Monitoring of Distributed Memory Multicomputer Programs

Maurice van Riek
Laboratoire de l'Informatique du Parallélisme, CNRS-URA 1398
Ecole Normale Supérieure de Lyon, 69364 Lyon Cedex 07, France.

Bernard Tourancheau*
Department of Computer Science, University of Tennessee
107 Ayres Hall, Knoxville, TN 37996-1301, USA.

Xavier-François Vigouroux†
Laboratoire de l'Informatique du Parallélisme, CNRS-URA 1398
‡Ecole Normale Supérieure de Lyon, 69364 Lyon Cedex 07, France.

abstract

Programs for distributed memory parallel machines are generally considered to be much more complex than sequential programs. Monitoring systems that collect runtime information about a program execution often prove a valuable help in gaining insight into the behavior of a parallel program and thus can improve its performance. This report describes in a systematic and comprehensive way the issues involved in the monitoring of parallel programs running on distributed memory systems. It aims to provide a structured general approach to the field of monitoring and a guide for further documentation. First the different approaches to parallel monitoring are presented and the problems encountered are discussed and classified. In the second part, the main existing systems are described to provide the user with a feeling for the possibilities and limitations of real tools.

Categories and Subject Descriptor

C.1.2	[Multiple Data Stream Architectures]	MIMD
D.1.3	[Concurrent Programming]	
D.2.5	[Testing and Debugging]	Monitors, Tracing
D.2.6	[Programming Environments]	
D.4.8	[Performance]	Measurements, Monitors

General Terms

Measurement, Performance, Supercomputing.

Additional Key words and Phrases

Distributed memory systems, collecting runtime information, program execution visualization, performance monitoring, debugging, monitoring environments

*On leave from CNRS-URA 1398, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, 69364 Lyon Cedex 07, France. Supported in part by the National Science Foundation under grant ASC-8715728, the National Science Foundation Science and Technology Center Cooperative Agreement CCR-8809615, the DARPA and ARO under contract DAAL03-91-C-0047, ARCHIPEL SA and MRE under grant 974, research program C^3, the CNRS-NSF grant number 950.223/07 and DRET. btouranc@cs.utk.edu

†Part of this work was realized when visiting the CS department of the University of Tennessee. vigourou@lip.ens-lyon.fr

Monitoring models, event-action paradigm, event classes, event detection, runtime information representations, software/hybrid/hardware monitoring, transporting and processing runtime information.

Acknowledgments

The authors gratefully acknowledge the valuable and constructive comments of T. Bemmerl, M. Heath, P. Worley and the referees that helped to improve the earlier version of this paper.

1 Introduction

Parallel programming is generally considered to be much more complex than sequential programming. This complexity makes the prediction of the behavior and the performance of a parallel program often impossible. Just as in other fields of science, measuring on a real system is therefore necessary to determine the real behavior and performance of a parallel computing system.

For measuring on a computing system monitors are used that gather runtime information about the observed system. In sequential programming, monitoring is a relatively well understood field. On the other hand, inherent properties of parallelism make parallel monitoring a complicated and vast field for which sequential monitoring approaches are insufficient. For this reason new techniques that deal with this parallelism are required.

The need for parallel monitors has become even more apparent with the arrival of distributed memory massively parallel machines. The often low speed-up achieved on these machines are a clear indication, that their behavior is often not fully understood. Moreover, regarding the announcement of all the supercomputer manufacturers, these multicomputers seems to be the supercomputers of the very near future.

This report describes in a systematic way the issues involved in the monitoring of parallel programs for distributed memory systems. It aims to provide a structured general approach to the field of monitoring and a guide to further documentation. Thus, stress has been placed on the completeness of the overview, more than on specific details.

Shared memory architectures are not the target of this survey because their limited number of processors keep them out of the trends of the parallel field and we do not want to enlarge an already very broad subject. However, most of the work done for distributed memory architecture holds for shared memory ones and vice-versa.

The aim of this paper is to provide the reader with an introduction to the field of gathering and using runtime information about parallel programs on distributed memory multicomputers, based on a synthesis of the published research in this field. This vast subject, includes many subfields, such as data collection, program tracing, trace transportation and analysis, and, execution and performance visualization.

After some definitions (section 2), two parts can be distinguished in this report. In the first part, we present the event-driven approach (section 3) and its three phases (section 4,5,6). In the second part, We introduce the well-known monitoring systems, and compare them (section 7), then we present the problems induced by scalability (section 8), and finally, we conclude (section 9).

2 Some basic definitions and concepts

We introduce in that section the basic models that we will need for the description of the monitoring of distributed memory parallel machines, and the explanation of resulting classifications.

2.1 Defining a machine model

We consider a parallel machine M that is of the *MIMD*[1] multi-computer type or workstation cluster. It consists of N independent *compute nodes*, that each execute their own instructions independently on their own data. Each node possesses a processor and local resources, such as memory. The model does not include any global resources and in that respect the machine is a truly distributed machine. The different nodes

[1] Multiple Instruction, Multiple Data (Flynn Classification)

are interconnected through a physical *communication network* C. This network consists of a finite set of channels c_i (and all the other devices : buffers, routers), that link two nodes together. The users can access the machine through one or several nodes that are connected to the outside and to the network.

Each node has a local clock. All the clocks have the same precision, but no synchronization mechanism is assumed because a global clock is a very strong assumption on a distributed system and it realization requires very complex hardware. Moreover, several technics will be describe in section 2.10 to restore the causal relationship and provide the user with an estimation of the global time (wall clock) that fulfill its needs.

Interactions between different nodes take place through the sending and receiving of messages. Our machine will be required to be at least *virtually fully connected*. This means that any node can directly send a message to any other node, i.e. an underlying router-mechanism takes care of the routing of the messages through the network if the machine is not physically fully connected. This *router-mechanism* can be implemented both in hardware and in software. Whereas fully connected machines provide constant communication times, router-mechanisms may introduce a non-constant communication delay due to its policy, buffer contentions, etc and can have a large impact on the performance of a program depicted by the monitor.

The communication protocol may be synchronous or asynchronous, and there is no assumption about the number of ports available per processor, and their ability to do half/full duplex communications.

2.2 Defining a program model

For the machine defined in 2.1, a distributed program P consists of m sequential programs. Two types of instructions can be distinguished; those that only concern the local program and those that interact with other programs. Two processes[2] interact with each other by sending and receiving messages that are transmitted over the communication network C (there is no assumption about the type of communication primitives, sends, receives, gets, puts, etc. correspond actually at a certain low level to basic sends and receives).

An executing distributed program can thus be seen as a set of sequential processes that, in an alternating way, compute locally and interact among each other through communication channels. The individual state of each of the programs can be defined just like the state of a sequential program. The state S of P (also called *global program state*) is defined as the sum of all these local program states (i.e. the reunion of the corresponding informations about program counters, registers, memories, etc.) and the state of the communication network (i.e. the messages currently going on, the connection of switches, the content of buffers, etc.).

2.3 Difficulties in understanding parallel program behavior

There are a number of reasons why understanding and evaluating distributed programs is more difficult than for sequential programs. According to [CBM90, GMGK84, JLSU87, Mil92] the following reasons can be distinguished:

- **Size of the system** : Distributed programs tend to be large due to the replication of code (SPMD[3] paradigm). Instead of dealing with one sequential process, the programmer now has to deal with many processes.
- **Multiple threads of control** : Distributed programs possess many threads of control. Therefore the concept of program state needs to be extended to include all the local process states and the states of the communication channels.
- **Inherent non-determinism** : Distributed, asynchronous systems are inherently non-deterministic. This means that two executions of the same system may be different, but produce the same result. Even when a program is correct, it still remains difficult to predict which of the possible correct executions will lead to the final results.

[2] informally defined as a sequential program in execution
[3] Single Program, Multiple Data

- **Nonhomogeneous communication delays** : Although many existing distributed machines are virtually fully interconnected, their communication delays are nonhomogeneous and non-deterministic, due to the usage of routing mechanisms and channel contentions. Predicting communication times is thus impossible.

- **Error latency** : Usually, there is a lag between the occurrence of an error and its discovery. Due to the significant communication delays and autonomous operations, this lag may be much larger in distributed systems than in sequential ones.

As far as the performance evaluation is concerned, the definition of the best algorithm performance criteria is not clear and actually, most of the authors refer to metric related to the problem size [Gus88, CRT89] or scalability and iso-efficiency function [CDW92, GK92].

2.4 The use of runtime information for increased program understanding

Understanding a system in science often means being able to predict the behavior of that system. With complex systems such as parallel computers, this may not always be possible. A less ambitious goal is to find out at least what is precisely going on in the system.

The advantage of knowing exactly what is going on in a system is considerable. For instance, it allows assumptions to be checked against realiy and thus to verify the validity of a model, to detect unsuspected system behavior, to localize performance bottlenecks and errors, ...

Parallel monitors collect data during execution to provide the user with measurements. As in other scientific domains, measurements on parallel computers might influence the behavior of the system in such a way that the measurements no longer reflect the real behavior of the unobserved system. However the precise effect of current monitors on the behavior of an executing program is a largely unresolved question. Two different approaches to this problem can be distinguished. The common approach is to suppose that the influence of the monitoring system on the observed execution exists and that its effects should be minimized. The remaining influence is negligible. The opposite approach judges any influence unacceptable and requires the addition of complex hardware to monitor the observed system without interference.

Whether the advantages of one of these approaches outweigh its cost also remains an open question, and many compromises are possible. The question of the intrusiveness will be discussed in more detail in the sections that deal with the actual generation of the runtime information (see section 3).

It is also possible to imagine that several software monitoring technics are buried into the system software and are always turn on. This does not enter the scope of our study because we consider that these hooks are not necessary and so could be remove or turn off to increase the program execution speed.

2.5 Approximation of the evolution of the program state

Since a parallel computer system is a collection of Turing machines, it goes through a finite number of states. So it would be enough to know all the states, the system went through from the start to the end of a program execution, to know precisely what went on in the machine. This observation can be done in two different ways [mL92]:

- The first one consists of **time-sampling** the program state. With the help of these time-samples, one can reconstruct the evolution of the program state. The advantage is that the user can choose the induced overhead by choosing the sample intervals. However, if the time interval becomes too large, an exact reconstruction is no longer possible.

  With the advent of microprocessors and chip caches, however, it has often become impossible to record the state of a program and to our knowledge no existing monitoring systems for message-passing architectures have been based on it.

- The other approach is **event-driven** and consists of recording all the program state changes. The initial state and its changes are enough to reconstruct the program evolution. Obviously, even if it were possible to collect all this information, the amount would be too enormous to be useful. Therefore often only the interesting events are traced and the evolution of the program state is approximated by leaving out the uninteresting information. This method has been generally adopted in monitoring tools [Mal89, PTV92].

By defining a higher level of abstraction (i.e. by changing the granularity) at which we look at the program (for instance looking at procedures instead of instructions), its state representation can be considerably simplified, thus reducing the number of changes to monitor. Obviously, there is a clear tradeoff between the precision of the approximation, the level of abstraction and the overhead generated by the monitoring system.

2.6 The event-action paradigm

In the following, we will refer to a process as being the elementary flow of control running independently. From the user point of view, it mostly correspond to the general notion of procedure or function that can be called independently. This notion is common in the CSP (Communicating Sequential Processes) programming model and appear in a more hidden way with the SPMD (Single Program Multiple Data) model if, for instance, communication procedures can be ran asynchronously on each node.

2.6.1 An informal introduction

An event is defined as a change in the state of a process at a specific point in time. In the *event-action paradigm* the execution of a distributed program is viewed as a set of sequences of events. The events in a sequence are ordered according to their moment of occurrence. An event has no duration. Globally two types of events can be distinguished : events that are local to a process, such as the assignment of variables and events that represent interactions between processes such as the communication of data. Figure 1 shows this way of looking at a program. Each process is displayed by a vertical arrow, representing its local time, where the dots stand for events. Pairs of events that represent an interaction between two processes are linked by a diagonal arrow. There is no assumption of the scale of each time-line.

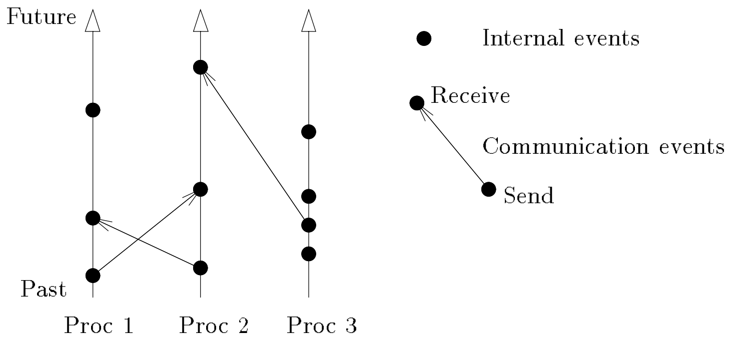

Figure 1: Model of a program as sequences of events

Monitoring a distributed program according to the event-action paradigm means looking for specific event occurrences.

2.6.2 A formal definition

Let a subsystem be a set of processes. Two disjoint subsystems can be identified in a parallel machine [MLCS92]. They are the subsystem $\mathcal{A}$ that consists of the application processes called *active processes* and the monitoring subsystem processes *reactive processes* $\mathcal{R}$ that provides the needed monitoring functions. This distinction allows to consider easily a non-monitored program as only the *active processes* while a general program is $\mathcal{P} = \mathcal{A} \cup \mathcal{R}$.

Consider the machine introduced in section 2.1, on which a finite set of processes $\mathcal{P}$ is running. A *message* is a data structure used to send information from one process to another. A *message channel* is an abstract structure that represent the receiver or the sender of a message. The state of a channel is the sequence of messages presently being sent along the network using that channel.

An active process $p \in \mathcal{A}$ is defined by a set of states, one of which is denoted as the initial state, and a sequence of events. The state of an active process is determined by the value of all its variables including its program instruction counter and its channels states.

Following [CL85], an *event* e in an active process p is defined as an entity without duration that reflects a change in the state of p or in a channel adjacent to p at a specific time.

A reactive process $r \in \mathcal{R}$ is defined by : the subset of active processes r is supervising, the set of event to recognized, and the set of action (sequence of instruction) to performed. A reactive process is initially in a suspended state i.e. it is waiting. It is activated by the occurrence of an event in one of the active processes it supervises. After identifying an event, the reactive process determines the action to perform, performs it and then returns to the suspended state.

Often interesting aspects in the behavior of a monitored system are determined by a sequence of events, called an *activity* [Moh90]. An activity could be defined by a regular expression of events or previously defined activities. Informally, one could say that the activity describes the actions that lead to the occurrence of an event. Whereas attributes can be defined for both events and activities, a duration-attribute is only meaningful for activities.

2.7 Definition of the different classes of events

Events can be grouped into different classes with common characteristics. The criteria for this grouping can be based on the level of abstraction in the system that is related to the event, the nature of the event, the way an event is detected and its role in a monitoring tool. Classes based on each of these criteria will be introduced in this section. These classes will be used later in the deduction of a "general purpose" set of events to monitor and in the discussion about trace generation.

Moreover, events can be combined to create new events :

- *Atomic events* mark a simple change of state of an object of the monitored system. Atomic events are local to compute nodes and can be detected by a local monitor. They form the base of most existing monitoring tools and are characterized by their moment of occurrence.

- *Compound events* are events that are defined as a combination of other events (atomic or compound). Compound events are specified by expressions over other events. A compound event has occurred if all the conditions of its defining expression are fulfilled. Just like an atomic event, a compound event has no duration and is defined by its moment of occurrence which corresponds to the moment when the expression is fulfilled. An *activity* can be associated with its corresponding duration (see section 5.2).

  According to [BLT90] the necessary operators for combining events are :

  - event e_1 or event e_2 happens : $(e_1 \vee e_2)$
  - event e_1 happens before event e_2 : $(e_1 \rightarrow e_2)$
  - both events happen in any order : $(e_1 \wedge e_2)$

  Example :
  - `The reception of a message could be defined as :` $(e_1 \rightarrow e_2)$, `where` e_1 `is the start of the receiving and` e_2 `the end of the receiving. Its moment of occurrence will be` e_2 `and it has no duration but an activity can be associated to` (e_1, e_2) `that, for instance, will have a duration` $(occurrenceof e_2 - occurrenceof e_1)$.

  Compound events can be both local and distributed among several nodes. However, the detection of *distributed compound events* requires complicated detection mechanisms and can introduce considerable overhead.

Compound events do not create new information, but essentially group events. Unless the runtime information is used in real-time, this grouping can also be done in a post-processing step by analyzing the generated traces. This activity is called clustering (see section 5.2).

Most tools do not allow the user to specify compound events because of the complicated detection mechanisms that it requires. TOPSYS (see section 6.4 [BLT90] is one of the few monitors that does allow the user-specification of compound events.

From a conceptual point of view, atomic events can be divided into three fundamental types of events (see Figure 2).

- *Execution events* trace the flow of control in a process. Such events occur when a process reaches a prior defined statement. Basically, all monitoring tools allow to a certain extent the detection of execution events.

  Example : • `process` $p-2$ `executes instruction #69 for the` 7^{th} `time`

- *Data events* Data events trace the flow of data in a process.

  Example : • `variable a of process p1 is assigned the value 350`

  Through data event detection, the evolution of the contents of a variable can be observed. This can reveal a powerful aid in tracking errors in distributed programs. Although this is traditionally the domain of parallel debuggers, *data event* detection is easy to implement for simple variables, thus providing a powerful feature at little cost. TOPSYS currently offers the possibility of monitoring the reading and writing of a user-defined variable [BLT90].

- *Parallelity events* deal with the parallelism in distributed programs. They occur when a process interacts with another process. Under normal conditions parallelity events always come in pairs, because a parallelity event in one process induces a parallelity event in another process.

  Example : • `process` p_2 `sends a message n to process` p_3
  • `the message queue of a process is incremented by one`

  Since parallelity events are the only events that deal with the parallelism of a distributed program, they play an important role in deadlock detection, and in the tracking of performance bottlenecks and communication errors.

  Monitoring parallelity events is what distinguishes distributed monitors from sequential monitors and all existing tools allow for the detection of parallelity events. Some high level tools mainly limit themselves to the observation of parallelity events.

Events can be grouped according to their visibility from within an application process (Figure 2). This division will be used in the discussion about the detection of events in sections 3.3, 3.4, 3.5. The following two classes can be distinguished :

- *Application events* are defined as events that mark the change of state of an application process. Since this state is visible from within a process, these events are easily to detect.

  The examples we have shown until now were essentially all application events. Communications, as far as sending and receiving are concerned, are considered to belong to this class.

- *System events* Informally, *system event*s are concerned with the characteristics of the system and are somewhat independent of the application. Although most system events are induced by application processes, they are normally invisible to the user.

  Example : • `a message buffer of a router overflows`
  • `a process is activated or put in a waiting queue`

System events are important for performance monitoring, because it is often at the system level that information about performance, such as waiting times, process scheduling and context switching is available. System events may also provide valuable clues about failing executions of apparently correct applications, due to, for example, buffer overflows.

To detect system events access to the state of the operating system on a node is required. Unfortunately this state is usually hidden from the application program and special provisions in the operating system are needed to access it. For this reason, system events are among the most difficult to detect and their detection mechanisms are the least portable.

Few tools offer the possibility of extensive system event detection. Those tools that do offer this possibility usually use a modified version of the operating system or use available system calls. The MMK kernel of TOPSYS is an example of a tool where the operating system has been modified [BL92]. There is also a modification of Intel OS NX/2 tool in [MRR90] and Trollius OS [vRT92c].

In [LCSM92] two classes of events are introduced that group atomic events according to their availability in monitoring tools. This distinction will be useful when defining a "general purpose" subset of events to monitor.

- *Intrinsic events* are events that are part of the standard set of events monitored by a monitoring tool. Their detection is automatic.

  Example :
  - `send a message`
  - `enter any procedure`

- *User specified events* are events that are specified by the user before the monitoring starts and their detection, therefore, is not automatic.

  Example :
  - `modification of the value of the variable ''counter''`
  - `enter the procedure ''Hello_world''`

  Currently, tools that allow for the specification and the automatic detection of user specified events are being developed [LCSM92].

Figure 2 summarizes the preceding discussion by classifying the different classes according to their level of abstraction.

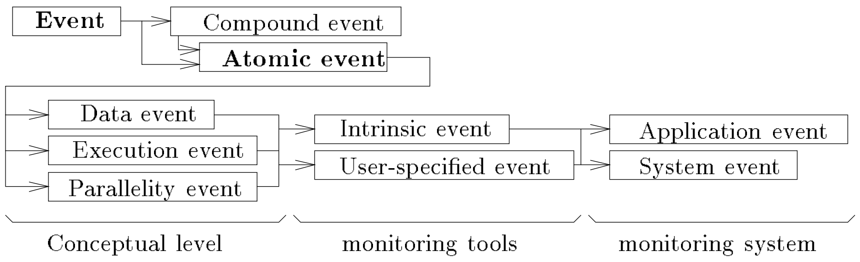

Figure 2: Relationship between event types and abstraction levels.

2.8 Definition of a "general purpose" set of events to monitor

In this section we will describe a set of events that could be considered as a "general purpose" set of events to monitor. It should not be very difficult for the reader to adapt this set to his own specific needs. Other authors have defined similar sets of important events to monitor [GHPW90, GMGK84, MN90] and in defining a "general purpose set" we drew it on their work, our on work and on discussions with users of parallel systems.

A general monitoring tool should both possess a predefined set of monitorable events and the possibility of user-specified events. The predefined set allows the user to start monitoring the execution of his programs right away. Defining user-specified events on the other hand allows the adaptation of the monitoring system to the specific needs of the user, thus increasing the power of the tool.

Basically three orientations can be distinguished in monitoring. Each of these orientations requires different types of events to be monitored :

- *Description oriented tools* are tools that concentrate on providing the user with insight into the behavior of a distributed program. They usually intend to help for a better understanding of the program and how it fit a given architecture or for teaching purpose. They usually collect information about parallelity events in a qualitative way (sending and receiving of messages between pairs of nodes, broadcasts, ...) and about the global flow of control in the processes (entering procedures).

  Examples of visualization oriented tools are TVIEW (see section 6.2) from the TIPS environment, VISTOP (see section 6.4) from the TOPSYS environment, MARITXU 6.7.

- *Debugging oriented tools* collect information about the state of processes (values of variables) and about the flow of control in a detailed way (entering specific procedures, loops, conditional statements, ...) [Noe92].

  Whereas a debugging oriented monitoring tool only gathers information about a program execution a real parallel debugger also allows the user to interact with this execution. In this report, we will concentrate on the monitoring aspects of debugging. For more information about parallel debuggers, the reader should refer to the more specialized publications [PU89, Pan92, LS92a, Cha91].

  An example of a debugging oriented monitoring tool is DETOP [BBB+90] of the TOPSYS environment.

- *Performance monitoring oriented tools* tend to gather statistical information (busy and idle times of processors, communication times, amount of messages exchanged, average lengths of messages, etc.) about an execution. This information is important for the user to understand and improve the performance of his programs.

  Examples of a performance oriented tool are PATOP (section 6.4) environment, ParaGraph (section 6.1), Pablo (section 6.5).

The idea of having one "general purpose" set of monitored events that can be used for different usages is becoming more popular with the development of complete programming environments rather then specific tools. The TOPSYS environment is an example of such a system, where one common monitoring system generates runtime information that is used for many different purposes [BL92].

In our set, we will only include atomic events. This is not a restriction since the occurrence of compound events can always be deduced by analyzing the atomic events (see SWM [BB88]). Atomic events have the advantage that their detection is relatively easy and often machine independent.

- **Monitoring the control flow** : First of all, we should be able to monitor the flow of control in each of the processors. To this purpose, we include the execution events below in our set. Some of them are related to the user defined procedures and library calls, others correspond to the OS control flow. Notice that with primitive OS will not have the multi-processing ability. Following our definition of events, the introduction of monitoring in the system will change its state, thus events corresponding to the use of reactive processes are also define. They will provide information about the overhead of the monitoring for instance.

  - **application events** : `trace_start` (beginning of the monitoring), `trace_end` (end of the monitoring), `trace_transport` (transport of event-records to the outside of the parallel machine for treatment), `procedure_begin` (a process enters a procedure, `procedure end` (a process quits a procedure), `user_defined_control_flow_event` (arrival at a user specified program instruction or control flow event).

- **system events** : `system_call_begin` (entry to a system call), `system_call_end` (exit from a system call), `process_start` (start of a process), `process_suspend` (suspension of a process by the scheduler), `process_activate` (activation of a suspended process), `process_end` (end of a process).

- **Monitoring the data flow** : To monitor the data flow, we will include the detection of a change of value of a "simple variable"[4] : `data_assignment` (this is an application event).

  No extensive data flow monitoring facilities will be included because this is typically the domain of parallel debugging.

- **Monitoring the parallelity events** : Monitoring parallelism includes two important activities, observing what is going on and recording statistical information for performance analysis. The list below enumerates the parallelity events that will be included in our set. Note that information such as network contention or buffer fill in could be deduce from these basic events and their corresponding attributes such as counters. As they are architecture and OS dependent, they should not appear in such a general set.

  - **application events** : `send_begin` (beginning of a send), `send_end` (end of a send), `receive_begin` (beginning of a receive), `receive_end` (end of a receive);
  - **system events** : `sending` (beginning of the actual sending after the rendezvous has been established (for synchronous communications only). The difference between the moments of occurrence of `send_begin` and `sending` is the interval that the process waited for the communication to happen (for example establishment of a rendezvous or interruption coming from the communication device.), `receiving` (beginning of the receiving)

2.9 Definition of the event attributes that need to be recorded

Once an event has occurred and has been detected, some information should be provided to the user of the monitor. This information is store in a record associated to the event and each information is called an attribute. The *event-record* will contain the *attributes* about the event that just occurred.

Obviously, the *attributes* that are recorded in an event-record depend both on the nature of the event and the subsequent use that is made of the event-record. Although it may not be obvious to define all the attributes an event-record should record, some common attributes can be defined (see table 1). Additional common attributes of communication events are represented in table 2.

`event_type`	this attribute encodes the nature of the event that occurred.
`timestamp`	is a number which uniquely identifies the moment when the event occurred in a processor.
`node_id`	defines the node where the event took place.
`process_identification`	defines the process where the event took place. On machines that only allow one process per node, this identification could be omitted.

Table 1: some common attributes

A further attribute that records the duration of the activity can be associated with compound event (see section 5.2). Many tools use it implicitly, by associating *duration-attribute* with an event which, in fact, is a compound event (PICL is an example [GHPW90].).

[4] Assignments without side-effects are implied here. By "simple variable", we mean a variable that is of a simple type (int, real, boolean, char, float) and that is easy to access. Pointers to simple variables are not allowed. The reason for this choice is related to the implementation of the detection of data events.

`destination_process`	the destination process of a message in send events.
`destination_node`	the node on which the destination process runs.
`source_process`	the origin of a message in receive events.
`source_node describes`	the node on which the sending process runs.
`message_type`	the type of message that is sent or received.
`message_length`	the length of the sent or received message.

Table 2: Additional common attributes of communication events

2.10 Time and the ordering of events

Time and the ordering of events are two crucial notions in monitoring. Measuring time intervals forms the base for the assessment of the elapse time of a program or system. Remark that even if several kind of performance metrics can be used by specific application (i.e. memory consumption, cache misses, shared object requests, etc.), the most common one is related to the elapse time taken by the application to end. The ordering of events is used to establish possible causality relationships which are needed to verify the correctness of a program. Time is a stronger notion than order, since if the time of occurrence of every event is known, the order of the events can be deduced.

Due to the presence of multiple independent concurrent processes in a distributed program, existing sequential notions of time and event ordering are not directly applicable.

- In most machines, time is defined by a local clock, just like in sequential machines. To relate two events that occurred at different nodes however, a "global" notion of time is required. This gives rise to a problem, because two independent local clocks do not necessarily indicate the same time.
- In a distributed program, events within the same process are ordered just like in a sequential program. When ordering the events from different processes, on the other hand, often only a partial order can be established and many different partial orders might exist that may lead to the same valid result.

2.10.1 Theoretical aspects of time and the ordering of events

The common notion of time is sometime not necessary for the tool that use the monitored information (debugger for instance). Thus, the causal order is sometime sufficient and less costly in terms of monitoring power. For this purpose, in this section, the notions "happened before" and "clock" will be introduced and the related issues will be discussed.

- In [Lam78] the *happened before* ("$\rightarrow$") relationship is introduced. This relationship is at the base of partial orders over sets of events. The "happened before" relationship only defines which events might have influenced each other, but does not tell us if they actually did. Thus, two events are said to be concurrent, if $\neg(e_1 \rightarrow e_2)$ and $\neg(e_2 \rightarrow e_1)$, where $\neg$ indicates negation.
- From an abstract point of view, a *clock* is just a way of assigning a number to an event. The entire system of clocks is represented by the function C, which assigns to any event e the number $C(e)$. For a clock to be correct, the following condition should hold [Lam78] :

  For any two events a, b : if $a \rightarrow b$ then $C(a) < C(b)$, but the opposite is not necessarily true ($C(a) < C(b) \not\Rightarrow a \rightarrow b$).

  Two different types of clocks can be distinguished. A logical clock is a clock that is incremented after every event occurrence independently of the elapsed time. A physical clock on the other hand is defined by $\frac{d(C(e))}{dt} = 1$. With a logical clock we can only determine whether an event took place before or after another event, whereas with a physical clock we can also define how much time elapsed between two events. Physical time is thus a more general notion than logical time, and given a physical time scale for all events in an execution we can deduce a logical time scale, but not vice versa.
- Due to the discrete nature of real physical clocks there is a quantification error between the real time and the clock of at most a $\frac{1}{2}$ tick. If two non-concurrent events occur within one tick, they will be

assigned the same clock value, and will thus be considered as concurrent. The probability of such an error will decrease by decreasing the length of a tick, but will never be zero. With a logical clock such an error is impossible, because the clock value is increased between every two non-concurrent events, no matter how small the physical time-interval that separates them.

- A disadvantage of the use of global clocks (logical or physical) is that the partially ordered set of events is implicitly mapped onto a totally ordered set. This results in a number of problems [Fid88]. First of all, it is impossible to distinguish between when the temporal ordering between two events is, and when is not enforced by a causal relationship (see Figure 8). Secondly, the monitor itself may be non-deterministically affected by this interleaving of events. For exactly the same computation with the same inputs which generate the same results, the monitor may produce different results due to the differences in interleaving.

  In [Fid88] the use of *partially ordered logical clocks* is proposed as a solution to this problem and a debugging strategy based on these partially ordered clocks is proposed [Mat89, SM92, DJ91].

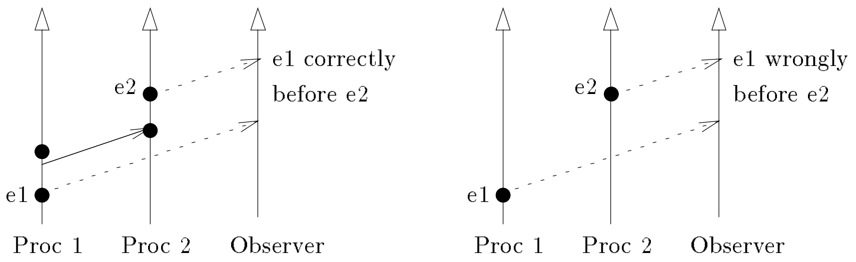

Figure 3: The effect of totally ordering a partially ordered set of events. The plain arrows represent communications between processors. With this total ordering, the observer is thus wrong from the causal relationship point of view.

2.10.2 Practical implementations

[IM93] shows that the global time precision can influence the understanding of the trace based representation.

Despite the fact that enforcing a total order in a set of events theoreticaly results in a loss of information, this method has been generally adopted because it is practically a very good way to compute a lot of metrics related to elapse time. All the tools that we have studied are based on the use of a global physical time (or wall clock), which not only allows the ordering of the events, but also the measurement of time intervals. Three different approaches can be distinguished to establish global time :

Using a global clock A *global clock* is the most accurate way to establish a global time. Unfortunately few machines are equipped with such a clock and adding one to an existing machine can be a complicated task (see Hypermon in [MRR90, MR90]).

Global clocks require additional hardware and are usually not portable. Either one central timer counter transmits its value to all the nodes in the system or the local clocks of the system works in a lockstep fashion. If the time tick becomes smaller than the traveling time of the clock signal, it is possible for concurrent events to be assigned a different timestamp. A larger time tick eliminates this problem, but increases the quantification error, and some compromise needs to be adopted.

Simulating a global clock Synchronizing the local clocks at the beginning of a program execution allows the simulation of a global clock in systems that are not equipped with a global clock. One problem is that the local clocks may drift apart. To solve this problem, resynchronizations may be necessary at regular intervals.

Hardware synchronization

An electrical signal is used to reset the local clocks in the machine. Compared to a complete global clock, this approach has the advantage of requiring less additional hardware, but the precision of the synchronization will be limited by the traveling time of the reset signal, as it is with a global clock. One disadvantage is that this reset mechanism usually needs to be added to the machine. Another is that the drift has not been eliminated.

Software synchronization

In [Lam78] Lamport proposes an algorithm that allows for software synchronization of the local clocks in a system. Each message that is sent from a node contains a timestamp of the local time when it was sent. In [DHHB86] a better algorithm is described that was originally proposed by Gusella and Zatti [GZ84]. The main idea of the algorithm is to exchange messages containing timestamps and current estimates of the time offset.

Practical implementations, as the PICL library on the Intel iPSC, used very simple solutions to synchronize the clocks. Pairs of nodes synchronized with each other through exchanges of messages that contained the local time of the sending node. Currently, a much better algorithm has been adopted in PICL [Dun91], it eliminates 99% of the drift between the different clocks, but it takes approximatively 60 seconds to synchronize.

Estimating global time from event-records

In this method the global time is calculated in a post-mortem way. Advantages are that this method is entirely portable, does not require any special hardware, and does not induce any additional overhead in the observed system.

During runtime event-records are generated that contain at least the origin of the trace and a local timestamp. Analyzing the send-receive pairs of our trace-file will thus result in a set of equations that bounds the values of the offset. The more pairs analyzed in this method, the better the obtained precision is. In [DHHB86] two methods based on this principle are proposed. Simulation results show that the global time can be estimated with high precision using these methods. The above principle could also be used for the on-line estimation of global time.

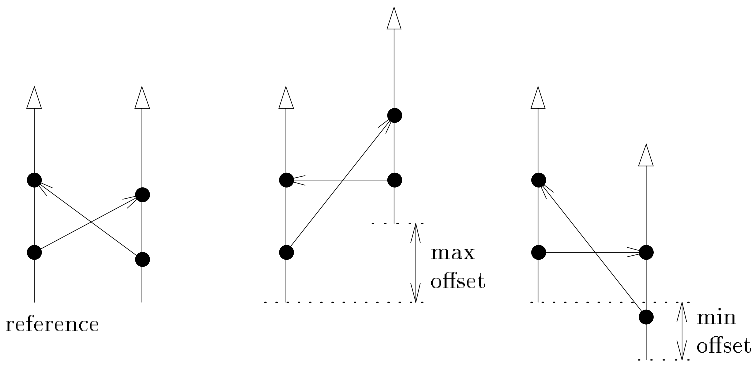

Figure 4: The bounds on the offset between two local clocks

2.11 Defining a global monitoring model

When observing a parallel system, the activity of gathering and using runtime information can be split into 3 reasonably independent phases (see Figure 5) :

1. The runtime information needs to be generated. This involves observing the system and generating the required information.

2. Once the information has been generated it needs to be stored and transported to the location where it is used.

3. The information that has arrived at its destination needs to be interpreted and used for the purpose for which it was created.

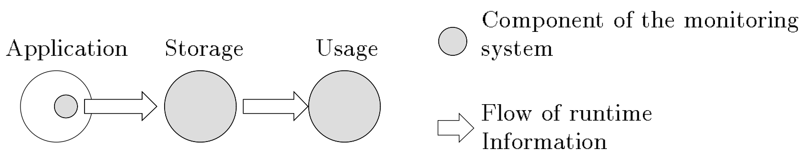

Figure 5: The three phases in gathering and using runtime information

This way of looking at the process of gathering and using runtime information has the advantage of allowing us to consider together many of the problems encountered separately. Most existing tools and programming environments, such as PICL/ParaGraph, TIPS, SIMPLE, TOPSYS, Pablo, GPMS, PIMSY, IPS/2, ... adopt this phase approach in their structure (see Chapter 6).

Each of these three phases will be presented in sections 4 and 5, after the next section which explains the event-driven approach.

3 Generation of the events

3.1 Description of the possible event-record formats

In the first place, event-records are a means of information storage for the event-attributes. The event-records are usually called *traces* because they represent the information let by the execution of the program.

For a trace-format to be valid, it is sufficient that the information can be encoded in and decoded from the event-record. Many other considerations, however, intervene when deciding upon the usefulness of a trace-format. Examples of these considerations are compactness, ease of encoding, and portability.

3.1.1 Encoding a trace-format

Mainly two classes of different encoding schemes can be distinguished:

- *Text based trace-formats* encode the event-records as sequences of ASCII characters. ASCII traces are human-readable and highly portable, but are time-consuming to encode. When encoding traces in a verbose manner, text based trace-formats are also space-inefficient. However, this is not necessarily true, when trace-records are encoded as sequences of integers represented in ASCII.

  Examples of monitoring systems that use text based trace-formats are the ParMod runtime system [Abs90] and the PICL instrumented library [GHPW90]. To limit the overhead of the encoding of the trace-records, PICL uses an internal binary trace-format because it is more time efficient.

- *Binary trace formats*, on the other hand, are compact and induce little overhead, because they are fast to encode. Unfortunately, byte orderings and number representations differ across different machines, and porting trace files across machines will almost certainly imply converting the trace files. According to [Hon89], the raw number of bits required to represent a event-record in text format can easily exceed five to ten times the number of bits required for an equivalent binary representation.

  In [RP91], a monitoring system that uses a binary trace-format is described.

3.1.2 Defining a trace-format

A *trace-format* defines both the syntax and the semantics of event-records (That is, what the event-records look like and what information is contained in them). Because of the growing interest in monitoring parallel program executions, some efforts are being made to standardize trace formats [PGUB92, MN90]. With a standardized trace-format, monitors and tools can freely exchange runtime information.

When defining a standard trace-format, problems arise at two levels :

- the definition of a **common syntax** is difficult because of the different usages that are made of event-records.
- defining **common semantics** may be even more difficult than defining a common syntax for two reasons. The field of parallel monitoring is just beginning to be explored, and important changes continue to be made on the target machines. Depending on the purpose of the tool that uses the runtime information, different events will be interesting to monitor.

Three interesting approaches to standardizing the runtime information exist :

- *Self-defining trace-formats* might provide an interesting solution to the standardization of trace-formats. A self-defining trace-format is a trace-format for which only the format of the header has been defined. This header contains both the syntactic and the semantic description of the event-records contained in a trace file. A tool reads this header to find out what information is contained in the trace file and where to find it [PGUB92, RP91, Ayd93].
- *object oriented* : Instead of defining a trace-format standard or just a header, the measured trace-formats are considered as a generic abstract data structure. Evaluation tools can only access the measured data via a uniform and standardized set of generic procedures. In order to be able to access and to decode the different parts of the measured data, these procedures use a so-called access-key file, which contains a complete description of formats and properties of the measured data. Tools that help the user in generating this access-key are available. In the SIMPLE environment [Moh90], this approach has been adopted.
- *standard exchange of Trace-Format (STIF)*[Hon89, MN90] : This format is designed to be easy to convert to and from and to transport between sites. It is text-based and most existing event-traces can be expressed in it. Although not intended as a general trace-format, the STIF format can be directly generated and used by tools that can handle the overhead of a text-based format.

3.2 Classification of the different approaches to monitoring

Among monitoring tools commonly used, three different types are distinguished : software, hardware and hybrid monitors. The distinction between these types of monitors is based on practical criteria, such as the amount of additional hardware and the intrusiveness of the monitor on the observed system.

We called $R_{\mathcal{A}}$ the whole resources used by the monitored active processes and $R_{\mathcal{R}}$ the whole resources used by the reactive processes in the parallel system. Examples of resources are processors, communication channels and memory. The different monitoring tools can now be classified according to the relation that holds between $R_{\mathcal{A}}$ and $R_{\mathcal{R}}$.

3.2.1 $R_{\mathcal{A}} \cap R_{\mathcal{R}} = \emptyset$: Hardware monitoring

Hardware monitors consist of completely independent monitoring systems that use physical probes to collect information from the electrical signals of the system. They require a considerable amount of additional hardware, but have the advantage of being non-intrusive. Due to their extensive use of dedicated hardware, they form the least portable class of monitors. Hardware monitors provide very low-level information about the execution and complicated mechanisms are required to generate application-level runtime information.

3.2.2 $R_{\mathcal{A}} = R_{\mathcal{R}}$: Software monitoring

Software monitors are implemented in software and require no additional hardware. This makes software monitors the most flexible and portable type of monitor. Since both the active and the reactive processes run on the same nodes, there is, however, a non-negligible impact on the monitored system and the software probes can change the behavior of the program. Software monitors provide easily understandable application-level runtime information.

3.2.3 $(R_{\mathcal{A}} \cap R_{\mathcal{R}} \neq \emptyset) \cap (R_{\mathcal{A}} \neq R_{\mathcal{R}})$: Hybrid monitoring

Hybrid monitors try to combine the best of both worlds, and are more flexible than hardware monitors and less intrusive than software monitors. Like hardware monitors, hybrid monitors usually require additional hardware and are often far from portable. However, like software monitoring, hybrid monitors usually provide application-level runtime information.

The advantage of this formalized classification is that it is based on our formal monitoring model and not on vague machine characteristics. To illustrate this, let us take a closer look at TMON (see section 6.2). TMON is defined as a software tool by its authors, since it does not require any additional hardware. However, since TMON requires the use of one system node for monitoring purposes, the overhead induced by TMON should be less than a pure software monitor. TMON could thus be considered as a hybrid monitor, because of the use of dedicated hardware (even if this hardware belongs to the machine). The above classification confirms this idea, since for TMON $R_{\mathcal{A}} \neq R_{\mathcal{R}}$.

3.3 Generation through software monitoring

3.3.1 The detection of an event

For the detection of events, small *probes*, i.e. small pieces of code, are inserted into the program code. Each time the execution reaches a monitored event, i.e. an instruction for which a probe has been inserted, this probe is executed. When executed, the software probe calls an "action"-routine (see 2.6) that handles the trace-generation.

To be fully accurate, it is not the execution of the monitored instruction itself that triggers the "action," but the execution of the probe. Formally speaking, the monitored event is thus the execution of the probe and not the execution of the monitored instruction. However, due to the neighborhood-relationship that holds between a probe and the instruction that it monitors (a probe is always inserted next to the instruction it monitors), one can usually conclude that the monitored event occurred as well. In critical cases, however, this difference is of crucial importance. If a probe is inserted before the monitored instruction, the generated trace might suggest that the instruction has been executed, whereas only the probe has been executed. In the opposite case, a monitored instruction could be executed without the generation of a trace. To avoid these problems, two probes could be used, one before the monitored instruction and one after. Only the execution of both probes would indicate a correct execution of the monitored instruction. Although this method ensures the occurrence of the monitored event, it nearly doubles the monitoring overhead. This overhead can be lowered by a distributed processing step that reduces the two traces to one trace. We know of no other implementation of double probes to ensure the actual occurrence of a monitored event.

3.3.2 The instrumentation of a program

The process of inserting software probes into program code is called instrumenting the code. We will discuss the most common approaches to code instrumentation.

Source-code instrumentation

With *source-code instrumentation* probes are inserted into the source-code of the application. Since the inserted probes are part of the application code just like the normal code, probes can be inserted at any point in the program. The monitoring system thus has access to any application-event of the application processes. This flexibility makes source-code instrumentation a very powerful monitoring method. System events can be detected as long as they are the direct cause of an application event and their effect is visible to the application.

Inserting the probes into the source-code can be done manually or automatically. One advantage of insertion by hand is that probes can be inserted anywhere in the source code (thus allowing the monitoring of any application-event); thus the user can adapt the set of monitored events fully to his needs. On the other hand manually inserting probes is very labour intensive and for this reason automatic instrumentation tools have been developed. The SIMPLE environment provides such an *automatic instrumentation* tool, called AICOS (Automatic Instrumentation of C Object Software) [Moh90] that automatically instruments procedures, procedure calls or arbitrary statements in source-code written in C. Another project [LCSM92] goes even further and aims to develop a complete *event-specification language*. The user defines the events that are to be monitored in this language and a "compiler" will automatically insert the probes in the source-code. The events that are currently monitorable are data-flow events such as assignments and incrementations.

A major advantage of source-code instrumentation is that the monitoring environment becomes completely portable as far as the monitoring of application events is concerned. A disadvantage is that each modification of the probes implies the recompilation of parts of the source code.

Instrumented-libraries

Instrumented-libraries have been developed to provide a solution to the labour-intensiveness of the source-code instrumentation process. An instrumented library provides the user with different types of primitives, such as communication primitives, that can be used in an application program. In the source-code of the library, software probes have been inserted. The PICL library is an example of a monitoring system based on instrumented libraries (see 6.1), although it also provides the user with the possibility of detecting user-defined events by inserting probes by hand.

To monitor a program using instrumented libraries, the user writes his program using the library primitives and links the instrumented library to the program at compile time. When running the program, the library routines will execute the probes in these routines thus generating the events. A disadvantage of instrumented libraries is that only those events that are generated by the library can be detected. To overcome this problem, a special trace-function could be created that is called by the application program each time an event needs to be detected. These calls, however, have to be inserted manually into the source-code.

Instrumented libraries provide a high-level of portability and allow the direct use of a monitoring system if the program has been written using the primitives of the library. When the monitoring is no longer required, only a linking step is required (to link in the un-instrumented library) instead a complete recompilation.

Object-code instrumentation

For *object-code instrumentation* a special *instrumenting compiler* is required. Such a compiler inserts the software probes in the code at compile time. Software probes can be either inserted in the object-code or in some intermediate representation.

Object-code instrumentation has the advantage of being completely transparent to the user and, like source-code instrumentation, provides access to all application-level events. It is likely to induce less overhead than software-instrumentation because the software probes consist of machine instructions instead of high-level language instructions that need to be compiled first.

Object-code instrumentation has the disadvantage of requiring an instrumenting compiler. At this time, we know of no such compilers that are commercially available for parallel machines. Thus, an existing compiler will have to be modified.

In [MAA+89, MRR90] a performance monitor based on object-code instrumentation is described. To insert the software probes a GNU C compiler was modified. This modified compiler inserts calls to the monitoring functions in the intermediate RTL representation instead of in the assembly code. The advantage is that the instrumentation is machine-independent, which is not the case in real object-code instrumentation. With compiler options, the user can control the instrumentation process.

An instrumented kernel

Inserting the probes (called *hooks* in this context) into the code of the system kernel of a node makes the monitoring completely transparent to the application program. Each time the application calls a kernel function, such as send or receive, the software probe is executed and the event is detected.

Equipping a kernel with software probes is the only way to detect most system events using software monitoring. Detecting application-events on the other hand, will be impossible as long as the application does not use any kernel functions. To overcome this problem, a solution similar to instrumented libraries can be used, that is, a special trace-function that can be called upon and added to the kernel (see [mL92]).

Today, few machines offer kernels that are equipped for monitoring (see [MN90] and section 2.8 for a discussion about the minimum set of measurement capabilities a parallel operating systems should provide). On most machines instrumenting a kernel will be a far from easy task and might even necessitate completely rewriting the kernel or writing a new kernel. This makes an instrumented kernel the least portable and the least flexible type of software monitoring.

The Crystal project [RR89b] and TOPSYS (see section 6.4) have adopted the instrumented-kernel approach. In Crystal, that runs on an iPSC/2, the original NX operating system has been modified by adding software probes to the NX source code. In the TOPSYS environment on the other hand, the probes have been added to MMK, the operating system kernel which is part of the project and which runs on top of NX.

In [MAA+89] the instrumented kernel approach has been adopted as a complement to the object-code instrumentation. The instrumented kernel detects the system events, whereas the instrumented object-code handles the application events.

Some other monitoring systems have modified the Operating system : [RR89a] for the Ncube OS, [vRT92c], for TROLLIUS.

3.3.3 A complete software monitor

A complete monitoring system not only detects events, but also generates, stores, processes (if there is a form of distributed processing), and transports the event-records.

The way these functions are implemented depends on whether the monitored parallel system allows just one or multiple processes per node. The "one-process-per-node" systems clearly offer the least flexibility and will be discussed first.

- *"One-process-per-node" systems* : In such systems, the application and the monitoring system belong to the same "hybrid"-process. This process executes the application code, but provides at the same time the monitoring functions (for instance, and application linked with an instrumented library). The monitoring routines can do more than simply generate an event-record, through an underlying system of global variables and calls to monitor-routines the whole functionality of the monitor can be implemented.

  The monitoring system can be implemented using any of the previously described methods of instrumenting code. The PICL/ParaGraph tool is an example of a monitoring system for a "one-process-per-node"-system that uses instrumented libraries to implement the monitoring system.

- *"Multiple-processes-per-node" systems* : With these systems many different implementations of a monitoring system could be devised. Only one, which we will call "specialized processes," will be discussed here.

  In this approach, the tasks of the reactive process $r \in \mathcal{R}$ are distributed among several specialized processes that work together in providing the local monitor functions. Probes are still responsible for the trace generation, but the storage, eventual processing, and transportation of the runtime information is performed by one or more independent reactive processes. One set of specialized processes could be provided per node, or each monitored active process could have its own set of specialized reactive processes.

"Specialized processes" allow a cleaner system implementation by separating the active and the reactive system. Only the trace-generation remains in the active process, but as we have seen this is unavoidable in software monitoring. Another advantage of this approach is that the user can issue commands to the reactive processes, thus changing on the fly the way the monitor behaves. This is a great flexibility advantage, because the monitoring load can be adjust during execution, regarding the needs of the users. The only drawback is that complex OS cost a lot in context switches and thus diminish the overall performance of the system and precision of the probes.

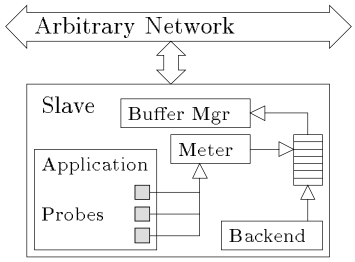

Figure 6: The different specialized processes in the TIPS environment

The TIPS programming environment [WCG+92] and GPMS [vRT92a] implement this "specialized processes" model. Three different processes can be distinguished in TIPS (see Figure 6) : a meter-process that collects the generated traces and stores them in a buffer, a buffer-manager that takes care of the transportation of the event-records, and a back-end process that implements some additional features.

3.3.4 The intrusiveness of software monitoring

Software monitoring of the execution of parallel programs requires the insertion of reactive code into the source code of an application program. When executed this results in a "hybrid-process" that alternatively executes *active code* (application) and *reactive code* (see Figure 7 and section 6.2.3). Since these instructions are not inserted according to a regular pattern, the execution of the active code is perturbed by altering, in an arbitrary manner, the timing of events in the multiple threads of control being monitored.

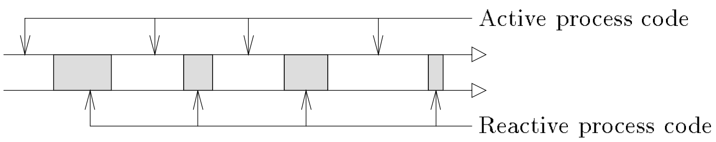

Figure 7: The code of the active and the reactive process mixed into one thread of control

This is equally valid in "multiple-processes-per-node" systems. Although the code is clearly separated in distinct processes (except for the probes themselves), the active and the reactive code are still running on the same processor (in a time-sliced way).

The altering of the timing of events due to the insertion of reactive code may [MLCS92]:

- lead to incorrect results
- create (or mask) deadlock situations when the order of events in different threads of control is affected
- cause a real-time program to fail to meet its deadlines
- increase the execution time of the monitored program

The existence of these effects on the behavior of a monitored system is generally recognized. However, no precise model has been developed that allows us to quantify these effects. A common approach is to try to minimize the influence of the software monitoring and neglect whatever influence remains.

In the PIE environment [LSV+89] a simple compensation mechanism has been developed that attempts to compensate the delays induced by the software probes. This algorithm that is used on uni-processor, multi-threaded executions, adjusts the timestamps of each event-record in accordance with the number of monitored events that occurred before it. A risk of reordering of the events using this simple compensation algorithm exists on multiprocessors. Therefore, the PIE project is currently examining other algorithms that could be used on multiprocessors.

3.4 Generation through hardware monitoring

3.4.1 Hits and high-level events

The classes of events introduced in section 2.7 were either atomic events or compound events consisting of several atomic events. These events can be easily related to the executing application and are considered as *high-level events*. At the machine-level an atomic event is composed of many *machine-level events* (see Figure 8). These events are also called *low-level events* and correspond to the execution of machine-instructions. The detection of a low-level event is called a *hit*. Just as the detection of a compound event required the detection of several atomic events, several hits are needed to detect a high-level event.

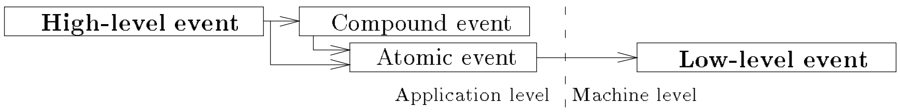

Figure 8: The different levels of events

3.4.2 The detection of an event

To detect event occurrences hardware monitors use *physical probes* that are connected to the electrical signals of the system. The on-chip integration of floating point, memory management and other functional units and the use of pipelining and on-chip caches have led to a hardware observation of the node-processors. Therefore, hardware monitors will probably be integrated on the chip in the future [BLT90]. Since much of the information is buried in the chip, often a special version of the node processor is required that provides access to signals in the chip. Figure 13 shows the integration of the TOPSYS hardware monitor into an observed system. The TOPSYS monitor makes use of a special processor version,which has extra signals added to its pins (the bondout lines in the Figure 9) [BLT90].

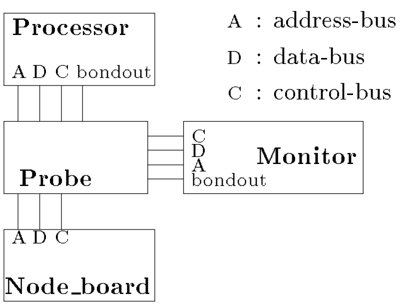

Figure 9: The integration of the TOPSYS hardware monitor in a node

In hardware monitoring, events are detected by matching patterns on the observed lines with predefined patterns. Banks of high-speed comparators are used for this matching. However, by observing the evolution of the busses and other electrical signals, only machine-level events can be detected. The user is not interested in this level of abstraction, because the relation between hits and high-level atomic-events is far from obvious. A hardware monitor should thus be able to detect those sequences of hits that constitute an interesting high-level event. For this purpose, hardware monitors are often equipped with a state-machine that recombines the hits into high-level event occurrences. When a defined sequence of hits, i.e. a high-level event occurrence, is detected, the hardware monitor generates an event-record.

From the above discussion, it may be obvious that the detection of high-level events is a very complicated matter. This is true not only because a considerable amount of hardware is needed, but also because of the different levels of abstraction that exist between the executing machine instructions and the source-code. To map application objects (instructions, data-structures, etc.) to machine-instructions and memory-locations, the hardware monitor should be equipped with a special mechanism that ensures this mapping.

3.4.3 A complete hardware monitor

To complete our hardware monitor, additional hardware is required for event-record storage, eventual processing and the transportation of the event-records to the external storage. A hardware monitor thus consists of three different parts: the probes that actually measure the electrical signals, independent local monitors that detect the low-level events and recombine these hits in high-level events and generate the event-records, and a separate communication network for the transportation of the event-records.

The availability in hardware monitors of a separate communication network constitutes a considerable advantage. Trace-records can be downloaded to the treatment place in a real-time fashion and distributed compound-events can be detected without interfering with the communication network of the observed system. Because of their independent communication network, hardware monitors can continue to function even if the observed system blocks at some point.

3.4.4 The intrusiveness of hardware monitoring

Hardware monitoring constitutes a completely non-intrusive method of monitoring a system. The only effect a hardware monitor may have on the observed system is the need for a decreased clock-speed [mL92], due to the additional loading of the system busses. This does not affect the ordering of the events, but only slows down the execution time.

3.5 Generation through hybrid monitoring

Hybrid monitoring tries to combine the best of both software and hardware monitoring. Since hybrid monitors are defined as those monitors that are neither completely hardware nor software monitors, many different types of hybrid monitors could be devised, ranging from those that are almost complete hardware monitors to those that are basically software monitors. The most common ideas in hybrid monitoring will be presented in this section.

3.5.1 The detection of an event

All of the hybrid monitors that we studied use software probes for event detection. This eliminates one of the main disadvantages of hardware monitoring, namely, the generation of low-level runtime information. A hybrid monitor offers the same level of abstraction and the same flexibility in event detection as software monitors. To instrument the code for hybrid monitoring, the same methods that are used for software monitoring can be used (see section 3.4).

3.5.2 The generation of the event-records

Two common approaches can be distinguished in the trace generation.

- The simplest solution is to have the software probes generate the complete event-record, as is done in software monitoring. This event-record is then sent to the hybrid monitor. One advantage of this approach is that all application-events are detectable and no additional hardware is required for the trace-generation.

  The efficiency of this approach depends largely on the mechanism used to communicate with the hardware part of the hybrid monitor. If a communication channel of the machine is used, it may be more efficient to use software monitoring that stores the event-records locally and downloads the event-records only after execution[5].

- In the second approach a minimum amount of information about an event-occurrence is written into a special part of the memory (or on a special communication channel). This part of the memory (or channel) is monitored by the hardware part of the hybrid monitor that detects the write and collects the written information (event type and probably some attributes). Based on this information, the monitor generates a complete event-record by adding further attributes (for example a time-stamp) and by converting the information to the right format.

  The advantage of this approach is that very little overhead (only one or a few simple write(s) or sends) is needed to detect a high-level event. If the hardware part of the monitor has access to the application's state, all the attributes can be collected by the monitor and the software probes have only to issue one single write to signal an event occurrence. This results in a minimal overhead.

  In [OQM91] the adaptation of the ZM4 monitor to a transputer network is described. The ZM4 monitor has been used with both a sort of memory mapped I/O[6] and direct channel I/O[7]. Whereas the overhead of the former approach is about 100ns (one assignment instruction), the overhead of the latter varies between 4μs and 200μs, depending on the number of processes on the observed node. The memory mapped I/O approach clearly induces less overhead, but requires considerably more additional hardware because the address and the data bus need to be monitored.

  The Hypermon hybrid monitor for the iPSC/2 implements a slightly different approach [MAA+89]. The hardware part of Hypermon is connected to the I/O-signals of each of the monitored node processors. The software probes in an application write small values on the I/O-lines of the processor that are detected by the monitor.

3.5.3 A complete hybrid monitor

To complete our monitoring system, provisions are needed that take care of the storage of the event-records, the eventual distributed processing, and the transportation. Often this is handled by additional hardware, just like in hardware monitors. However, this is not a requirement and many different hybrid monitors could be imagined that share, in various ways, the workload between the observed system and the monitoring hardware.

Most existing hybrid monitors consist of independent monitor-nodes with their own local resources. A monitor-node can monitor one or more system-nodes. Monitor-nodes and system-nodes can interact in various ways, such as (for example) through shared memory, I/O channels, or memory mapped I/O. The storage of the event-records is done locally, as is the distributed processing. For the transportation both a separate and the monitored-system's communication network could be used. A global description of a real hybrid monitor can be found in section 6.3.3.

3.5.4 "Specialized processors"

In this section, a new approach to hybrid monitoring that we call "specialized processors," will be introduced. The idea seems promising, but to our knowledge, it remains to be exploited.

[5] This assumes that the sending of a message takes more time than a memory access, and off-line monitoring is used.

[6] In this approach, a part of the memory addresses are reserved for monitoring uses. The monitor observes the address bus and reads the data from the bus that is written to a reserved address.

[7] In this approach, runtime information is sent over an unused communication channel. This channel is connected to the monitor which receives the runtime information.

"Specialized processors" is an approach to hybrid monitoring that requires no additional hardware. In this approach, the available nodes of a system are divided into two sets. One set of "active processors" runs the active processes and generates event-records using software probes, whereas the other set (of "reactive processors") runs only reactive processes and ensures the storage, the processing, and the transportation of the event-records. Notice that the size of these sets does not have to be the same, and moreover, that the monitoring processes can be on the same processor as computation processes on multi-processes per node machines. Figure 14 illustrates this approach with a three dimensional hypercube, in which one cube of dimension 2 monitors another cube of dimension 2 that does the actual processing. Many other configurations, however, can be conceived.

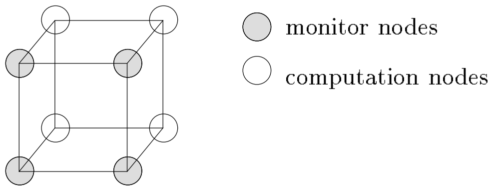

Figure 10: The "specialized processors" approach to hybrid monitoring

One of the strong points of "specialized processors," in comparison with other hybrid monitoring systems is that it requires no additional hardware. This property allows the construction of portable hybrid monitoring systems, that is, of course, as long as the target machine has enough nodes.

In our example of the hypercube, a further advantage is the separation of the network into "active communication channels" and "reactive communication channels". The traces are transported to the "reactive" nodes over the "reactive channels" and do not interfere with the "active communication network".

We think "specialized processors" will be especially useful in real-time monitoring for scalability reasons mentioned in 7. In off-line trace processing, storing the traces locally may result in less overhead because storing data in memory is usually faster than sending it over a channel.

3.5.5 The intrusiveness of a hybrid monitor

Hybrid monitors are intrusive, due to the use of software probes, but are usually less intrusive than software monitors.

The actual intrusiveness of a hybrid monitor depends largely on how much of its functionality has been implemented in additional hardware. A monitor such a TMON (see section 6.2), which mainly consists of a software monitor with one independent node, will be slightly less intrusive than its software equivalent. On the other hand, hybrid monitors such as the Hypermon [MAA+89], which are close to hardware monitors, hardly induce any overhead when used with memory mapped I/O (see section 3.5.2).

4 Transportation of the events

To our knowledge, no entirely distributed monitoring tool has yet been developed (i.e. a complete parallel program that will gather and use the monitored information). In the existing monitors, the runtime information is used at a different place than where it was generated. Therefore the event-records need to be transported. As we will see in 7, this implies the transportation of a considerable amount of data.

Unless a completely independent hardware monitor is used, this transport of data will affect the observed system at two levels : at the processor level where an extra load will be created due to the need of processes that control the transport, and, at the level of the communication network where the communication load will be increased. This is always true for software monitors, but may only be partly true for hybrid monitors since at a certain point they use hardware means.

To minimize the effect of this trace-data transport, different *transport-strategies* have been developed. The most common strategies will be described in the following sections. The aim is to allow the gathering of the monitored data to a certain point in the system with the least possible perturbation. Without that ability, the monitored information is not still valid since it could represent something very far from the application running without monitoring.

4.1 Immediate transport

Immediate transport strategies send the monitored data as soon as they are recorded. They are implemented in those monitors that need to deliver the runtime information in real-time (See [LS92b] on iPSC/2). According to the importance of the instantaneous availability of the information (i.e. the possibility of using it outside of the application), two strategies can be distinguished.

- *Real-time availability* : in critical applications where the occurrence of an event needs to be detected instantaneously the only viable approach is the use of a hardware monitor. As we discussed in section 3.3, software probes cannot provide the exact moment of occurrence of an event. Therefore, hardware probes that monitor the electrical signals of the monitored system need to be used.

  To ensure the immediate transport to the destination, a separate communication network is required.

  Since a hardware monitor is used, this strategy obviously has no impact on the monitored system.

- *Delayed availability* : this means that the event-records become available soon after the event occurred. There may, however, be some unpredictable communication delay before their availability. From a practical point of view, this strategy means that as soon as a event-record has been generated, it is sent to its destination. Typical examples that use "delayed availability" are visualization tools that allow the "real-time" observation of the executing program and *output debugging*[8] [LS92b].

  "Delayed availability" definitely has an impact on the behavior of the communication network of the system, if no separate network is used. With blocking sends and a loaded communication network, the impact may even become dramatic, due to the continuous blocking of the active processes by sends.

  This is one of the transport strategies that can be used by the TOPSYS monitoring system.

4.2 Store and unload progressively

"Store and unload progressively" strategies aim to minimize the impact of the trace-generation on the communication network. The main idea behind these strategies is that the unloading of event-records in blocks will economize the channel set-up times. The three most commonly encountered strategies are:

- *Unload when buffer full* : are the simplest strategies; the event-records are stored in a memory-buffer until the buffer fills up. When this happens, the active process is suspended and the event-records are downloaded. A disadvantage of this method is that an active process is suspended at unpredictable moments.

  "Unload when buffer full" is one of two transport strategies that have been implemented in the PICL library [GHPW90], the second being described in "store and unload afterwards". The user has the choice of either of the strategies.

- *Unload when requested* : This strategy allows the user to define the moments of unloading by calling a `trace-flush function`. This way, the suspension of the active process in critical sections, as might happen with the "unload when buffer full" strategy, can be avoided. In case a buffer fills up, the previous strategy is applied.

  In the PICL library [GHPW90], a trace-flush function has been provided that allows the user to unload the contents of the trace-buffer.

[8] Output debugging consists of inserting debugging probes, usually output statements at carefully selected places in a program. These probes are then used to try to understand the behavior of a program [CBM90].

- *Unload when low load* : Each time the load of the communication is less than a defined value, the reactive process will unload the event-records contained in its buffer. Obviously, the "unload when buffer full" strategy often complements this strategy.

  The "unload when low load" complemented by the "unload when buffer full" strategy has been adopted by the TMON monitor of the TIPS environment [WCG+92].

4.3 Store and unload afterwards

The "store and unload afterwards" strategy does not affect the behavior of the communication network. The event-records are stored in a local memory buffer and unloaded when all the active processes have finished their activities. A serious limit of this approach is the size of the allocated memory buffer. Once this buffer fills up, the generation of the traces is suspended.

Of all the strategies discussed the "store and unload afterwards" strategy has the least impact on the monitored system. There is no impact on the communication network and less overhead for the processor (assuming that storing data in memory takes less time than initiating the sending of this data).

"Store and unload afterwards" is the second transport strategy that has been implemented in the PICL library [GHPW90]. Furthermore, this strategy has been used in [Imr92] for the ECS[9].

5 Representation of the events

Until now we have mainly discussed the generation of the runtime information. In its raw form, this information consists of sequences of event-records. These records may be contained in one or more files or may arrive as continuous streams as is the case with real-time monitoring. Each event-record contains some attributes that are related to the event that was at its origin. Depending on the chosen trace-format the event-records may be human-readable or not. The timestamps may or may not be related to some global time and if a self-defining trace-format is used, it may not even be known beforehand what information will be recorded in the event-records.

Different tools exist that help the user in assessing the information contained in the event-records. These tools range from the elementary text-editor to an advanced programming environment.

In this section, we will present in a systematic manner the different approaches, methods, and tools that have been developed to deal with the runtime information. In spite of the considerable variety among these methods and tools, we will try to provide an integrated view by stressing the common points. First, we will discuss the building of a global view of the runtime information. Then, we will take a look at different mechanisms that reduce the amount of runtime information. In section 5.3 we will present some approaches to the analysis of the runtime information. In the last section, we will discuss different user-interfaces.

We will concentrate in this section on the processing itself since distributed versus centralized processing will be discussed in 7.

5.1 Building a global view of an execution

The original aim of collecting runtime information was to provide the user with a consistent global view of the monitored system. As we saw in section 3, the runtime information is generated as a set of sequences of event-records. Usually one sequence of local event-records is generated per node. In order to gain a global system view, these sequences have to be related to each other.

Although different approaches to relating local trace-sequences are conceivable, the studied tools simply merge them into one global trace-file. Therefore our discussion will be limited to the merging of the trace-sequences, but first some further concepts will be introduced.

Merging the different local event-traces into one global event-trace essentially means grouping the event-records from the different event-traces into one global event-trace and sorting this event-trace according to the global timestamps of the event-records. If the timestamps of the event-records are not related to a global time, this global time can be deduced from the local time stamps of the event-records. In Pablo, each node

[9] Edinburgh Concurrent Supercomputer

generates its own trace file, thus such an algorithm is used [Noe92] to merge data into one file. To our knowledge no tools use the concepts of partial trace-segments in their analysis of the run-time information.

5.2 Reducing the amount of runtime information

As we will see in 7, the amount of runtime information generated by an average-sized parallel system, can easily be enormous. A separate processing step that reduces this amount of information could considerably limit the amount of further processing needed. In those cases, where only a restricted view of the monitored system is required, this may be particularly useful.

The two different ways of reducing the runtime information will be described here in more detail.

5.2.1 Event-record filtering

Event-record filtering aims to eliminate the uninteresting event-records and generates a subset of the recorded trace-data. A simple selection scheme can be based on the event-type, on the attribute-values of an event, or on a combination of both, but more complicated schemes can also be devised.

Filtering of the events can be done at different levels.

- *Conditional event-record generation* generates only a limited set of event-records and the filtering takes place at the moment of event-occurrence. When an event is detected, a condition is verified and only when this condition is met is an event-record generated. Properly speaking, "conditional event-record generation" takes place when the tested condition depends on parameters external to the event, whereas in other cases simply the occurrence of a different, more specific event is monitored. PICL offers the possibility of "conditional event-record generation" by allowing the user to set the level of tracing required [GHPW90]. Depending on the level set, more and different event-records will be generated. An on-the-fly method is, also, explained in [HM93]. The PIE tools [LSV$^+$89] proposed also some filtering to reduce the amount of data.

- Filtering can also take place in a distributed manner, just after the event-record has been generated or before the transport of the event-record takes place. Since most filtering is based on information contained in the event-record itself there is no problem in distributing the filtering operation on each monitored processor, thus the filtering is done on a smaller amount of data and in parallel. Only in the case of inter-trace filtering-rules is extra communication required. Distributed filtering has the advantage of distributing the processing load among several processors and reducing the load of the communication network. In the on-line monitoring of large systems this type of filtering might be a necessity.

- The last moment at which the filtering can take place is just before the processing of the event-records. The existing tools that provide filtering possibilities have usually adopted this approach and preprocess the trace-records before starting the "real" processing.

5.2.2 Event-record clustering

In *event-record clustering* the event-records are scanned for the presence of compound events and activities and their attributes are calculated. Clustering thus provides the user with a higher level of abstraction and a often more compact representation of the monitored execution.

Event-record clustering can, like event-record filtering be done at different levels.

- The lowest level is found in hardware monitors that cluster the hits to detect application-level event-occurrences [BLT90].

- In non-hardware monitors, event-records can be clustered in a distributed manner, during or after the point at which the application process is terminated. This type of clustering has the same advantages as distributed filtering (see 5.2.1).

- Clustering the event-records just before the usage is the highest level where clustering can be done. The existing tools that provide clustering features have usually adopted this approach. In the SIMPLE environment [Moh90, Imr92] a tool called ADAR (Activity Definition And Recognition system) has been provided. ADAR reads an activity-definition file and processes the event-trace in order to find the defined activities and compute their attributes.

5.3 Analyzing the runtime information

The information contained in the event-records may not always be the information we are looking for. Trace-records only contain information about the occurrence of events or, at most, about activities, i.e. sequences of events (see 5.2.2). To gain a more global view of the observed system further analysis is often necessary to determine global parameters or to detect special conditions that may have occurred in the system.

Performance monitoring is a typical example where further processing of the runtime information is required to determine parameters such as average busy/idle time, total communication overhead or the number of context switches. Deadlock detection is an example of analyzing that could be used by debuggers.

5.3.1 Off-line versus on-line analysis

Off-line analysis is the easiest way to analyze runtime information. The analyzer simply reads the event-records that have been stored in a trace-file by the monitoring system. Since all the information about the execution is available at the moment of analysis, a global view of the execution can be easily constructed.

In on-line systems, only a part of the runtime information is available at the moment of analysis. Therefore global parameters cannot be computed and need to be estimated.

An approximation scheme based on interpolation is often used. With a few available event-records a rough initial estimation is made and each time new event-records become available this estimation is adjusted. The precision of this method usually converges to the same precision as off-line analysis. The on-line estimation of global time based on local time stamps, as described in section 2.10, is an example of the use of *on-line analysis* of the runtime information.

5.3.2 Low-level versus complex model based analysis

Many different approaches to the analysis of runtime information can be adopted depending on the use that one makes of the runtime information. These methods range from the simple compilation of statistics from event-records to complex systems such as tools based on Event Based Behavioral Abstraction (EBBA). Some of these methods will be briefly introduced here.

The simplest way of analyzing runtime information consists of collecting statistics from the individual event-records. ParaGraph [HE91b] and the Crystal [RR89b] are typical examples of tools that use this approach. Among the gathered statistics are cumulative busy/idle times, cumulative communication times, number of bytes sent, etc.

More complex analysis is provided by the integration into the monitoring environment of statistical analysis packages, that allow the interactive analysis of the trace-data. The SIMPLE environment adopted this approach by integrating the data analysis and graphics package S from AT&T [Moh90].

The *Event Based Behavioral Abstraction* (*EBBA*) approach is more than simply a way of analyzing event-records, and constitutes a complete high-level approach to debugging [Bat89]. Globally speaking the approach consists of constructing high-level models that describe the expected behavior of the monitored program. When executing a program, EBBA-based tools [Bat88] automatically compare the generated runtime information with the expected behavior as described in a model. The differences are analyzed and an interactive tool allows the user to explore these differences.

In [CBM90] the use of artificial intelligence techniques for the analysis of runtime information is suggested. A system based on this approach would capture the expertise of an experienced programmer in a knowledge base and make it available to all users by assisting the user in developing fault hypotheses. To our knowledge, no systems, for distributed-system runtime-information analysis that are based on AI techniques yet exist. However, in [CBM90] a system for the debugging of single processor, concurrent systems, called the Message_Trace_Analyzer [GS84] is mentioned.

5.4 Interacting with the runtime information

There are essentially two problems that the user faces when dealing with the runtime information:

1. Monitoring systems can generate enormous amounts of runtime information that are usually of a relatively low-level of abstraction. Separating the essential from the irrelevant may not always be an obvious matter.

2. The parallelism of the monitored program is implicitly contained in the event-records and is not very easy to conceptualize.

The user-interface forms the key element to the interaction with the run-time information and assists the user in dealing with the above problems. It determines to a large extent the power of a monitoring system.

In this section, we will start by describing the different representations that can be used to present the runtime information to the user. Next, the notion of time will be integrated in our representations and different aspects of this will be discussed. The last part will deal with complete user-interfaces and some useful features will be described.

5.4.1 Different representations of the runtime information

Different *views*[10] of the execution can be constructed by emphasizing or ignoring selected information. For each view, different mediums and different representations can be used to present the run-time information to the user. In this section, both the use of images and the use of sound as mediums to represent runtime information will be discussed.

Textual data representation

Textual data representations present the event-records to the user as plain text and constitute one of the most elementary ways to represent runtime information.

Advantages of textual data representations are that runtime information is easily converted to it and that no special devices are required to represent it. A disadvantage is that complex relationships are usually much harder to grasp when expressed in text than when expressed graphically.

Although the use of graphics has become the dominant way to represent runtime information, textual representations are still used, often in combination with graphical representations. An example of the use of text is the Jade programming environment [JLSU87], which provides amongst others a text-based console that prints messages as events are detected. In SIMPLE, a completely text-based representation tool SMART that can be used on ASCII terminals is provided (see section 6.3.3).

Graphical data representation

Graphical data representations[11] present details of views in the form of one, two or pseudo three-dimensional pictures, such as curves, diagrams, barcharts, pie graphs, meters, leds, contour plots and surface plots. Color or gray-scales are often used to increase the expressive power of visualizations.

Graphical representations have become the dominant way to represent runtime information. It is commonly admitted that graphical representations are more intuitive and easily understood by the user than textual representations. Furthermore, complex relationships can be easily expressed in graphical representations.

Essentially, two different types of graphical representations can be distinguished:

- graphical representations that represent information related to one object, such as the system as a whole, a processor, a process, a message channel, etc.
- graphical representations that represent interactions between different objects.

[10] A view describes what information is represented to the user, whereas a representation describes how it is presented. Visual representations are also called visualizations.

[11] Graphical data representations are also called *graphical visualizations* [LMCF92].

Representing information related to one object The information related to an object usually consists of some statistical information about that object. For example, the utilization of a processor, the number of messages that have been sent through a channel, the number of memory accesses, etc.

To represent this information, curves, diagrams, barcharts, pie graphs, meters, leds, contour plots and surface plots are used[12].

Representing interactions between objects In distributed programs objects interact essentially in two ways, by exchanging messages and by synchronizing their activities. Different representations have been adopted to represent these interactions and two of them will be described here[13]. The spacetime diagram is one the most commonly encountered representations, In a spacetime diagram (see Figure 11) each process is represented by an horizontal line. A slanted line between two process lines indicates the exchange of a message. The following refinements can be made to this representation. A dashed horizontal line can represent a process that is blocked by a send or a receive (see ParaGraph in section 6.1). Synchronizations can be represented by a vertical line that crosses each of the processes participating in the synchronization. A process blocked in a synchronization can also be represented by a dashed line.

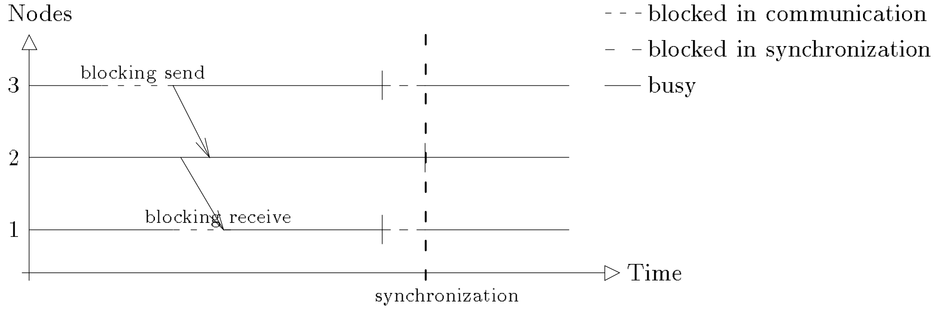

Figure 11: Example of a spacetime diagram

Sonic data representation : *Sound* represents a rich communication channel. The dimensions of the audio parameter space include duration (time), pitch (frequency), volume (amplitude), timbre (waveform) and stereo balance. In spite of the fact that psychological studies have shown that users respond more quickly to audio cues than to visual signals in certain instances [RP91], the possibilities of sound remain largely unexplored in monitoring tools.

The performance monitor that are currently under development implement *sonic data representation* are [RP91, FJA91, ZT92b].

The use of sound is based on sonic *widgets* that allow the mapping of sequences of scalar or higher order dimensional data values to frequencies, durations, timbres, attacks/decays and stereo balance. In addition to widgets, *earcons*[14], i.e. audio warnings (sampled voice warnings, enumerations, alarms and bells, etc...), will be integrated in the monitor.

Notice that all these representations can be time-line while associated with the corresponding filtering on timestamps.

5.4.2 Features of user-interfaces

Understanding the behavior of distributed programs is a complicated task. A user-interface should possess some additional features that help the user to interact with the different views on the data. In this section,

[12] For examples of these graphical representations see [Abs90, GGJ+89, HE91a, LMCF92, LSV+89, CK90, SBN88].

[13] For other representations that represent interactions between different object see section 6 and [Abs90, GGJ+89, HE91a, LMCF92, LSV+89, MAA+89, CK90, SBN88].

[14] As opposed to icons.

features of existing tools that we feel are particularly useful will be briefly presented. Such descriptions can be found in [Mil92].

Different levels of abstraction

When analyzing a program's execution, the focus of interest often begins with the entire program and then moves to smaller groups of processes or even to a single process. The collective behavior of the processes defines the results of the program, but the behavior of individual processes may be the cause of anomalies [LMCF92]. Therefore a monitoring tool should have the capabilility of presenting information at different *levels of abstraction*, thus allowing the user to analyze the behavior of a program at the desired level of detail.

Most tools implement at least two different levels of abstraction: the system level and the process level. At the system level the interactions between the different processes can be observed and global statistics are displayed, whereas at the process level local process related events can be observed.

Some tools such as TOPSYS (see section 6.4) also allow the observation of compute nodes and program objects, such as mailboxes and semaphores.

Using icons to manage the levels of abstraction

From an object-oriented point of view, the execution of a program on a distributed system involves many different objects, such as processors, communication channels, processes, buffers, messages, event-occurrences, etc.

Some tools, such as TOPSYS (see section 6.4), TIPS (see section 6.2) and Faust [GGJ+89] include in their graphical data representations icons which are associated with objects. By clicking on such an icon, a window pops up with additional information about that object.

The use of icons results in structured tools with a clear separation of the different levels of abstraction. The user starts observing the program at the system level. Each time additional information about an object is desired, the user clicks on this object, decreasing the level of abstraction at each step.

Icons allow PIMSY [PTV92] to be scalable by implementing the strategy of "clumping" the information.

Controlling the time

Some tools allow the user at any time to interrupt, to proceed in a step-by step manner, to resume or to start again from the beginning the process of representing the runtime information. Sometimes the above possibilities are referred to as the *replay of an execution*[15].

This control over the time during the presentation of the runtime information is a powerful feature, because it allows a more careful observation of the evolution of the program. A disadvantage of the "replay" feature is that its implementation requires the storage of the run-time information.

ParaGraph features a partial implementation of the replay feature. The user is free to interrupt, to resume or to proceed in a step-by-step manner the process of displaying the runtime information [HE91b] However ParaGraph does not provide a step-by-step backward option. The Seeplex performance monitor on the other hand provides a complete implementation of the replay feature [CK90].

Multiple views

A user-interface should include mechanisms for multiple *views* of the runtime information, presented through one or more media in several representations. Different views are useful for different purposes and often complement each other. This way the user can observe several aspects of the system at the same time and correlate information from different views.

[15] The term "replay of an execution" is used in this context because the user can replay the representation of the runtime information as many times as desired in precisely the same way. The correct term would be "replay the representation of the runtime information". Care should be taken not to confuse this "replay of the execution" with the replay of a program execution. Replaying a program execution implies the re-execution of a program on a machine, in such a way that the events are forced to occur in the same order as in a previously monitored execution.

Adding new views

Often there is a considerable gap between the user's mental conception of a problem and the creation of a program that solves the problem. In Voyeur [SBN88], an original solution has been found to this problem that will be briefly discussed here.

The main idea of Voyeur is that the user should be allowed to add high-level application-specific views of parallel programs to the user-interface. These views represent the program and its behavior in the way the user conceives it (high level objects) and not as abstract (low level) data-structures that are modified by data-flow events. For this purpose Voyeur provides the user with an easy-to-use, language- and system- independent mechanism for specifying views and building graphical representations of parallel programs.

Paragraph also offers the possibility of adding new views, but this is a much more complicated matter since no special language has been provided and new views have to be coded in straight X11.

6 Tools Inventory

We will take a closer look at some real tools and programming environments. The monitoring systems that will be presented cover most of the aspects of parallel monitoring that were discussed in the previous section.

The main idea behind these tools is that when dealing with the enormous complexity of distributed programs, visual dynamic representations of the execution of such programs might considerably help the user in a better understanding of her/his program and thus lead to significant improvements.

6.1 PICL/ParaGraph

6.1.1 Introduction

ParaGraph is a trace-based, highly graphical, interactive visualization tool that has been developed at the Oak Ridge National Laboratory [HE91a, HE91b]. The Portable Instrumented Communication Library (PICL) is the instrumented library that generates runtime information for ParaGraph [GHPW90, Wor92]. Paragraph is used in many laboratories [DL93, FJA91] and has been ported on top of different parallel systems [vRT92b, GHSG92]; thus it has became a de facto standard and will be presented as a basic tool.

ParaGraph displays the run-time information through different animated graphical views. The user is free to select one or more of those views according to his needs, thus implementing the idea of multiple views presented in section 5.4.

ParaGraph was used first together with the PICL subroutine library that handles the trace-generation. ParaGraph runs on most workstations with an X-Window graphical interface, whereas PICL implementations exist for the Intel iPSC family, the Ncube/3200 family and the Ncube/6400.

6.1.2 Generation of the run-time information

Whereas ParaGraph forms the user-interface to the run-time information, the *PICL library* actually generates this information. PICL is a subroutine library that can be used to develop parallel programs that are portable across several distributed-memory multiprocessors [Wor92].

The routines that are provided by PICL implement many of the system dependent functions, such as the communication primitives, synchronization, and other system calls. Part of the library are some high-level functions, such as broadcasting. A program that has been written using PICL primitives is completely portable to any machine to which PICL is supported.

The functions of the PICL library have been instrumented with software probes and PICL can be used to automatically generate event-records. PICL generates both event-records about event-occurrences and activities. ParaGraph uses a subset of the event-records which are generated by the PICL library.

ASCII and binary trace-formats are supported by PICL 3.1.1 :

Example :
- `Verbose format :  SEND clock 0 2035 node 12 to 6 type 8 lth 256`
- `Numerical format :  4 0 2035 12 6 8 256`

e	the sending of a message (`send_begin`, `send_end`)
e	the receiving of a message (`receive_begin`, `receiving`, `receive_end`)
e	event-records related to the tracing itself (`trace_start`, `trace_end`, `trace_transport`)
e	control-flow events (`procedure_begin`, `procedure_end`, `user_defined_control_flow_event`)
a	the cumulative busy/idle times of a node
a	the cumulative communication statistics (number of messages, bytes) of a node

Table 3: events monitored by ParaGraph (e for events and a for activities

Several functions have been provided in PICL to control the trace-generation [GHPW90]. Three different types can be distinguished:

functions that allow the user to control the mechanism of "conditional event-record generation" and thus the events that are traced.

functions that control the actual tracing. These functions allow the user to start and stop the tracing individually at each node at any time during the program execution. Using these functions during the execution of a program will result in partial trace-segments in the global event-trace.

functions that allow the user to control which transport strategy is used. The user has the choice between "store and unload when buffer full," "store and unload at request," and "store and unload afterwards."

To relate events from different nodes, PICL uses a global time approximation scheme. This scheme implements a global clock by synchronizing the local clocks at the beginning of the tracing and then taking the local time of these clocks for the global time (see section 2.10).

Simply using the PICL routines increases the execution time very little, while tracing increases the execution time for a single interprocessor communication call from 20 to 50 μs on the Intel iPSC/860, depending on the level of tracing. When two processors are exchanging 1 byte of information (the worst scenario), this corresponds to an increase of 37% and 99% of executiontime respectively. The overhead is independent of the length of the messages and thus the longer the messages, the smaller the relative amount of overhead. Note that only interprocessor communication calls are automatically traced, and only they introduce overhead when tracing is enabled. Thus, a program with little interprocessor communication or one where most messages are fairly large will notice little perturbation. On the other hand, a program dominated by the sending of small messages may notice a significant perturbation. The technic used by PICL to measure busy/idle times is relatively accurate, with most of the overhead being lumped with the busy time. Thus, modulo changes in the order of events, the idle time is a reasonable measure independent of the type of program.

6.1.3 Visualizing the runtime information

The *ParaGraph tool* is fully independent of the PICL library, the only link between the two being the PICL trace-format. ParaGraph executes as a post-processor that replays the execution, based on the information contained in a trace-file.

ParaGraph can simultaneously display the trace information of various views. At each occurrence of an event the affected representations are updated. The user has the possibility of interrupting this process at any time or of proceeding in a step-by-step manner (see also section 5.4). During the execution, new views can be opened or displayed views can be closed. ParaGraph uses colors extensively to represent the runtime information and is thus most effective on color screens, although it can be used on any machine that runs X11.

ParaGraph currently provides more than 25 representations of the runtime information. These representations will be presented here. They have been grouped according to the classifications introduced in section 5.4.

Representing interactions between different objects

Spacetime Diagram : In ParaGraph the spacetime diagram is used to represent the interactions between different processors (see section 5.4).

Animation : Animation also displays the interactions between different processors. The multiprocessor is depicted by a graph, whose nodes represent processors and whose arcs represent communication links. The status of each node (busy, idle, sending, receiving) is indicated by its color or shading. An arc is drawn between the source and the destination processors when a message is sent and erased when the message is received. This representation could be particularly useful in real-time systems for visualizing the behavior of processors, because probable blocked processes are easily detectable.

Communication Matrix : Messages are represented by color or shading in a two-dimensional array whose rows and columns correspond to the sending and receiving processors. The length of each message is indicated by the color or shading used. During the simulation, this view gives current information (each message appears when sent and disappears when received), but at the end of the simulation the display shows the cumulative communication volume for the entire run between each pair of processors.

Representing information related to one object

- System level information

  Kiviat Diagram : This view gives a geometric depiction of the individual processor utilization and the overall load balance. Each processor is represented by a spoke of a wheel.

  Utilization Count and Utilization Meter : Utilization Count displays a histogram showing the total number of processors busy as a function of time, whereas Utilization Meter shows the current number of busy processors in a barchart.

  Concurrency profile shows the percentage of time during the run that exactly k processors were in a given state (i.e. busy/idle/overhead).

- Node level information

  Gantt Chart depicts the activity of the individual processors by a horizontal bar chart in which the color or shading of each bar indicates the busy/idle/overhead status of the corresponding processor as a function of time.

  Node statistics provides detailed communication statistics for a single, user-selected processor. The choices of statistics plotted are: source, destination, length, and type of messages sent to or from the chosen processor.

Representing other information

Trace-record : This is an example of a text-based representation. It displays the event-record that is currently read from the trace-file in a verbose format.

Task displays : By inserting calls to the `block_begin` and the `block_end` functions the user can define the beginning and the end of tasks. Tasks can be nested and several nodes can work on the same task.

ParaGraph provides different views that display information about the execution of tasks. Task Count shows the number of processors that are executing a given task at the current time. Task Gantt depicts the activity of individual processors by a horizontal bar chart in which the color of each bar indicates the current task being executed by the corresponding processor as a function of time. Task Status shows the state of execution of each task. At the end of each run, Task Summary shows the percentage of execution time of each task as a function of the total execution time.

Other views include Critical Path, Clock, Phase Portrait, Message queues, Communication meter, Communication traffic, Processor status, Statistical Summary, etc. For more information about these views refer to [HE91a].

6.2 TMON

6.2.1 Introduction

The Transputer MONitor forms an integral part of the *TIPS* parallel programming environment as described in [WCG+92] that was developed at the University of British Columbia.

The TIPS parallel programming environment consists of 4 major components: TMON (a performance monitor), TMAP (a process to processor mapping tool), TVIEW (a graphical interface), TRES (a tool to identify and analyze the resource requirements) of an application.

The TIPS environment runs on the 74 node transputer system of the University of British Columbia that is hosted by a Sun-4 Unix workstation and requires the Trollius operating system as a software environment.

6.2.2 Main ideas

Four main goals guided the design of TMON: *extensibility* (the system should allow the incorporation of a wide range of user interfaces and analysis packages), *transparency* (the instrumentation should be as transparent to the user as possible), *efficiency* (the impact on the performance of the system should be as little as possible), *accuracy* (the results should reflect the behavior of applications when run without the monitor).

6.2.3 The global system structure

TMON is a hybrid monitor (see 3.5) built on top of the Trollius operating system [BD91]. One transputer is assigned as the master (reactive) node and is capable of interrupting all other nodes in the system to start the monitoring simultaneously. By sending commands to this node, the user can control the monitoring.

In TMON the "specialized processes" model has been adopted. On each slave node a meter process, a back-end process and a buffer manager work together to provide a complete monitoring system. The meter process collects the event-records generated by software probes and stores them in buffer pools. The back-end process performs sampling and the clock synchronization in response to global interrupt signals. The transportation to the host is handled by the buffer manager ("unload when low load" complemented by "unload when buffer full" strategies have been adopted).

On the host, a collector process collects event-records from all slave nodes. These event-records are then sent to the display process that handles the real-time graphical representation of the runtime information.

TMON simulates a global clock using hardware synchronization (see section 2.10). For this purpose a special synchronization mechanism controled by the master node has been implemented. Originally a software synchronization solution was examined, but since hardware synchronization resulted in a maximum offset between the clocks that was 10 times inferior to software synchronization, the latter method has been adopted.

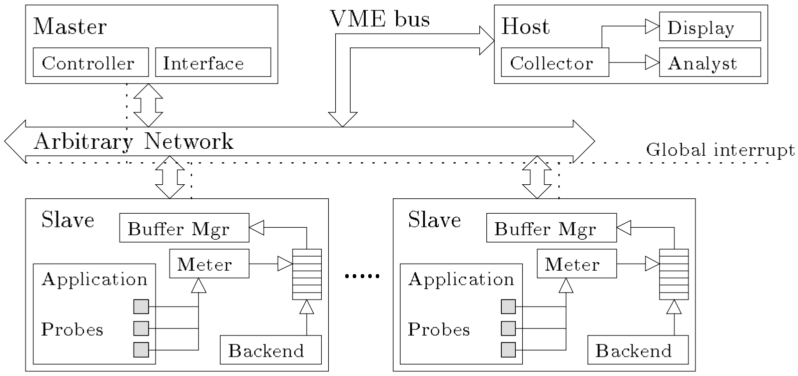

Figure 12: Basic architecture of the Parallel Monitor

6.2.4 Generation of the run-time information

Software instrumentation is used for the event generation. The probes that generate these events are permanently implanted into the Trollius run-time library. To instrument a program a user simply recompiles the program with the instrumented version of the run-time library (see also section 3.3). For the generation of the timestamps the value of the synchronized local clocks is used.

6.2.5 The TVIEW graphical interface

The *TVIEW* tool constitutes the user-interface to the TIPS environment. TVIEW incorporates system and node levels of abstraction. Icons have been used to structure the access to these different levels of abstraction. TVIEW allows the representation of several views at the same time and most windows are updated dynamically, although not in real time, because of the delayed availability of the traces.

The following representations are currently available in TMON:

Representing interactions between objects The interactions between the different processors are represented in a spacetime diagram (see section 5.4). The use of icons has been added to this diagram. Each event is represented by a specific event icon and if the user clicks on this icon, a small window pops up with information pertaining to that specific event.

Representing information related to one object

Topology display and utilization information

: The topology display reflects the network topology of the transputers, i.e. how the transputer nodes are interconnected. Each node in the topology display maintains a regularly updated bar chart that shows additional statistical data about processor and link-utilization, memory load statistics and event statistics for that node. Clicking on a node gives additional information about the current processor utilization and the processes currently running on the node, whereas clicking on a link provides information about the current link utilization.

Message passing display

: This display is organized as a two-dimensional array. The rows correspond to the sending nodes, whereas the columns corresponds to the receiving nodes. In each square of this array, a bar represents the number of messages sent within some interval of time and the density of bars reflects the message activity between a pair of nodes.

6.3 SIMPLE

6.3.1 Introduction

SIMPLE is a performance evaluation tool environment that has been developed at the University of Erlangen and is described in [Moh90].

Much attention was paid in the SIMPLE project to produce a portable tool environment that was not bound to any particular machine. This resulted in an object-oriented data access interface that makes the SIMPLE code easily portable and easy to interface to other existing tools.

SIMPLE runs on Sun workstations and uses the runtime-information generated by the ZM4 hybrid monitor. This monitor is capable of monitoring most parallel and distributed systems. SIMPLE monitors the real-time sequences of interesting activities in the system and makes their interactions visible. An implementation of SIMPLE exists for Sun workstations and a ZM4 monitor has been built.

6.3.2 Generation of the run-time information

The ZM4 is a hybrid monitor (see section 3.5) and the events are generated by inserting small software probes into the program code. The execution of these probes can be detected by the hardware part of the monitor. This way a minimum system overhead is achieved, while maintaining a maximum degree of flexibility.

As can be seen in Figure 13, the ZM4 is structured as a master/slave configuration that consists of one central control and evaluation computer (CEC), and a variable number of distributed monitors. The ZM4 uses a distributed collection system (see section 7), and a global clock that has been implemented by running the local clocks in a lock-step fashion (see section 2.10). The runtime information is stored locally at the monitors. At the end of a program execution the monitors transfer the recorded data to the CEC over the data channel of the ZM4.

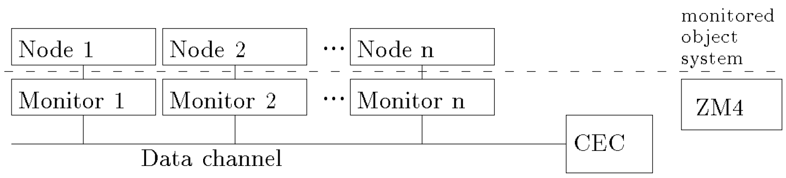

Figure 13: The ZM4 hardware architecture

In [OQM91] two different approaches to event detection with the ZM4 are described. In a sort of memory mapped I/O, a part of the memory addresses are reserved for monitoring uses. The monitor observes the address bus and reads the data from the bus that is written to such a reserved address. With direct channel I/O, the runtime information is sent over an unused communication channel. This channel is connected to the monitor that receives the runtime information. Whereas the overhead of the former approach is about 100ns (one assignment instruction), the overhead of the latter varies between 4μs and 200μs, depending on the number of processes on the observed node.

6.3.3 The SIMPLE environment

The SIMPLE environment consists mainly of two different parts (see Figure 14): the *TDL/POET tool* and the performance evaluation tools.

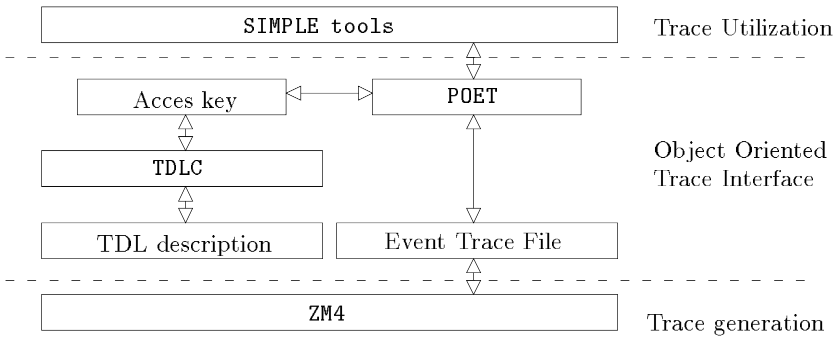

Figure 14: The architecture of the SIMPLE environment

TDL/POET-a basic tool for accessing measured data The basic idea of the TDL/POET tool is to consider the measured trace-data as a generic abstract data structure. Evaluation tools can only access the measured data via a uniform and standardized set of generic procedures. The advantage of this approach is that the evaluation is independent of their recording, and thus allows the use of other monitors or other evaluation tools.

The *POET* (Problem Oriented Event Trace interface) library, allows the user to access the measured data in an object oriented manner. With a great variety of functions such as “get_next”, “forward”, “goto” the user accesses the different event-records. Other functions such as “get_time”, “get_token” that allow the user to obtain specific fields of the selected traces. In order to be able to access and decode the different parts of the measured data the POET functions use a so-called access-key file, which contains a complete description of formats and properties of the measured data.

In order to facilitate the construction of the access-key, a trace-format description language has been developed. This language called *TDL* can be compiled with TDLC (TDL Compiler) into an access-key.

The performance evaluation tools of SIMPLE SIMPLE offers a set of tools that implements most of the steps described in section 5.

Building a global view of the execution : The first step after the program execution is to integrate the local trace-files (one per node) into a global trace-file that provides an integrated view of the whole object system (see section 5.1). This action is performed by the tool *MERGE*, which creates one global trace-file with one access-key. The event-traces in this new file are sorted according to their timestamps.

Validation and plausibility test : The second step in the SIMPLE approach is to test whether all used monitor devices have worked correctly and the measurement was performed without errors. *CHECK-TRACE* performs some simple tests on the given trace-file. In *VARUS*, on the other hand, the user can specify validation rules specific to a measurement and object system in a formal language. Both tools generate a report with all detected errors.

Filtering and clustering the events : Before evaluating the trace-file, the event-records can be filtered and clustered. Filtering means that only a subset of the generated event-traces is retained. The tool *FILTER* allows one to do this by selecting event-records on their attribute values. *ADAR* on the other hand allows the clustering of events. For this the user defines composite events with their attributes, called activities. Using this specification, ADAR then automatically finds the occurrences of these activities and computes their attributes. (see also section 5.2)

Visualizing trace-files : The *SMART* (Slow Motion Animation Review of Traces) tool is a simple visualization tool that can be used on ASCII terminals. *VISIMON*, on the other hand, is based on X-Windows and has enhanced graphics capabilities. The use of the S package of AT&T in SIMPLE provides the user with a high-level programming language for data analysis and graphical evaluations like histograms or piecharts.

Other tools

One very interesting tool available in the SIMPLE environment is the *AICOS* (Automatic Instrumentation of C Object Software) tool. This tool provides automatic instrumentation of procedures, procedure calls and arbitrary statements for programs written in the programming language C (see section 3.2).

6.4 TOPSYS

6.4.1 Introduction

TOols for Parallel SYStems is an integrated tool environment that has been under development at the TU-Munich since 1986.

TOPSYS has been developed for distributed memory systems that communicate through message passing. The environment consists of a small kernel (MKK) that runs on each of the nodes of a target machine, a monitoring system that gathers runtime information, a mapping tool and automatic load balancer, a parallel runtime debugger (*DETOP*), a performance analyzer (PATOP), and an execution visualizer (VISTOP). For a more detailed presentation of TOPSYS refer to [BBB+90, BB91, Bem90].

Currently, an implementation of the *MKK kernel* exists for the iPSC/2 with both software, hardware and hybrid versions of the monitoring system. The tools have so far been implemented on Sun 3 workstations using the X-Window graphical environment.

6.4.2 Main ideas

The main aim of the TOPSYS environment is to simplify the use of parallel systems by hiding from the user details of the underlying system. To this end, an object-oriented programming model [BL92] has been chosen that offers active objects (tasks), communication objects (mailboxes), synchronization objects (semaphores) and storage objects (memory). When using this programming model, the programmer thinks about his parallel program in terms of objects and standard operations upon these objects. Automatic tools take care of the object handling, such as the mapping of the different objects and load balancing between the different nodes.

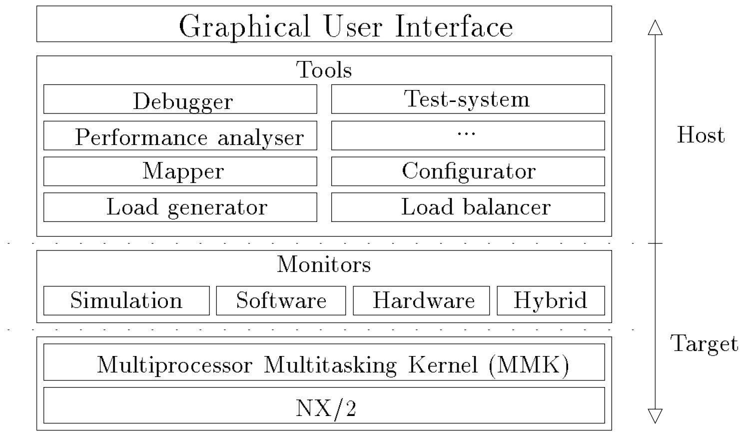

Figure 15: The TOPSYS architectural model

Runtime information forms the input of many of the TOPSYS tools. Therefore all tools have been built on top of a common monitoring system that gathers runtime information (see Figure 15). On top of the tools a common graphical interface to all tools has been implemented in X11.

6.4.3 Generation of the run-time information

The monitoring model The monitoring system of TOPSYS is based on the event-action model (see section 2.6). All three different classes of atomic events (see section 2.7) can be monitored with TOPSYS:

- execution events,
- data events,
- parallelity events.

The tools allow the specification of compound events that can be detected by the different monitors (see section 2.7). For a more detailed list of the events that can be monitored with TOPSYS, see [BBB$^+$90].

The monitors TOPSYS offers hardware, software and hybrid monitoring [BLT90]. The interface to the different monitors is the same and their use is therefore interchangeable. Due to the clear definition of the monitoring interface, simulations of a program execution can also be used as long as they comply with this interface (see Figure 15).

In TOPSYS a distributed collection system is used. At each node a local monitor is responsible for the monitoring of the processes on that node. For monitoring distributed compound events, the local monitors communicate via a communication network.

The three different monitors will be briefly presented here:

The software monitor : has been implemented using the "specialized processes" model and consists of two MMK-tasks. One task does the communication with the other monitors while the other task is responsible for the event detection (see section 3.2).

Local events are detected by software probes that are inserted in the source code or by hooks in the MMK kernel (see section 3.2).

The hardware monitor of the TOPSYS environment consists of dedicated hardware that directly monitors the electrical signals of the processor of a node. The different monitors are tied together via a bus that allow the detection of distributed compound events.

The hardware monitor uses physical probes to detect the occurrence of hits and possesses special hardware that allows the clustering of hits into application-level events (see section 3.4). The hardware monitor possesses both a local and a global clock for time interval measuring and timestamping.

Due to the complexity of the hardware-monitor card, a project has been undertaken to integrate this monitor into a VLSI chip.

The hybrid monitor

The TOPSYS hybrid monitor uses part of the hardware of the hardware monitor. However, instead of monitoring the electrical signals of the system, the relevant data is written directly into the registers of the hardware monitor. This increases the flexibility and only slows the execution down by about 2% [BB88].

6.4.4 The tools

In TOPSYS different tools that allow the observation of the runtime information are provided. These tools provide graphical representations of views of the runtime information. Multiple views from one or multiple tools can be displayed at the same time. Among the tools available in TOPSYS, PATOP and VISTOP will be discussed in more detail here.

PATOP-Performance Analysis TOol for Parallel systems PATOP [BHL90] is a performance oriented tool that incorporates system, node, and object levels of abstraction. Icons have been used extensively to structure access to these different levels of abstraction.Multiple views on the runtime information can be displayed at the same time in either curve diagrams or barcharts.

The following measurements are available in PATOP [BBB+90] :

At the system level

- Idle time of the whole system,
- Average percentage of time that tasks spend in ready queues.

At the node level

- Idle time of a processing node,
- Average percentage of time that tasks on a node spend in ready queues,
- Number of kbytes sent to other nodes,
- Average percentage of time that tasks on a node spend in remote actions.

At the object level

- Percentage of time tasks are waiting in send or receive operations,
- CPU usage of each task,
- Waiting time at semaphores,
- Number of kbytes sent to or received from each mailbox on any node,
- Delays of tasks waiting at mailboxes.

VISTOP-VISualization TOol for Parallel systems VISTOP [BBB$^+$90] displays the dynamic behavior of concurrent programs running under the distributed operating system kernel MMK. The main idea behind VISTOP is that to understand the behavior of a parallel program, it is crucial to know what communication and synchronization events take place. Therefore, VISTOP concentrates on the monitoring and visualization of these two types of events.

VISTOP's interface is entirely graphical. Objects such as tasks, mailboxes and semaphores are represented either in iconified or de-iconified form. The de-iconified form provides additional information concerning the state of the object. The interactions between different objects are represented by arrows linking the objects that interact. TOPSYS provides a complete replay feature (see section 5.4) that allows continuous or step-by-step (forward and backward) running of the representation clock.

6.5 Pablo

The following introduction is extracted from [RAM$^+$92].

> Pablo is a performance analysis environment designed to provide performance data capture, analysis, and presentation across a wide variety of scalable parallel systems. The Pablo environment includes software performance instrumentation, graphical performance data reduction and analysis, and support for mapping both graphical and sound performance data.
>
> [...], an ideal performance analysis environment should support interactive insertion of instrumentation points, as well as data analysis, reduction, and display. Moreover, the environment should be *portable* across a range of parallel architectures, its performance and data analysis capabilities should be *scalable* with the size of the system being studied, and it should be extensible, allowing the user to add environment functionality as needed.

6.5.1 portability and scalability

For Pablo, portability means homogeneous interface. Therefore, Pablo is designed to "hide" the machine on which the program runs to allow the user to compare the behavior of his program on several machines. This standardization of interface is twofold; it allows not only the machine to be changed, but also supports an increase in the number of processors.

6.5.2 Pablo users

Reed et al. believe there are "three classes of potential performance environment users : novice, intermediate, expert" :

- the novice wants a performance tool which identifies problems easily.
- the intermediate wants to modify the existing tool, for example, combining metrics.
- the expert knows the parallel architecture and the system software and wants to increase the power of the tool by adding new toolkit components.

The aim of Pablo is to be useful to these three kinds of users.

6.5.3 Pablo software overview

Previous work by the Pablo research group has shown the necessity of having a "portable, scalable and extensible" tool. Thus, the two parts of Pablo satisfy this request : (1) portable software instrumentation, and (2) portable performance data analysis.

The instrumentation The instrumentation part allows the user to monitor three classes of events :

- trace events : represent the occurrence of specific events (with an arbitrary amount of user-specified data).
- count events : allow the user to count the number of occurrences of an event (the timestamp is insignificant).
- time interval events associate an event with a pair of source code points.

For events of all three classes, the Pablo trace capture library supports optional user-written extension functions that can process event data before it is written to the trace file. Users can thus generate high level events (computing inclusive and exclusive lifetimes of procedure calls), or filter them. Concerning the amount of data generated, two filtering techniques are included in this tool. A notion of local and global *threshold* is added (for example, if the generation rate of interval events is too high these events are converted to count events). On the other hand, the user can define an interval rate in which the generation must stand. The adaptable generation does not affect the generation of the event but does affect its storage in the trace file.

The data analysis The second part of Pablo is a graphical interface (using Motif) to analyze the data contained in trace files. The link between the instrumentation and the analysis is a trace file following the "self-describing data format" (SDDF) [Ayd93]. This meta-format allows for changes in the instrumentation without modifying the analysis tool. Obviously no semantic can be added in the trace file, so every field is recognized by a syntactic analysis.

For example [RAM+92] :

```
/* "Trace generation date" "November 1, 1991"
 *
 */ ;;

#1:
// "event" "message sent to one or more processors"
"message send" {
        double  "timestamp";
        // "From" "Processor sending message"
        int     "SourcePE";
        // "To" "Processor(s) receiving message"
        int     "DestinationPE"[];
        // "Size" "Message length in bytes"
        int     "Message Length";
};;

#2:
"context switch" {
        double "timestamp'';
        int    "processor_number";
        char   "processor_name"[];
};;


"context switch" { 100.150000, 2, [30] { "Process 23" } };;
"message send"   { 100.100000, 0, [4] { 1, 3, 5, 7 }, 512 };;
"message send"   { 102.150000, 7, [1] { 1 }, 1012 };;
"context switch" { 108.000000, 4, [30] { "File I/O" } };;
```

A new analysis view can be created by forming a directed acyclic graph. Its nodes represent transformation modules and its edges re[resent the data flow. The modules must compute with no semantic to be standard and portable. For example, a module that computes the average of a field can receive any scalar array and send the average without any knowledge about the semantic.

Some modules are already designed, such as :

- Statistical transforms : skew, kurtosis, median, mode,...
- Mathematical transforms : counts, sums, products, differences, ratios, maxima, minima, averages, absolute values, power, logarithmic,...
- data filtering transforms : sorting, ranking, linear interpolation, extraction of sub-vectors or sub-arrays, scaling, histogramming and curve fitting.
- graphical displays : bar graphs, bubble charts, strip charts, contour plot,s dial, interval plot, kiviat diagrams, LEDs, X-Y line/scatter plots, matrix displays, pie chart, polar plots, 3-D scatter plot.

6.6 IPS-2

This tool was developed at the University of Wisconsin by Jeff Hollingsworth, Bruce Irvin and Barton P. Miller. IPS-2 [HIM91], is "*an interactive, trace driven performance measurement system for parallel and distributed programs*".

The IPS-2 can measure shared-memory and message-passing parallel programs running on a heterogeneous collection of machines. The events are only generated by software probes inserted in the source (C or Fortran). The visualization is managed by a X window interface.

The approach of this tool is interesting because the program execution can be analyzed across four different levels : the program level, the machine level, the process level and the procedure level. Different metrics are associated with each of these levels. For example, the amount of CPU time used can be displayed for the whole program or for the individual machines, processes, or functions. The metrics can be displayed via different shapes : histograms (evolution of the metrics along time), tabular summary, for example.

Furthermore, users can add their own metrics. The program and the representation of the new metrics are linked by *External Data collectors units* (EDCU). The EDCUs are divided into two parts : the metric specific part (how to generate the information), the histogram generation part (how to use it).

6.7 Maritxu

Maritxu [ZT92a], developed at the University of York (UK) by Eugenio Zabala and Richard Taylor is a tool close to the hardware (processors, network,...). Its aim is to be:

- a set of tools for the run time visualization of many models of parallel computation, focusing on the *processors*;
- processor type, network size and topology independent;
- problem and language independent;
- capable of the visualization of multiple parallel programs running simultaneously in subnetworks of a complete network;
- capable of being run on any hardware platform that supports Unix and the X-Windows/Motif graphical systems.

The Maritxu tool has been designed by taking into account the psychological aspect of perception. Therefore, the use of color has been chosen because human being has a better graphical perception than a textual one. Furthermore, Maritxu has been enhanced by the addition of sound [ZT92b]. The user of Maritxu has thus not only visual stimuli but also audio ones.

6.8 PIE

PIE [LSV+89] (Parallel Programming and Instrumentation Environment) was developed at Carnegie Mellon university. Its philosophy is to be "*efficiently mapping parallel applications onto specific architectures and observing them when they execute*". PIE supports language such as C, MPC, C-threads, Ada and Fortran.

Programs in PIE are instrumented by software probes. The information generated is then displayed in a X windows environment using PIEScope. This tool contains views such as histograms, time lines.

6.9 PIMSY

PIMSY is being developed at the Laboratoire de l'Informatique du Parallélisme of the Ecole Normale Supérieure of Lyon (France). The aim of this project is to build up a scalable data visualization tool.

To make the trace analyzer scalable, the traces stay split into different files according to their generation position.

The software is divided mainly into two parts : the servers and the clients. The server set gives the clients the information they want by communication between them and trace filtering. A server manages not only a set of trace files but also a set of clients who can only communicate with it.

A typical session of PIMSY can be summarized as follows :

- a client, called source-client, asks its server, called source-server, for information that can be filtered on the time, the space and/or the event types.
- the source-server asks the corresponding set of servers the information they have,
- the information is gathered[16],
- the information is broadcast to a set of destination-clients that the source-client specifies (via destination-severs),
- each destination-client returns an acknowledgment at the end of its work to the source-server,
- a final acknowledgment is sent back to the source-client by the source-server.

The set of clients can be extended indefinitely because a client is simply a task that communicates with its server using a predefined protocol. Two types of clients have been developed. The first are hierarchical graphical clients, the second are audio clients.

The hierarchical clients avoid bottlenecks by using the "clumping" method to reduce the information displayed at a time. Different "clumping" views have been implemented with automatic detection and localization of performance problems. The aim is to begin the search with the coarsest view, and afterwards to refine more and more at each step.

Since the clients can be as generic as possible, we implemented sonification tools to map events with sounds. Different mapping can be produced to localize abnormal behavior.

6.10 Comparison

To summarize and compare the above described tools, we selected several criteria which correspond to the general characteristics discussed in the paper. The table 6.10 allows the reader to have a synthetic view of the different tools.

Some remarks need to be made :

- The column ParaGraph contains PICL and ParaGraph,
- The column PIE concerns the union of PIE and PIEScope.
- in the language row, C++ means that at least a part of the tool is written in C++. As the C is contained in the C++, other parts may be written in C. For example, the data capture software of Pablo is in C, and the data analysis software is in C++.

[16] several strategies can be used to gather the information

	ParaGraph	TMON	SIMPLE	TOPSYS	Pablo	IPS-2	Maritxu	PIE	PIMSY
Generation									
Hardware	⊗	○	○	○	⊗	⊗	(?)	⊗	(?)
Software	○	○	○	○	○	○	(?)	○	(?)
Transport									
Immediately	⊗	⊗	(?)	○	⊗	(?)	(?)	⊗	(?)
Progressively	○	○	(?)	⊗	○	(?)	(?)	(?)	(?)
Afterwards	○	⊗	(?)	⊗	⊗	(?)	(?)	(?)	(?)
Analysis									
On line	⊗	○	⊗	○	⊗	⊗	⊗	⊗	○
Off line	○	○	○	○	○	○	○	○	○
Data representation									
Graphics	○	○	○	○	○	○	○	○	○
Audio	⊗	⊗	⊗	⊗	○	⊗	○	⊗	○
Performance	○	○	○	○	○	○	○	○	○
Communication	○	○	○	○	○	○	○	⊗	○
Execution behavior	○	○	○	○	○	○	⊗	○	○
Monitored Machine									
Shared Memory	⊗	⊗	⊗	⊗	⊗	○	⊗	○	⊗
Message Passing	○	○	○	○	○	○	○	⊗	○
Software Specification									
Extensible	⊗	⊗	○	⊗	○	○	⊗	⊗	○
Language	C	C++	C		C++				C++
Requirements									
X11	○	○	○	○	○	○	○	○	○
Motif	⊗	⊗	⊗	⊗	○	⊗	⊗	⊗	⊗

7 Scalability : The need for a distributed monitoring system

7.1 A centralized versus a distributed collection system

A *centralized collection system* could be described according to the monitoring model introduced in section 2.6 by one reactive process $r \in \mathcal{R}$ that monitors the entire set $\mathcal{A}$. A *distributed collection system* on the other hand consists of a set of reactive processes (one per node), where each reactive process monitors those processes that are local to its node. *Partially distributed collection systems* with one reactive process supervising several nodes could be devised. The PIMSY tool [PTV92, vRTV92] is based on this approach and tries to prove the advantage of such a system.

With a simple example, we will now show that only a distributed collection system can provide a general solution to the problem of the gathering of runtime information for massively parallel systems :

Given a target system with 16 processors, that each run at 10 MIPS (which corresponds roughly to the power of today's RISC processor). Suppose that every 1000 instructions one event is generated (this implies an overhead of less than 0.5% [mL92]). This will result in the generation of 10,000 events per second per processor. With 16 processors this results in an event generation rate of 160,000 events per second. With 128 processors the event generation rate will be 1,280,000 events per second.

Detecting and collecting runtime information about 160,000 events per second might still be possible, although a high bandwidth communication network and a powerful central processor would be needed. With 1,280,000 events per second it is obvious that a centralized collection system will no longer work. The bottleneck stems from both the communication network and the processing power of the processor.

Therefore, a centralized collection system could only be implemented as a hardware monitor. This creates another disadvantage because it requires considerable additional hardware.

Hence, the only practical and scalable approach to gathering runtime information is the distributed collection system and, to our knowledge, this approach has been generally adopted in existing tools.

7.2 Centralized versus distributed processing

Once generated, the runtime information needs to be processed. This can be done in a more or less distributed manner, and using on-line or off-line processing.

Usually runtime information is centralized at some point during the processing (for example, in Pablo [Noe92], the merging operation is done by hand, just before the visualization), because most applications use runtime information to obtain an accurate global view of the execution of a distributed program. Building this view in a completely distributed manner is feasible, but would require a great deal of communications between the different processes (for example, in PIMSY [vRTV92], the data is filtered and gathered by servers just before being given to processes that requested it).

A partially distributed approach however might be a good compromise : Each of the distributed monitors executes a processing step on the collected runtime information before this information is centralized and further processing takes place. In this *distributed processing step* the amount of generated traces can be reduced by filtering (eliminating unimportant event-records) and clustering (grouping event-records into a higher level of abstraction). How much of the processing work can be done in a distributed manner and the communication overhead of this processing remains to be investigated.

Let us continue the exploration of the example of the previous section :

Suppose that each event-occurrence results in a event-record that contains 10 bytes. Under such circumstances, one processor (read "compute-node") will generate 100 kbytes of trace-data per second. The system as a whole with 16 processors will generate 1.6 Mbytes per second of trace-data, whereas the system with 128 processors will produce roughly 13 Mbytes per second. In the off-line case this will result in roughly 100 Mbytes of trace-data on a 16-processor system and 0.8 Gbytes on a 128-processor system for one minute of monitored execution time. Suppose further that 100 instructions are needed to process one event-record. This means that the on-line processing of the event-records generated at one node require 1 MIPS. On-line processing of all the event-records will require 16 MIPS for a 16-processor machine and roughly 130 MIPS for a 128-processor machine.

From the above example the following conclusions can be drawn:

- *On-line processing* : in a centralized manner with the current state of technology seems possible for small parallel machines, that is machines that do not exceed something like 16 processors. With bigger machines the communication network, but also the power of the central processor will become bottlenecks. Partially distributed processing might provide a solution to this problem by reducing the amount of information to be transported and to be processed.

- *Off-line processing* : in a centralized manner is possible. Most existing tools have adopted this approach, but these tools usually address themselves to small machines. With a 128 node machine however a considerable amount of storage is required and the transporting and processing of the trace-data will take quite some time. Through a first distributed processing step, this situation can probably be considerably improved, requiring less central storage capacity and processing power.

The examples discussed seem to favor a distributed approach to monitoring both at the collection and at the processing level. This is especially true for on-line processing and for very large distributed machines. However, the usefulness of this approach entirely depends on how much of the work done by the central processor can be done in a distributed manner without introducing an important communication overhead.

8 Conclusion and Future Developments

In this report a general approach to the monitoring of distributed memory machines has been presented. Constantly, an effort was made to integrate the work of the different authors into a consistent framework.

Remark that monitors have also been developed for shared memory computers but most of them rely on the common memory assumption and were not suitable for our study.

Today the field of parallel monitoring remains a relatively new field that still has to mature. Theoretical support for many aspects of parallel monitoring is still missing and few unifying concepts have been established. For this reason, a wide variety of different tools exist that all offer more or less the same functionality. Most tools incorporate their own valuable ideas, but there is no compatibility between the different tools and a tool is usually only portable to a small family of similar machines.

The availability of more and larger parallel machines already shows a growing demand for monitoring environments and authors of tools have become more and more concerned with the development of open, portable tools with interchangeable trace data. In our opinion, this tendency will gain importance in the near future and we will assist in the development of monitoring environments that are modular and interchangeable. Such a monitoring environment would consist of several functional modules built on top of a common interface monitoring platform. The user can compose his own monitoring system by choosing those modules that suit best his needs and constraints.

The appearance of such monitoring tools is even more likely, since there are no apparent technical reasons in favor of non-modular monitoring tools. However, for such tools to evolve, it is important that a common theoretical foundation for parallel monitoring and some common standards be developed. Some important theoretical questions that remain to be answered are:

- What is the real impact of monitors on the monitored system and how important is it to precisely know this impact?
- How scalable are the current concepts and techniques for massively parallel machines?
- How should the enormous amount of runtime information that can be generated by parallel machines be analyzed and represented ?
- How accurate is/will be the generated information regarding to the concurrent processes behavior ?

In our research project PIMSY/VIST, we try to incorporate most of the positive ideas described in the preceding in a portable, modular, trace-based monitoring environment that aims to be able to scale with the massively parallel machines.

References

[Abs90] F. Abstreiter. Visualizing and analyzing the runtime behavior of parallel programs. In Burkhart [Bur90], pages 828–839.

[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`.

[Bat88] P. Bates. Distributed debugging tools for heterogeneous distributed systems. In IEEE, editor, *8th International Conference on Distributed Computing Systems*, pages 308–315, 1988.

[Bat89] P. Bates. Debugging heterogeneous distributed systems using event-based models of behaviour. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):11–22, January 1989.

[BB88] H.-J. Beier and T. Bemmerl. Software monitoring of parallel programs. In C. R. Jesshope and K. D. Reinartz, editors, *CONPAR 88*, Manchester, 1988. British Computer Society's Paralle Specialist Group, Cambrige Univerity Press.

[BB91] T. Bemmerl and A. Bode. An integrated environment for programming distributed memory multiprocessors. In *2nd European conference on Distributed Memory Computing*, pages 131–142, April 1991.

[BBB+90] T. Bemmerl, A. Bode, P. Braun, O. Hansen, P. Liksch, and R. Wismüller. TOPSYS - tools for parallel systems (user's overview and user's manuals). Technical Report TUM-I9047, SFB-Bericht Nr. 342/25/90 A, Institut für Informatik der Technischen Unversität München, January 1990.

[BD91] G. Burns and R. Daoud. Trollius reference manual for c programmers. Occam User's Group Newsletter, March 1991. Document Series 2/2.

[Bem90] T. Bemmerl. The TOPSYS architecture. In Bukhart [Bur90], pages 732–743.

[BHL90] T. Bemmerl, O. Hansen, and T. Ludwig. PATOP for performance tuning of parallel programs. In Burkhart [Bur90], pages 840–851.

[BL92] T. Bemmerl and T. Ludwig. MMK - a distributed operating system kernel with integrated dynamc loadbalancing. In Burkhart [Bur90], pages 744–755.

[BLT90] T. Bemmerl, R. Linfhof, and T. Treml. The distributed monitor system of TOPSYS. In Burkhart [Bur90], pages 756–765.

[Bur90] H. Burkhart, editor. *CONPAR 90 – VAPP IV*, volume 457 of *Lecture Notes in Computer Science*, Zurich, Switzerland, September 1990. Springer-Verlag.

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

[CDW92] J. Choi, J. Dongarra, and D. Walker. The design of scalable software libraries for distributed memory cocurrent computers. In Dongarra and Tourancheau [DT92], pages 17–30.

[Cha91] S. Chaumette. A replay mechanism within an environment for distributed programming. In *Proceedings of Supercomputing Debugging workshop '91*, Albuquerque, NM,, November 1991.

[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.

[CRT89] M. Cosnard, Y. Robert, and B. Tourancheau. Evaluating speedups on distributed memory archie ctures. *Parallel Computing*, 10:247–253, 1989.

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

[DJ91] C. Diehl and C. Jard. Interval approximations of message causality in distributed executin. IR 617, IRISA, November 1991.

[DL93] C. Derr and V. Lo. Selection and reduction : Techniques for visualizing massively parallel programs, 1993.

[DT92] J. Dongarra and B. Tourancheau, editors. *Environments and Tools for Parallel Scientific Computing*, volume 6 of *Advances In Parallel Computing*, Saint Hilaire du Touvet, France, September 1992. CNRS-NSF, Elsevier Science Publishers - North Holland.

[Dun91] T. H. Dunigan. Hypercube clock synchronization. Technical Report TM-11744, Oak Ridge National Laboratory, TN, February 1991.

[Fid88] J. Fidge. Partial orders for parallel debugging. In ACM, editor, *Proceedings DIGPLAN/SIGOPS workshop on parallel and distributed debugging*, pages 183–194, May 1988.

[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.

[GGJ+89] V. Guarna, D. Gannon, D. Jablonowski, A. Malony, and Y. Gaur. FAUST : An integrated environment of parallel programming. *IEEE Software*, 6:20–29, July 1989.

[GHPW90] G. Geist, M. Heath, B. Peyton, and P. Worley. A user's guide to PICL (a portable instrumentes communication libary). Technical Report TM-11616, Oak Ridge National Laboratory, TN, November 1990.

[GHSG92] I. Glendinning, S. A. Hellberg, P. A. Shallow, and M. Gorrod. Generic visualization and performance monitoring tools for message passing parallel systems. In Topham et al. [TIB92], pages 139–150.

[GK92] A. Gupta and V. Kumar. Analysing Performance of Large Scale Parallel Systems. Technical Report TR 92-32, Department of Computer Science - University of Minnesota - Minneapolis, November 1992.

[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.

[GS84] N. K. Gupta and R. E. Seviora. An expert system approach to real-time debugging. In *Proceedings of the 1st conference on Artificial Intelligence Application*, pages 336–343. CS Press, 1984.

[Gus88] J.L. Gustafson. The Scaled Sized Model: A Revision of Amdahl's Law. In L.P Kartashev and S.I. Kartashev, editors, *Supercomputing'88*, volume II, pages 130–133. International Computing Institute, 1988.

[GZ84] R. Gusella and S. Zatti. Tempo : A network time controller for a distributed berkeley unix system. *Distributed processing technical communication newsletter - IEEE*, 6(SI-6):7–15, June 1984.

[HE91a] M. Heath and J. Etheridge. Visualizing performance of parallel programs. Technical Report TM-11813, Oak Ridge National Laboratory, TN, May 1991.

[HE91b] M. Heath and J. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8:29–39, September 1991.

[HG92] G. Haring and Kotsis G., editors. *Workshop on Performance Measurement and Visualization of Parallel Systems*, volume 7 of *Advances in parallel Computing*. North Holland, October 1992.

[HIM91] J. Hollingsworth, B. Irvin, and P. B. Miller. The integration of application and system based metrics in a parallel program performance tool. To appear in Proceedings of the 1991 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 1991. `accessible by anonymous FTP in grilled.cs.wisc.edu:technical_reports/edcu.ps.Z`.

[HM93] J. Hollingsworth and P. B. Miller. Dynamic control of performance monitoring on large scale parallel systems. To appear - International Conference on Supercomputing, Tokyo, July 1993, accessible by `ftp grilled.cs.wisc.edu:technical_papers/w3search.ps.Z`, 1993.

[Hon89] R. Hon. A simple trace interchange format. Technical Report Apple Computer Inc., May 1989.

[IM93] R. Irvin and P. B. Miller. Multi-application support in a parallel program performance tool. Technical Report 1135, University of Wisconsin-Madison, 1210 W Dayton Street, Madison, Wisconsin 53706, 1993. `accessible by anonymous FTP grilled.cs.wisc.edu:technical_papers/multiapp.ps.Z`.

[Imr92] K. Imre. Experiences with monitoring and visualising the performance of parallel programs. In Haring and G. [HG92].

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

[Lam78] L. Lamport. Time, clock and the ordering of events in a distributed system communication. *Communications of th ACM*, 21(7):558–565, July 1978.

[LCSM92] J. E. Lumpp, T. L. Casavant, H. J. Siegel, and D. C. Marinescu. Specification and identification of events for debugging and performance monitoring of distributed multiprocessor systems. In IEEE, editor, *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 476–483, 1992.

[LMCF92] T. J. Leblanc, J. M. Mellor-Crummey, and R. J. Fowler. Analysing parallel program executions using multiple views. *Journal of parallel and distributed computing - academic press*, 9(2):203–217, June 1992.

[LS92a] E. Leu and A. Schiper. Execution replay : A mechanism for integrating a visualizing tool wth symbolic debugger. In L. Bouge, M. Cosnard, Y. Robert, and D. Trystram, editors, *Proceedings of CONPAR 92 – VAPP V*, Lyon, September 1992. Ecole Normale Supérieure, Springer-Verlag.

[LS92b] E. Leu and A. Schiper. ParaRex : a programming environment integrating execution replay -ns visualization. In Dongarra and Tourancheau [DT92], pages 155–170.

[LSV+89] T. Lehr, Z. Segall, D. F. Vrsalovic, E. Caplan, A. L. Chung, and C.E. Fineman. Visualizing performance debugging computer. *IEEE*, pages 38–51, October 1989.

[MAA+89] A. Malony, J. Arendt, R. Aydt, D. Reed, D. Grabas, and B. Totty. An integrated performance data collection analysis, and visualization system. In *Proceedings of the 4th conference on hypercube concurrent computers and applications*, pages 229–236, 1989.

[Mal89] A. Malony. Multiprocessor instrumentation : Approches for cedar. Technical report, University of Illinois at Urbana-Champaign, Santa Fe, New Mexico, May 1989.

[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.

[mL92] R. mac Laren. Instrumentation and performance monitoring of distributed systems. In *Proceedings of the 5th Distributed Memory Computing Conference*, pages 1180–1186. IEEE, 1992.

[MLCS92] D. Marinescu, J. Lumpp, T. Casavant, and H. Siegel. Models for monitoring and debugging tools for parallel and distributed software. *Journal od parallel and distributed computing - Academic press*, pages 171–184, June 1992.

[MN90] A. Malony and K. Nichols. Standards working group summary. In Simmons and Koskela [SK90], pages 261–278.

[Moh90] B. Mohr. Performance evaluation of parallel programs in parallel and distribted systems. In Burkhart [Bur90], pages 176–187.

[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 Simmons and Koskela [SK90], pages 73–97.

[Noe92] R. Noe. *Pablo instrumentation environment User's Guide.* Noe, R., Department of computer Science, Univeristy of Illinois, Urbana, Illinois 61801, October 1992.

[OQM91] C. W. Oehlrich, A. Quick, and P. Metzger. Monitor-supported analysis of a communication system for transputer-networks. In *Proceedings of the 2nd European Distributed Memory Computing Conference*, pages 120–129, April 1991.

[Pan92] Cherri M. Pancake. Graphical support for parallel debugging. In *NATO sponsored Advanced Research Workshop on Software for Parallel Computation*, pages 216–228, Cosenza, June 1992.

[PGUB92] C. Pancake, D. Gannon, S. Utter, and D. Bergmark. Supercomputing '90 bof session on standardizing parallel trace-formts. unpublished document available in Postscript from anonymous ftp from`eagle.cnsf.cornell.edu` in `pub/BOF` as `bof.ps`, November 1992.

[PTV92] S. Poinson, B. Tourancheau, and X. Vigouroux. Distributed monitoring for scalable massively parallel machines. In Dongarra and Tourancheau [DT92], pages 85–101.

[PU89] Cherri M. Pancake and Sue Utter. Models for visualization in parallel debuggers. In *Proceedings of Supercomputing '89*, pages 627–636, Reno NV, November 1989.

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

[RP91] D. Reed and the Picasso group. Scalable performance environments for parallel systems. Available in PostScript from reed.ps, April 1991.

[RR89a] D. Reed and L. Rudolph. Experience with hypercube operating system instrumentation. *International journal of high-speed computing*, pages 517–542, December 1989.

[RR89b] D. C. Rudolph and D. A. Reed. Crystal : Intel iPSC/2 operating system instrumentation. In *Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications*, pages 249–252, 1989.

[SBN88] D. Socha, M. Bailey, and D. Notkin. Voyeur : Graphical views of parallel programs. In ACM, editor, *Proceedings SIGPLAN/SIGOPS workshops on parallel and distributed debugging*, pages 206–215, May 1988.

[SK90] M. Simmons and R. Koskela, editors. *Performance Instrumentation and visualization*, Frontier Series, Santa Fe, New Mexico, May 1990. ACM, Addison-Wesley Publishing Compagny.

[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.

[Sto88] J. M. Stone. A graphical representation of concurrent processes. In ACM, editor, *Proceedings SIGPLAN/ SIGOPS Workshop on Parallel and Distributed Debugging*, pages 226–235, May 1988.

[TIB92] N. Topham, R. Ibbett, and T. Bemmerl, editors. *Programming Environments for parallel computing*, volume A-11 of *IFIP Transactions*, Edinburgh, Scotland, April 1992. IFIP, North Holland.

[vRT92a] M. van Riek and B. Tourancheau. The design of the general parallel monitoring system. In Topham et al. [TIB92], pages 127–137.

[vRT92b] M. van Riek and B. Tourancheau. A framework to parallel monitoring on distributed memory multicomputers. In *Transputer'92*, Besançon, France, March 1992. IAO Press.

[vRT92c] M. van Riek and B. Tourancheau. A parallel monitoring system and its implementation under the trollius operating system. In Topham et al. [TIB92].

[vRTV92] M. van Riek, B. Tourancheau, and X. Vigouroux. The massively parallel monitoring system (a truly approach to parallel monitoring). In Haring [HG92].

[WCG+92] A. Wagner, S. Chanson, N. Goldstein, J. Jiang, H. Larsen, and H. Sreekantaswamy. Tips : Transputer-based interactive parallelizing system, 1992.

[Wor92] P. Worley. A new PICL trace file format. Technical Report TM-12125, Oak Ridge National Laboratory, Oak Ridge, TN 37831, October 1992.

[ZT92a] E. Zabala and R. Taylor. Maritxu : Generic visualisation of highly parallel processing. In Topham et al. [TIB92], pages 171–180.

[ZT92b] E. Zabala and R. Taylor. Process and processor interaction: Architecture independent visualsation schema. In Dongarra and Tourancheau [DT92], pages 55–72.