Bly, S. A. Presenting information in sound. *Proc. CHI '82 Conference on Human Factors in Computer Systems, 1982*, pp. 371–375.
2. Cherry, E. C. Some experiments on the recognition of speech with one and two ears. *J. Acoust. Soc. Am.* **25** (1953), 975–979.
3. Couch, A., and Krumme, D. Monitoring parallel executions in real time. *Proc. Fifth Distributed Memory Computing Conference.* IEEE Comput. Soc., New York, 1990.
4. Francioni, J. M., Jackson, J. A., Albright, L. The sounds of parallel programs. *Proc. Sixth Distributed Memory Computing Conference, Portland, Oregon, April 1991*, pp. 570–577.
5. Francioni, J. M., and Rover, D. T. Visual–aural representations of performance for a scalable application program. *Proc. Scalable High Performance Computing Conference.* IEEE Comput. Soc., New York, 1992, pp. 433–440.
6. Frenkel, K. A. An interview with Fernando Jose Corbató. *Comm. ACM* **34,** No. 9 (1991), 83–90.
7. Geist, G. A., Heath, M. T., Peyton, B. W., Worley, P. H. A user's guide to PICL: A portable instrumented communication library. Technical Report, ORNL/TM-11616, Oak Ridge National Laboratory, October 1990.
8. Gustafson, J., Rover, D., Elbert, S., and Carter, M. The design of a scalable, fixed-time computer benchmark. *J. Parallel Distrib. Comput.* **12** (1991), 388–401.
9. Heath, M. T., and Etheridge, J. Visualizing performance of parallel programs. *IEEE Software* (special issue on software for performance analysis, September 1991).
10. Lunney, D., and Morrison, R. C. High technology laboratory aids for visually handicapped chemistry students. *J. Chem. Ed.* **58,** No. 3 (1981), 228–231.
11. Madhyastha, T. M. *A Portable System for Data Sonification.* M.S. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1992.
12. Mezrich, J. J., Frysinger, S., Slivjanovski, R. Dynamic representation of multivariate time series data. *J. Am. Statist. Assoc.* **79** (1984), 34–40.
13. *MIDI—Musical Instrument Digital Interface. Specification* 1.0, The International MIDI Association, North Hollywood, CA, 1983.
14. Rausch, R. A., and Plomp, R. The perception of musical tones. In Deutsch, D. (Ed.). *The Psychology of Music.* Academic Press, New York, 1982.
15. Reed, D. A. *et al.* Scalable performance environments for parallel systems. *Proc. Sixth Distributed Memory Computing Conference, Portland, Oregon, April 1991*, pp. 562–569.
16. Scaletti, C., and Craig, A. B. Using sound to extract meaning from complex data. *Proc. SPIE Conference 1459, San Jose, California, 1991.*
17. Smith, S., Bergeron, R. D., and Grinstein, G. Stereophonic and surface sound generation for exploratory data analysis, *CHI '90 Conference Proceedings.* Addison-Wesley, Reading, MA, 1990.
18. Sonnenwald, D. H., Gopinaht, B., Haberman, G. O., Keese, W. M., III, and Myers, J. S. InfoSound: An audio aid to program comprehension, *Proc. Twenty-Third Hawaii International Conference on System Science*, **11,** (1990), pp. 541–546.
19. Wenzel, E. M., Fisher, S. S., Stone, P. K., and Foster, S. H. A system for three-dimensional acoustic "visualization" in a virtual environment workstation, *Proc. Visualization '90, Los Alamitos, October 1990*, pp. 329–337.
20. Wightman, F. L., and Kistler, D. J. Headphone simulation of free-field listening, I, II. *J. Acoust. Soc. Am.* **85** (1989), 858–878.
21. Yeung, E. S. Pattern recognition by audio representation of multivariate analytical data, *Anal. Chem.* **52,** No. 7 (1980), 1120–1123.

Accepted January 22, 1993

JOAN M. FRANCIONI received the B.S. degree in Mathematics from the University of New Orleans in 1977, and the M.S. and Ph.D. degrees in Computer Science from Florida State University in 1979 and 1981, respectively. Currently, she is an Associate Professor of Computer Science at the University of Southwestern Louisiana. Dr. Francioni's main area of research falls into the category of debugging/performance tools for parallel programs. Dr. Francioni is a member of the IEEE, ACM, and Sigma Xi organizations.

JAY ALAN JACKSON received the B.S., M.S., and Ph.D. degrees in Mathematics from Florida State University in 1977, 1979, and 1985, respectively. He is currently an Assistant Professor of Computer Science at the University of Southwestern Louisiana. The focus of Dr. Jackson's research is on numerical algorithms for parallel computers and on auditory display techniques. Dr. Jackson is a member of the SIAM and ACM organizations.