Monitoring the behavior of parallel programs: how to be scalable?

Jean-Yves Peterschmitt, Bernard Tourancheau, Vigouroux Xavier-Francois

Laboratoire de l'Informatique du Parallélisme

Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

Monitoring the behavior of parallel programs: how to be scalable?

J.-Y. Peterschmitt
B. Tourancheau
X.-F. Vigouroux

August 26, 1993

Research Report N° 93.22

Ecole Normale Supérieure de Lyon
46, Allée d'Italie, 69364 Lyon Cedex 07, France,
Téléphone : + 33 72 72 80 00; Télécopieur : + 33 72 72 80 80;
Adresses électroniques :
lip@frensl61.bitnet; lip@lip.ens−lyon.fr (uucp).

Monitoring the behavior of parallel programs: how to be scalable?

J.-Y. Peterschmitt
B. Tourancheau
X.-F. Vigouroux

August 26, 1993

Abstract

It is easy to find errors and inefficient parts of a sequential program, by using a standard debugger/profiler, but there is no such tool in a parallel environment. The only way to study the race conditions of a parallel program is to execute it and collect data about its execution. The programmer can then use the generated trace files and specialized tuning tools to visualize and improve the behavior of the program: idle processors, communications, etc. The problem in large parallel systems is that these tools have to deal with an enormous amount of data. The classical approach to monitor and trace analysis (i.e. sequential, event driven, post-mortem monitoring) is no longer realistic. To avoid this bottleneck, we introduced PIMSY (Parallel Implementation of a Monitoring System). The main idea of PIMSY is to let the trace data distributed among the parallel storage and to distribute the program (or the programs) that deal with the trace data.

Keywords: monitoring, scalability

Résumé

Grâce à l'utilisation d'un débogueur/profiler, il est facile de trouver les erreurs et les parties inefficaces dans un programme séquentiel. Mais il n'existe pas d'outils homologues dans un environnement parallèle. La seule solution pour étudier le comportement d'un programme est de l'exécuter et de récupérer les informations concernant cette exécution. Le programmeur peut alors traiter à l'aide d'outils appropriés les fichiers de trace afin de visualiser et d'améliorer le programme : processeurs inactifs, communications, ... Un problème apparaît avec les systèmes massivement parallèles, c'est celui de la grande quantité d'information qu'ont à traiter ces outils. L'approche classique du monitoring et de l'analyse de trace (c.-à-d. séquentiel, post-mortem, basé sur l'événement) n'est plus viable. Pour éviter ce goulot d'étranglement, nous présentons PIMSY (Parallel Implementation of a Monitoring System). L'idée centrale de PIMSY est de conserver l'aspect distribué des fichiers trace lors de leur génération. Pour cela on utilise un système distribué de fichiers de trace qui sont manipulés par un programme, lui-même, parallèle.

Mots-clés: monitoring, scalabilité

1 MONITORING

1.1 Introduction

The behavior of parallel programs depends on many parameters [CBM90, GMGK84, JLSU87, Mil92] that are in general independent of the user program. This non-deterministic behavior makes the programming difficult. Furthermore, because of the lack of global state [DHHB86, DHHB87, CL85], the classical (i.e. sequential) debugging is no longer possible. The programmers must find a different way to make their programs work. One solution is to record the events that occur when the application runs on the parallel machine, and then compare the theoretical predicted behavior of the program and the observed behavior.

To precisely record the behavior of the program, every events must be saved to allow for replay after the program has executed: variable assignment, messages exchanges, instant of occurrence, etc. However the amount of information would be enormous.

With the following example, the reader should get a better grasp of the problem: Given a target system with 16-i860 processors, running at 100 MIPS (2 instructions per clock cycle, and 50 MHz clock). Suppose that one event (8-byte long) is generated every 10,000 instructions, by a simple computation, we find that the number of bytes generated per second is **1,280,000**. Furthermore, with 128 nodes **10 Mbytes/sec** would be generated. This is impossible to manage such a flow without altering the network behavior or allocating the entire memory of each node.

To reduce the amount of information, we must select the type of *events* we want to monitor.

1.2 Three-phase monitoring

When observing a parallel system, the activity of gathering and using run-time information can be split into three reasonably independent phases (see figure 1):

The generation of the runtime information is done by software probes inserted in the source code, instrumented libraries or hardware components of the machine. The first two solutions are *intrusive* but portable. The hardware one is not intrusive (if the monitoring system has its own bus), but is not portable at all.

Storing the information and making it available where it is required. It is possible to chose when this stage takes place: download immediately, download progressively, download afterwards. Obviously, a fourth method must be added to the first three ones to avoid overfull: downloads when buffer is full.

The analysis of the information consists of interpreting the data and using for the purpose it was created for.

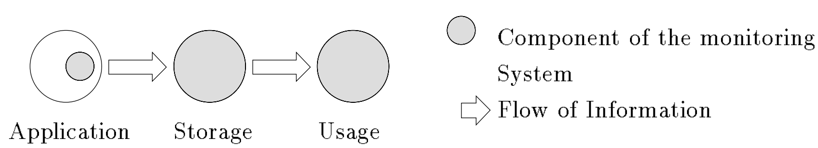

Figure 1: The three phases of gathering and using runtime information

Each component of the operation deals with the total amount of data. With the increase in the number of nodes, it is obvious that these three phases will not be able to manage the entire trace file any more. The aim of PIMSY is to make a first step to make the monitoring really scalable. To succeed, we need a fundamental assumption: The trace file is distributed on different storage sites.

2 PHILOSOPHY OF PIMSY

As we have seen in the previous section, the problem with monitoring a parallel program is in the amount of data generated during event tracing.

A number of efforts have been proposed to reduce the amount of data in performing a trace: [NM92] evaluates if a communication has to be monitored to only keep the causality, [CK90] deals with clumping (recursive grouping of information), [Imr92] explains the combination of low level events to obtain high level ones, and [GHSG92, Moh92, MN90, MRR90, ROA+91, vRT91] speak about trace formats and (general) filtering.

The solution we consider is different than these. We choose to parallelize all the phases of the monitoring process. The first stage is already a concurrent computation but the two others are usually sequential. Thus we try to have them run in parallel.

The parallelization of the *storage phase* can be achieved by saving the information on different storage sites. This is the central request of PIMSY. Fortunately, many new parallel machines [Del91] usually provide distributed storage. In this way, the `load` and the `save` operations are quicker (see figure 2 and 3).

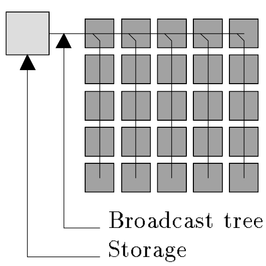

Figure 2: One storage place

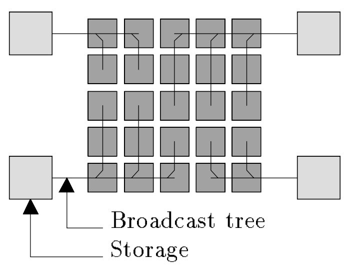

Figure 3: Multiple storage places

Concerning the third phase ("using the monitoring information"), the parallelism is a consequence of the second phase. Indeed, if the trace file is split according to time, space (processors) or event type, the information analysis can be then distributed in the same way.

Another goal is to reduce the time between the generation and the analysis of trace data to provide on-line monitoring. Thus, by performing the monitoring directly the parallel machine, a trace generating process will be able to communicate efficiently with the analysis tool. This situation has the advantage of being between the on-line[1] and the off-line[2] approach.

With these considerations, we will introduce PIMSY with its two main components: first, the scalability and then the extensibility.

2.1 Scalability

"Scalability has no commonly accepted, precise definition" [NA91]. although the authors present the *algorithmic scalability* as opposed to the *architectural scalability*. Their definition is quite good:

[1] As soon as produced, the data is used

[2] The data is first stored and afterward analyzed

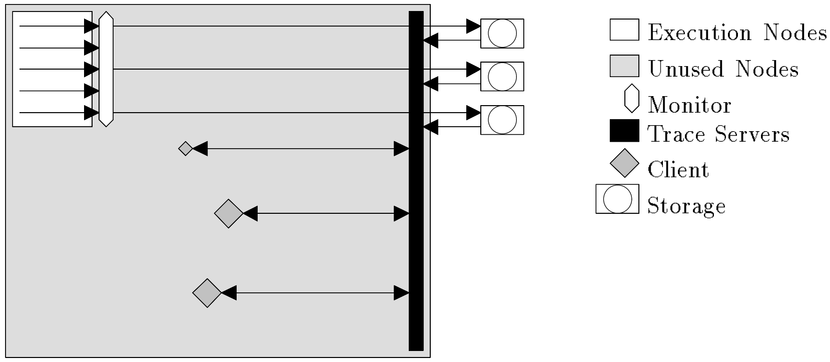

Figure 4: Relation between the generation of trace data and its analysis in PIMSY components

Algorithmic scalability is related to the parallelism inherent in an algorithm, and can be measured through its speedup on an architecture with an idealized communication structure.

Following this last definition, we want to have the best scalability for the entire monitoring tool.

2.2 Extensibility

The second characteristic that is satisfied by PIMSY is the *extensibility*. We want a tool as general as possible, so that, each user can configure it as he wants. It's obvious that the visualization of SIMD[3] computers applications is not the same that the ones used for MIMD computers[4]. Furthermore, an expert does not want the same information displayed as a novice [RAM+92a]. Thus, a user must be able to build his own set of analyzing view that he wants to work on. He must be given a set of tools to allow him to add the ones he wants. And, also, the possibilities of building his own, on top of the management layer must be possible.

Basing our conception on that paradigm, the chosen structure of PIMSY is very simple: the software is layered. One layer managed the visualization tools (video, audio, text,...) which are tasks running on the parallel machine. So their number and type can be chosen by the user. Another layer gives the appropriate information to the first one. Finally, a third layer deals with the files and filters.

3 PIMSY

3.1 Hardware

The parallel machine is composed of several nodes and several hard disks,

The number of **hard disks** is proportional to the number of nodes. For example, we can suppose that if there is $O(p)$ nodes, the machines has $O(\sqrt{p})$ hard disks.

During the generation phase, each node can save the information generated locally in a trace file on the associated hard disk.

Each node has a local memory and a local clock. One problem in the analysis of monitoring data is the lack of global time. There is no way to synchronize perfectly two nodes by exchanging

[3] Single instruction, Multiple Data (according to the Flynn Classification)
[4] Multiple instruction, Multiple Data

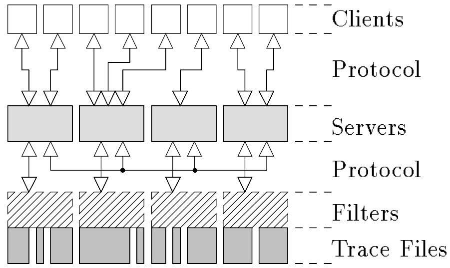

Figure 5: The architecture of PIMSY

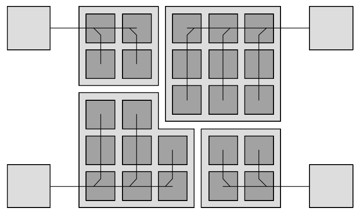

Figure 6: Example of machine partition with regard to the nearest hard disk

messages. A hardware solution has been built by [MR90] with *Hypermon* to solve this problem, but additional hardware is always costly, complex and not portable. Several papers [CL85, DHHB86, DHHB87, Jéz89, Mat89, SM92] have proposed different software approaches to construct a global time as accurate as possible.

There are two classes of such algorithms:

- The first ones are based on a linear drift of the clocks [DHHB86, DHHB87]. A statistical study can then be used to synchronize them.
- [CL85, Jéz89, Mat89, SM91] order the events with these two rules:
  - two events on the same node are ordered,
  - the reception of a message takes place after the emission of the same message (see fig. 7).

  [SM92] enumerates very clearly the different existing algorithms.

We chose is to synchronize the different clocks afterwards. Since the trace file is split in two parts, we can perform the synchronization in parallel. We consider that the clocks have the same speed or that their speed difference is negligible. We made some tests on the Volvox machine of Archipel (see results on fig. 8). This machine is composed of i860 and T800. The drift between the nodes was approximatively: $d(t) = cte + 1.23.10^{-5}t$. According to the constructor, the oscillator frequency is accurate $\pm 5.10^{-5}$ seconds, therefore we are in the accuracy interval ($\pm 10^{-4}$).

The synchronization is achieved by just adding an offset to each local clock. To compute this offset we use the communications that are recorded in the trace file.

Concerning the physical **network**, no assumption is made, but we can say that, to make PIMSY faisable, the logical network must allow at least the communications shown figure 9. This logical

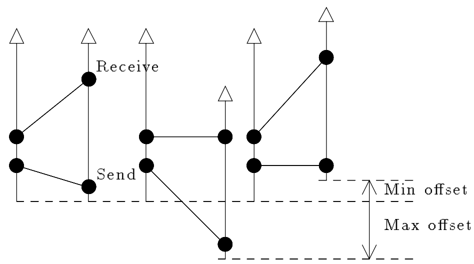

Figure 7: Minimum and maximum offset allowed according to two communications

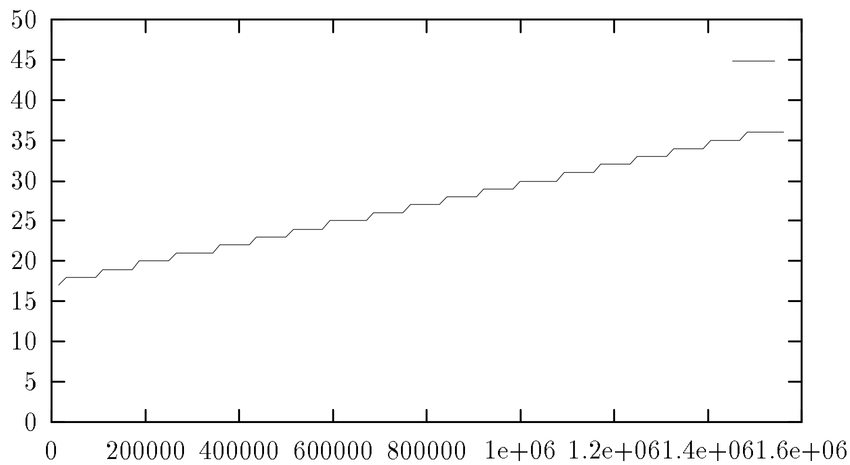

Figure 8: drift between two T800 as a function of time

topology is clearly induced by the communications described in section 3.2 and 3.6. Note that clients and servers can be placed on the same nodes.

With a physical topology that matchs the logical one, the communications do not need to be routed across intermediate nodes, since they are point to point.

We assume that the communications are asynchronous to avoid wasting time when the source and the destination are not synchronous, the messages are received in a mailbox that is checked as soon as possible.

3.2 Software

3.2.1 the operating system

The trace files are split on different hard disks (or storage sites), thus, not to lose the advantage of the repartition of the servers , the operating system must make it possible to the servers to select the hard disk they want to read. If this is not possible, the repartition of the data will be hidden and the mapping of the server will no more be possible.

Furthermore, always for the sake of efficiency, we must be able to choose a mapping from process to processor. This functionality must exist for the two kinds of processes: one for the servers and

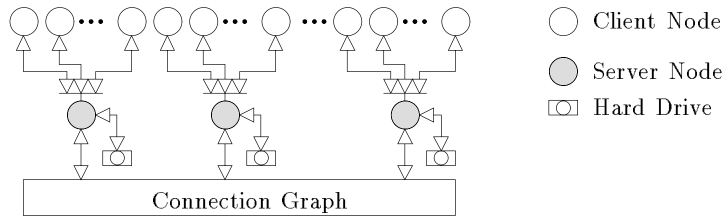

Figure 9: The minimal network

another for the clients. The former comes from the fact that the servers only access their own disks. Therefore, the distance between them and the disks must be as short as possible. The latter is also induced by efficiency constraints but is not really necessary. Actually, the selection of a server for a new client and the load induced by the client can not be predicted. Thus, the server should be able to transfer a client to another server because of overload. This possibility implies to monitor the server themselves.

3.2.2 The source

All the PIMSY servers are written is C++. This choice comes from the fact that the C++ is an oriented-object language and because it is a superset of C. Furthermore, the reusability of the C++ ensures the lifetime of the project.

3.2.3 The parallel machine

Instead of directly using a parallel machine to execute PIMSY, we use a *Parallel Virtual Machine*, thanks to **PVM** [BDG+91, BDG+92]. PVM is currently developed by the Computer Science Department in the University of Tennessee, Knoxville. The primary goals of the tool are portability and the use of heterogeneous systems. We are particularly interested in the first one.

PVM is a software package that allows a heterogeneous collection of serial, parallel and vector computers hooked together by a network. The user views the resulting machine as a loosely coupled, distributed-memory computer, programmed in C, Fortran or C++ with message passing extensions. To configure the machine, PVM only uses a list of names or network addresses. This is made possible by a deamon that runs and manages the communications. One user can get only one virtual machine at a time.

PVM library contains a set of routines that allow parallel programming: synchronous and asynchronous communications, status of a process, processes spawning, barriers, etc.

In PIMSY, we try to use as many standard and portable routines as possible, to allow easy portability of the resulting source.

3.3 Trace Files

During the execution of the application, the events are generated locally. Usually, these local files are merged into a single one before reaching the data analysis step. This is the case in Pablo [RAM+92b, RAM+92a].

For PIMSY, the trace files must not be merged, because the parallelism that we want to have would disappear.

The only assumption on the trace files is the consistency : the local clocks need to be synchronized thanks to the addition of an offset. If there is no synchronization, the local timestamp of a send could be greater than the one of the reception. Currently, the accepted trace files follow the ParaGraph format [HE91] defined in [vRT92]. PIMSY will accept self-defining trace files. Two approaches are possible: (1) each file contains a header that defines the grammar used in the body of the file. This strategy is used in the SDDF[5] of Pablo which description can be found in [Ayd93]. The other solution, chosen by B. Mohr for SIMPLE[6], consists of uses a separate description file. This file, written in TDL[7], can be reused several times. This file can be seen as a monitor description rather than a file description.

For us, this last point of view is better for PIMSY. Because, the replication of the data is not too abusive. Each hard disk contains one description file per generator type which will be read before the data. So different machines (even more different monitor version) can use different trace formats and be analyzed indifferently by using PVM and PIMSY.

3.4 Trace Servers

The *trace servers* are a set of tasks that reply to requests made by the analysis tools (*views*). Each one takes care of the part of information it has. Each trace servers is associated to a hard disk, more generally a storage site. Thus we can equally speak of a trace server or its hard disk.

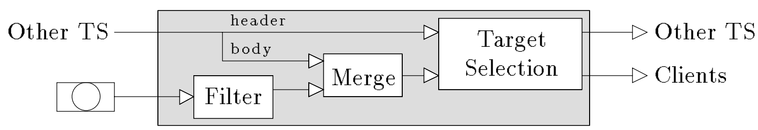

Figure 10: The architecture of a Trace server

A trace server has four communication channels used to propagate the trace information.

Hard Disk IN : a TS uses this channel to get trace data from the hard disk it owns. Before using the information, the TS filters it according to the associated request. This filtering operation must be done as soon as possible to limit the amount of data that goes through on the network.

Note that one goal of PIMSY is to reduce the gap between generation and analysis of the trace data. Thus PIMSY could be supplied with its own event generator. This generator would directly use this channel, without storing the information on the hard disk. This way, PIMSY would manage the three phases of monitoring (see figure 1). Currently, PIMSY uses static files generated by a trace system

Trace Servers IN : this channel is used to receive data from the other TS. The received information is already correctly filtered, therefore no additional processing needs to be performed on it.

[5] **S**elf **D**efining **D**ata **F**ormat

[6] **S**ource related and **I**ntegrated **M**ultiprocessor and computer **P**erformance evaluation, mode**L**ing and visualization **E**nvironment

[7] **T**race **D**escription **L**anguage

The trace information that comes from the filtering operation and the one that comes from the other TS are merged into one. The *merging* operation must choose a total order, for example, the one implied by the timestamps[8].

Trace Servers OUT : this channel is used to send trace data. Once the trace information generated, the TS select the destinations according to the header.

Clients OUT : the trace data that transit by this channel has been necessary asked by a client. As a client can ask information for a set of clients, a destination has not necessary asked for it. But, once a client receive data, it must send back a acknowledgment.

Note that we are not speaking of the channel in. because the clients do not use it to transfer trace information but to ask for information, to indicate their states or to return an acknowledgment.

3.5 Clients

The conditions on the TS and the parallel machine have to be fulfilled by the clients. For example, if the clients are not scalable then there is no need to make PIMSY scalable. The graphic tolls were presented in [PTV92]. Some clients that satisfy the scalability request were also introduced in this paper.

As the user may want to have different views on the same instant of a parallel program, the clients must be able to be synchronized. The solution we choose to achieve this synchronization is to broadcast the result of a request to a set of clients. The client that initiates the request chooses the destination of the trace data. The TS forward the result of the request to this set. Then when all the destination views finish their work with the data, a global acknowledgment is sent to the source. The problem with this solution is that the client must be able to be driven by other ones. When a client is created, it indicates to its TS if it can be synchronized. The following protocol explains more precisely a request of PIMSY.

3.6 Protocol

Here are some definitions to simplify the following:

execution : set of trace files generated during the execution,

workers : set of trace servers that have information about an execution,

source : client that send a request,

source-TS : Trace Server that manages the source,

destination : set of clients that will receive the trace data requested by the source,

destination-TS : Set of TSs that manage at least on destination.

The protocol is straightforward. The *source* asks for a *data slice* (filtered information about the execution) by sending a request to the *source-TS*. We then build a linear network of workers, ending with a merging task. Eventually, the merging TS sends the information to the destinations through their associated destination-TS. After having processed the data, each client sends an acknowledgment to the source-TS which sends global acknowledgment in return.

[8] The timestamp is the common field of all events. Obviously, we do need a field giving the type of the event. The location should not be a necessary field. Indeed trace file could contain non local information, such as statistics

3.6.1 Complexity

Suppose that we have a request for a slice of size l, and that q workers have l/q events of that slice. Since we have to generate and send a sequence of size l, it's obvious that the complexity will be at least in $O(l)$.

Thus, we can choose a simple algorithm to reach this complexity. We suppose that we have a linear network of workers, and that the first worker is to get the entire data. On such a network we use the well known odd-even algorithm (see fig. 11) to sort the data. This way, we only have to concatenate the local lists in linear time.

With the odd-even algorithm each worker communicates in turn with its left and right neighbor. The exchanged message consists in the local trace information. Each worker merges its list and the received one (in $O(l/q)$) and keeps one half: the left worker keeps the lower half and the right worker the upper one. Note that the workers can limit the merging to the half they keep. This operation must be repeated q times, the global time being therefore in $O(l)$.

The concatenation phase, is obviously in $O(l)$.

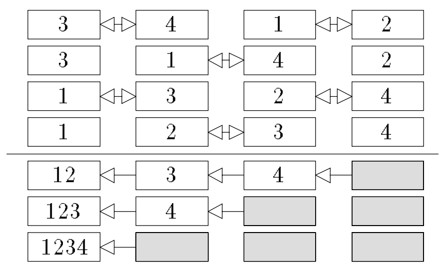

Figure 11: The odd-even algorithm

4 AUDIO TOOLS

4.1 Introduction

We present here our attempt to make small, stand alone programs, that use sound to convey monitoring information. These programs can generate sound in real-time on a common SPARCstation, and can be easily modified to suit the needs of a given user (e.g. doing on-line monitoring). In particular, it will be easy to interface them with PIMSY.

4.2 Conveying data with sound

Using sound in a visualization application allows the programmer to convey new information, without using conventional displays. This has been named *sonification* or *auralization*. Concerning monitoring, [FJA91] focuses on the mapping of events to the MIDI format, and uses the resulting sounds in parallel with ParaGraph. [Mad92] introduces a more general purpose sonification tool, and uses it in the Pablo monitoring environment (see also [RAM+92a]). This tool allows the user to switch easily between using MIDI or SPARCstation sound.

What is maybe most important is the fact that conventional displays rely on seeing, whereas programs using sound related dimensions rely on hearing. These two ways of gathering information 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 its *basic parameters* (pitch, timbre, amplitude, envelope and duration), and have them change over time.

We can also mix sounds together, or change their placement in space using two or more speakers. Note that for obvious technical reasons, we cannot achieve all these sound effects on a standard SPARCstation.

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. 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 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. Yet, we believe that the users will be able to understand increasingly complex parallel programs, thanks to the use of sound, with some appropriate training.

4.3 Sound on a SPARCstation

Sound programs on a 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.

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. For more details, see [Sun92a, Sun92b, VR93].

4.4 Implementation

In this project we did not want 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. Our programs needed to be fast and have at the same time low memory and disk-space requirements.

We got interesting enough results with seemingly very simple sounds waves: basic sine waves.

4.5 The AudioTrace programs

4.5.1 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) 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. The content of the trace file is sorted according to increasing timestamps. 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[9]. The sound is created with a valid audio header, and can be either played directly, or stored for future use.
- the programs are fast. This allows us to generate and play the created sound on the fly. This way, we only have to store the trace file, instead of the much longer resulting sound file.

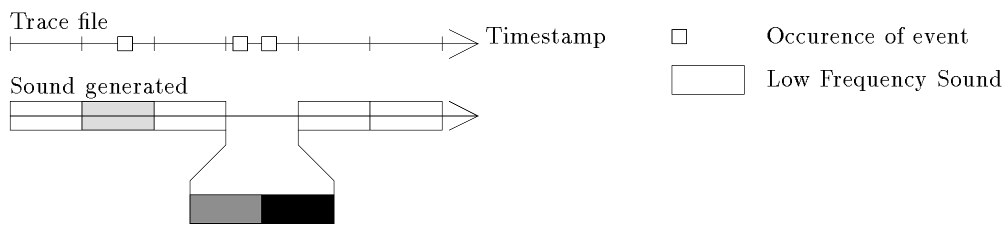

Figure 12: Computing a new wave

Figure 13: Relation between the execution time and the sound duration

- the duration of the created sound is proportional to the execution time of the 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 total duration of the generated sound depends on two parameters, `length` and `scale`, as shown on figure 13. 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[10].

  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 will not 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.

4.5.2 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 trace files). They have four common parameters, specified on the command line:

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

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

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

nb is the number of events we want to map to frequencies. It can be, for instance, the total number of processors involved in the parallel program.

length and scale have already been explained above.

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

```
cat file.trf  |  tr_xxx nb length scale | play
```
[11]

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.

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.

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.

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.

We have shown how easy it is to use sound on a SPARCstation with our approach, and how sound can be used to convey data. 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.

5 FUTURE WORK & CONCLUSION

We are continuing an implementation of PIMSY. A prototype has already been implemented on a LAN of SPARCstations using PVM package. This version shows the efficiency of our approach. The next prototype will be implemented on another distributed memory multi-computer called Volvox manufactured by Archipel. This implementation will show the portability of our approach. Real-time implementation of the trace server is also under study. Such a trace-server will store the runtime information in local memory and be able to serve client requests in a real-time fashion.

We will implement several others tools to read a sufficient set of representations (i.e. visualization and auralization). The existing set of tool is limited but exists.

In this paper, we have first presented our client-server based approach to massively parallel monitoring. In order to avoid the traditional bottleneck of parallel monitoring, we have designed a monitoring system in which not only the generation of the runtime information is distributed, but also the storage and the processing of this information.

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

References

[Ayd93] R. Aydt. The pablo self-defining data format. Department of Computer Science, University of Illinois at Urbana-Champaign, March 1993. `available by ftp anonymous bugle.cs.uiuc.edu:pub/Release-1.1/Documentation/SDDF.ps.Z`.

[BDG+91] A. Beguelin, J. Dongara, G. Geist, R. Manchek, and V. Sunderam. A users' guide to pvm (parallel virtual machine). Technical Report ORNL/TM-11826, Oak Ridge NAtional Laboratory, University of Tennessee, July 1991.

[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 Science Publishers - North Holland.

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

[CBM90] W. Cheung, J. Black, and E. Manning. A framework for distributed debugging. *IEEE Software*, 7:106–115, January 1990.

[CK90] A. Couch and D. Krumme. Monitoring parallel executions in real time. In *Proceedings of the 5th distributed memory computing conference*, volume 2, pages 1187–1196. IEEE, 1990.

[CL85] K. Chandy and L. Lamport. Distributed snapshots : determining global states in distributed sytems. *ACM transaction s on Computer Systems*, 3(1):63–75, February 1985.

[Del91] Intel Supercomputer Systems Division, Intel Corporation, 15201 N.W. Greenbier Parkway, Beaverton, Oregon 97006. *A Touchstone DELTA System Description*, February 1991.

[DHHB86] A. Duda, G. Harrus, Y. Haddad, and G. Bernard. Monitoring of distributed systems. Technical Report 52, ISEM, December 1986.

[DHHB87] A. Duda, G. Harrus, Y. Haddad, and G. Bernard. Estimating global time in distributed systems. In *7th international conference on distributed computing systems*, pages 299–306, Berlin, September 1987. IEEE Press.

[FJA91] J. Francioni, J. Jackson, and L. Albright. The sounds of parallel programs. In Q. Stout and M. Wolfe, editors, *The sixth distributed memory computing conference proceedings*, Frontier Series, pages 570–577, Portland, Oregon, April 1991. IEEE, IEEE computer society press.

[GHSG92] I. Glendinning, S. A. Hellberg, P. A. Shallow, and M. Gorrod. Generic visualization and performance monitoring tools for message passing parallel systems. In N. Topham, R. Ibbett, and T. Bemmerl, editors, *programming environments for parallel computing*, volume A-11 of *IFIP Transactions*, pages 139–150, Edinburgh Holland, April 1992. IFIP, North Holland.

[GMGK84] H. Garcia-Molina, F. Germano, and W. H. Kohler. Debugging a distributed computing system. In IEEE, editor, *Transactions on Software Engineering*, pages 210–219, March 1984.

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

[Imr92] K. Imre. Experiences with monitoring and visualising the performance of parallel programs. In *Workshop on performance measurement and visualization of parallel systemsq*, October 1992.

[Jéz89] J.M. Jézéquel. Building a global time on parallel machines. In LNCS Springer-Verlag, editor, *the 3rd International Workshop on Distributed Algorithms*, pages 136–147, 1989.

[JLSU87] J. Joyce, G. Lomow, K. Slind, and B. Unger. Monitoring distributed systems. *Transactions computing systems - ACM*, 5(2):121–150, May 1987.

[Mad92] T. Madhyastha. A portable system for data sonification. Technical Report UIUCDCS-R-92-1761, University of Illinois at Urbana-Champaign, 1992. `available by ftp anonymous at cs.uiuc.edu:UIUCDCS-R-92-1761`.

[Mat89] F. Mattern. Virtual time and global state of ditributed systems. In Cosnard, Quinton, Raynald, and Robert, editors, *international workshop on parallel and distributed algorithms*. North Holland, November 1989.

[Mil92] B. Miller. What to draw ? when to draw ? an essay on parallel program visualization. to appear - Journal of Parallel & Distributed Computing, 1992.

[MN90] A. Malony and K. Nichols. Standards working group summary. In M. Simmons and R. Koskela, editors, *Performance Instrumentation and Visualization*, Frontier Series, pages 261–278, Santa Fe, New Mexico, May 1990. ACM, Addison-Wesley Publishing Compagny.

[Moh92] B. Mohr. Standardization of event traces considered harmful – or – is an iplementation of objet-idependent event trace monitoring and analysis systems possible ? In J. Dongarra and B. Tourancheau, editors, *Environments and tools for parallel scientific Computing*, volume 6 of *Advances In Parallel Computing*, pages 103–124, Saint Hilaire du Touvet, France, September 1992. CNRS-NSF, Elsevier Science Publishers - North Holland.

[MR90] A. Mallony and D. Reed. A hardware-based performance monitor for the intel iPSC/2 hypercube. In Miller B. and McDowell C., editors, *Proceedings of the ACM International Conference on Supercomputing*, Amsterdam, June 1990. ACM press.

[MRR90] A. Malony, D. Reed, and D. Rudolph. Integrating performance data collection, analysis and visualization. In M. Simmons and R. Koskela, editors, *Performance Instrumentation and Visualization*, Frontier Series, pages 73–97, Santa Fe, New Mexico, May 1990. ACM, Addison-Wesley Publishing Compagny.

[NA91] D. Nussbaum and A. Agarwal. Scalability of parallel machines. *Communications of the ACM*, 34(3):56–61, March 1991.

[NM92] R. Netzer and B. Miller. Optimal tracing and replay for debugging message-passing parallel pograms. In IEEE Computer Society Press, editor, *SuperComputing '92 - Proceedings*, pages 502–511, Minneapolis, Minnesota, November 1992. IEEE, IEEE Computer Society Press.

[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+92a] D. Reed, R. Aydt, T. Madhyastha, R. Noe, K. Shields, and B. Schwartz. An overview of the pablo performance analysis environment, 1992.

[RAM+92b] D. Reed, R. Aydt, T. Madhyastha, R. Noe, K. Shields, and B. Schwartz. The pablo performance analysis environment, 1992.

[ROA+91] D. Reed, R. Olson, R. Aydt, T. Madhyastha, T. Birkett, D. Jensen, B. Nazief, and B. Totty. Scalable performance environments for parallel systems. In Q. Stout and Wolfe M., editors, *The sixth distributed memory computing conference proceedings*, Frontier Series, pages 562–569, Portland, Oregon, April 1991. IEEE, IEEE computer society press.

[SM91] R. Schwarz and F. Mattern. Detecting causal relationships in distributed communications :in serch of the holy grail. IR 215/91, Universität Keiserslautern, Postflach 3049, D-6750 Keiserslautern, November 1991.

[SM92] R. Schwarz and F. Mattern. Detecting causal relationships in distributed communications :in search of the holy grail. Technical Report 15/92, Universität Keiserslautern, Postflach 3049, D-6750 Keiserslautern, December 1992.

[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.

[vRT91] M. van Riek and B. Tourancheau. A general approach to the monitoring of distributed memory machines. Research Report 91-28, LIP – Ecole Normale Supérieure de Lyon, 1991.

[vRT92] M. van Riek and B. Tourancheau. The trace-formats that are used in picl, paragraph and gpms. Technical Report 92-02, LIP – Ecole Normale Supérieure de Lyon, 1992.

[ZT92] E. Zabala and R. Taylor. Process and processor interaction: Architecture independent visual-sation schema. In J. Dongarra and B. Tourancheau, editors, *Environments and tools for parallel scientific Computing*, volume 6 of *Advances In Parallel Computing*, pages 55–72, Saint Hilaire du Touvet, France, September 1992. CNRS-NSF, Elsevier Science Publishers - North Holland.