MONITORING PARALLEL PROGRAMS FOR PERFORMANCE TUNING IN CLUSTER ENVIRONMENTS

J. CHASSIN DE KERGOMMEAUX AND É. MAILLET AND J.-M. VINCENT

1. Introduction. Performance debugging is an important part of the development cycle of parallel programs since obtaining high performances is the main goal of using cluster systems. The objective of performance measurement tools is to help programmers to get the highest possible performances from their programs on their target architecture. Performance debugging usually includes several phases: monitoring to gather performance data, data analysis to correct raw data and compute performance indices and presentation of the performance indices to the programmer, usually by sophisticated visualization tools. This latter phase is presented in Chapter 8. A good survey of monitoring and visualization tools was written by Kraemer and Stasko in [13].

Performance monitoring can be used in at least two different contexts. The first one occurs when a dynamic control of the execution of a parallel program is required. In this case, data collected by monitoring tools are analyzed on line because a rapid feedback is necessary. This is the case of some tools monitoring operating systems activities, such as *xload, perfmeter, top, etc.* of Unix systems [31], used by system engineers to manage computing resources. In addition, the very long execution time of such applications — operating systems may run for days — are not compatible with data collection and post-mortem analysis of the application behavior. Monitoring tools used by real-time systems also help implementing a dynamic control of the systems. The situation is similar for monitoring tools whose objective is to help steering parallel programs on-line to improve their performances [8]. Only applications adapted for being steered can benefit from the use of such tools, for example to balance the load among the processors. Critical to such tools is the latency with which program events are transferred from the monitored program to the end-user, low-monitoring latency conflicting with low monitoring perturbations. In addition, interactivity constraints reduce the amount of analysis that can be performed on monitored data. Another use of on-line data analysis is aimed at limiting the amount of recorded data. For example, in order to analyze long running applications, Paradyn [24] performs problem detection on line, automatically or under the user's direction, without requiring to store any monitoring data. A proposal for a standard interface between on-line monitoring and analysis tools is proposed by Ludwig et al in [17].

On the contrary, to perform a global analysis of the behavior of the observed programs, during a performance debugging cycle, as much performance data as possible needs to be collected, with the lowest possible intrusion. Such constraints are best combined when monitored data can be extracted after the execution of the observed programs and data can be analyzed post-mortem. On-line and post-mortem monitoring are not contradictory since, for example, steering choices based upon on-line monitoring need to be validated with performance measures, which can be best performed using the second type of monitoring and performance data analysis. This chapter being dedicated to monitoring for performance debugging, it mainly deals with this type of monitoring activities.

In order to support performance debugging, a large number of performance indices must be delivered by performance measurement tools so that programmers can detect and reduce the overheads of their programs [1]. These indices can be divided in two main classes:

Completion times which ought to be reduced as much as possible. The objective of performance debugging is most often to reduce the completion time of a program or *makespan*. This index can be decomposed into several measures concerning the time spent by the execution of various parts of the programs such as procedures, communication protocols, etc.

Resource utilization rates indicating what percentage of resource utilisation is spent executing "useful" work. If we consider processor utilisation rates, programmers need to know what percentage of time is spent in various overheads, executing synchronization code, task creation, termination or scheduling, communications or idling. Global resource utilization rates might indicate a problem such as a low processor utilization by the application program and an important idling rate. However to correct such a problem, it is often necessary to use more detailed data. For example, an important idling rate might indicate the presence of a bottleneck, whose origin might be a lack of parallelism in the program, a poor performance of the task scheduler, an excessive use of synchronizations, etc.

The execution of a parallel program can be monitored at several possible abstraction levels: hardware, operating system, runtime system and application. Intuitively, the application level is the most significant one for parallel application programmers, since it is the only place where they can control or adjust parameters. However, it may occur that poor performances of an application can only be explained by observing the impact of programming choices on the runtime or operating systems, during the execution of the application. Being able to relate application level design or programming decisions to runtime or operating system behaviors is still a research issue and will not be dealt with in this chapter.

The organization of this chapter is the following. After this introduction, the techniques for monitoring parallel programs are surveyed. The next section focuses on tracing, deemed the most general monitoring technique for parallel programs. Software tracing is considered as the most portable and widespread tracing technique but suffers of two important drawbacks which hinder the quality of traced data: the lack of global clock in most distributed-memory systems — such as clusters where each node uses its private clock — and the overhead of tracing. Solutions to these issues are presented in the two next sections. The last section identifies performance problems which cannot be identified by tracing alone and sketches a possible detection approach, combining tracing and sampling.

2. Principles of parallel programs monitoring.

Most monitoring tools are either clock driven or event driven [27] (see Figure 2.1).

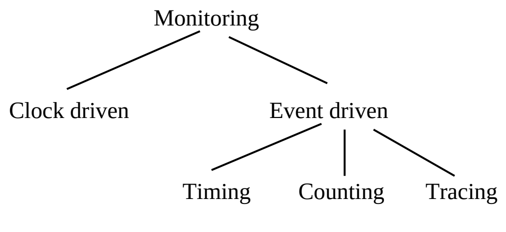


FIG. 2.1. *Classification of monitoring tools*

2.1. Clock driven monitoring or sampling. Clock driven monitoring amounts to have the state of the observed system registered at periodical time intervals, by a process independent of the observed process. The periodicity of recording generally depends on the operating system (typically 20 milliseconds under Unix). The recorded information can be used on line or off line to compute global performance indices.

An example of online monitoring tool is *mon* [34], general-purpose resource monitoring system, which can be used to monitor network service availability, server problems, environmental conditions, etc. Resource monitoring can be viewed as two separate tasks: the testing of a condition, and triggering some sort of action upon failure. *mon* was designed to keep the testing and action-taking tasks separate, as stand-alone programs. *mon* is implemented as a scheduler which executes the the monitors (which test a condition), and calls the appropriate alerts if the monitor fails.

The well-known tools *prof, gprof* [7] belong to the latter category: these tools register the instruction counter value. The registered data is used to compute post-mortem global performance indices. For example, the time elapsed in a procedure of the program being executed is supposed proportional to the number of hits of the procedure in the registered samples.

Performance measurement tools based on sampling are intensively used for performance debugging of sequential programs. It is possible to observe the execution of programs at the programmers' abstraction level without being disturbed by the interaction with the operating system. However, this sort of tool may fail finding the causes of some overheads of parallel programs: global performance indices are of little help to show bottlenecks or to evaluate communication or idling times (unless a processor can be traced busy waiting). In addition, the fairly low periodicity of sampling may be unsuited to exhibit phenomena of very short durations.

2.2. Event driven monitoring. Event driven monitoring is triggered by the occurrences of *events*. We assume that the processes executing an application perform observable events. In this chapter, an event will be defined as an action changing the state of the monitored system, such as a procedure call or the reception of a message. Event driven monitoring aims at associating a date to each of the observed events. The observed events depend on what the programmer is interested in but, in case of parallel programs monitoring, include emissions and receptions of messages as well as "user defined" events. There exist different types of event driven monitoring approaches called timing, counting and tracing, depending on the amount of recorded information and the way it is used.

2.2.1. Timing. The time spent in various parts of the observed program is measured. For example, the time elapsed in a procedure can be obtained by subtracting the clock value at the beginning of the procedure to the clock value measured when it terminates. Such measurements require a low latency clock. The amounts of recorded data are limited to one counter per measured value [4, 24]. Timing intrusion depends on the number of instrumentation points but is potentially high if detailed timing is required.

2.2.2. Counting. The number of occurrences of the observed events is recorded into global performance indices. Counting is generally considered as minimally intrusive and involves the management of limited amounts of data [4, 24].

2.2.3. Tracing. Tracing is done by recording each of the observed events into a performance trace. Each record includes at least the type of the recorded event and the recording date. Additional information is also recorded depending on the type of the event. For example, if the recorded event is a message emission (reception), the record usually includes the identity of the receiver (sender) process and the length of the message.

Tracing is the most general event driven monitoring technique. It is very well suited to measure communication times – it is sufficient to record the emission and reception events – and exhibit bottlenecks – by recording where the processes executing an application spend the time. It can also be used to obtain global or detailed timing or counting information. For example, it is possible to measure the time spent executing a procedure by recording the beginning and the end of each execution of this procedure. For all these reasons, most performance measurements tools for parallel programs executions are based on tracing [29, 9, 35, 2].

However, tracing suffers of several drawbacks. First of all, it may be very intrusive if detailed information is collected. Another problem is that the validity of the recorded data can be corrupted by the interaction with the operating system. For example, the time elapsed in a procedure is the difference between the dates of execution measured at the end and at the beginning of the procedure *only if* the process executing the procedure is not suspended during the execution of the procedure. Therefore, all tools based on tracing applications are well suited to measure performances of single-user systems but may fail obtaining exact performance data of loaded multi-users systems.

3. Tracing parallel programs. As defined above, tracing is the recording of performance events into a trace. As it is the case for all monitoring techniques, tracing can be performed at several levels of abstraction. There exist several tracing implementation techniques [10]: hardware, software and hybrid. The quality of the traces indicates how faithful the recorded information is. It is mainly affected by the lack of global clocks in distributed systems, which makes it difficult to order events occurring on different nodes, and the intrusion of tracing or probe effect, which changes the behavior of traced executions, with respect to untraced ones. The quality of the traces depends on the tracing technique. This section surveys the tracing techniques as well as the factors affecting the quality of the traced information.

3.1. Implementation techniques for tracing.

Hardware tracing: hardware tracers are included in the hardware of the observed parallel system [11]. Such tracers require specific hardware developments and are for this reason considered costly. However they are not intrusive at all. Their use can be difficult for an application programmer since it may not be obvious to relate a hardware event to an algorithmic choice at the application level.

Hybrid tracing: hybrid tracers combine specific monitoring hardware with tracing software [10]. As software tracers, hybrid tracers are triggered by application level instructions. It is therefore easier to relate a traced event to a source program instruction than with a hardware tracer. The traced information is written on dedicated hardware ports, connected to a dedicated monitoring hardware, such that the monitoring intrusion remains extremely low. In addition, the monitoring hardware may include a global clock for dating registered events. Although such monitoring technique can be considered ideal for programmers because it is easy to use and delivers high quality traces, it is not widespread because of its lack of portability and its development cost.

Software tracing: software tracing is the most portable and cheapest tracing technique. It can be done without programmer's intervention, when the tracer is included in a communication library which can be used in "tracing" mode [6]. Tracing can also be done by calling a tracing library from the traced program, calls being inserted by the programmer [29] or by a pre-processor [18]. Software tracing is the most widespread tracing technique because it is cheap and fairly easy to implement. However it makes it difficult to obtain high quality traces because of the lack of global clocks in most distributed memory systems and because of the intrusion caused by the recording and transportation of the trace by the parallel system simultaneously to the execution of the monitored program.

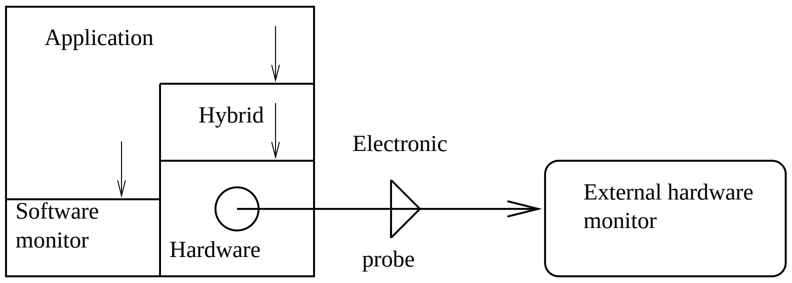


Fig. 3.1. *Hardware, software and hybrid monitoring techniques*

3.2. Software tracing instrumentation techniques. Instrumentation is the insertion of code to detect and record the application events. It can be done at several possible stages during the construction of a parallel program (see Figure 3.2):

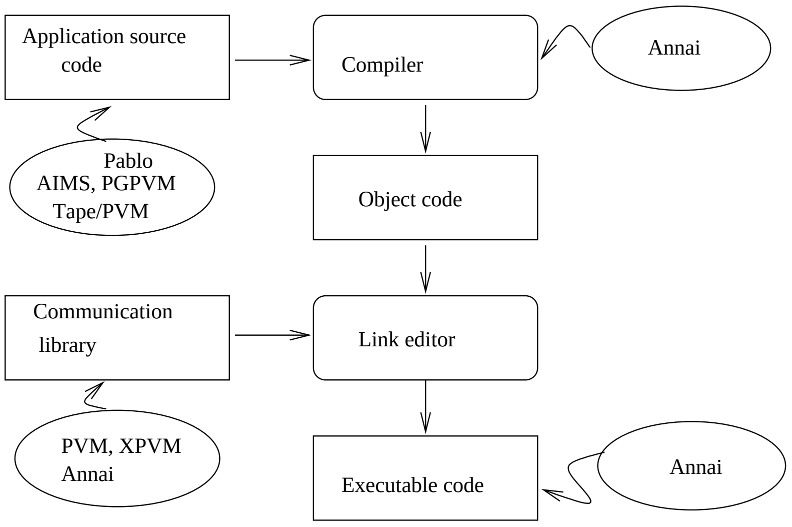


Fig. 3.2. *Instrumentation techniques*

1. Direct source code instrumentation[28, 35, 19]: the instructions generating the events are inserted in users' programs before compilation. Insertion is usually performed by a preprocessor but it can be also done by the user, manually or

through an interactive program. Although it can be implemented easily, this method has the drawback of requiring to recompile monitored programs.
2. Compile-time instrumentation gives monitoring tools access to the information computed by the compiler such as loop dependencies[2]. However it requires to have access to the source code of the compiler and to modify it.
3. Instrumentation of the communication library or of the runtime system has the advantage of requiring no modification nor recompiling of the monitored application[12, 2]. However, application specific events cannot be detected.
4. Direct instrumentation of the compiled object code is independent of the programming language and does not require any recompiling of the applications [24, 2]. Both the application and the communication library can be traced. In addition, users can dynamically adjust the grain and the localization of the instrumentation. The main drawback of this instrumentation technique is that it is not easily portable on various operating systems or hardware platforms.

3.3. Trace format. A trace is composed of event records. Each record contains at least the following information: type of the event, (physical) date of the event and process identification of the process having performed the event. Some records contain additional parameters of the traced event such as receiver (sender) identification and message length in case of message emission (reception) or data fields allowing users to pass information to data analysis and visualization tools. In addition to predefined event records, usually associated to synchronization or communication primitives, there might be some user-defined event allowing programmers to record whatever information they are interested in.

There is no agreement among the scientific community upon the possibility nor the necessity of defining a standard trace format. The approach consisting in using self-defined trace formats such as SDDF for PABLO [29] seems very powerful: the structure of the event records is defined in the headers of the trace files. Besides, the PICL trace format is widely used because traces can be passed to the widespread ParaGraph visualization tool [9]. It seems that converting traces from a format into another one does not raise any serious technical problem: several trace converters were already developed to convert traces collected in various formats into PICL format [18].

3.4. Quality of traced information. Ideally, the storage capacity of the system would be infinite, the recorded events would be dated with an infinite precision global clock and there would not be any tracing intrusion. However this is not the case in general and especially in the case of software tracing. The amount of information to be recorded may exceed the storage capacity of the system, resulting in a reduction of the traced information. The lack of global clock in distributed parallel systems may result into incoherencies between events dated with different local clocks. The intrusion of tracing may change the behavior of the observed program execution.

3.4.1. Buffering and data extraction. The amount of traced data depends on the number of traced events: it can be limited when tracing is restricted to communication events; it may generate a huge amount of tracing data if more detailed measures are needed or the tracing tool is misused. In any case, a potentially large amount of trace data has to be stored and extracted from the parallel system. Various trade-offs can be considered between memory overhead, resulting from the allocation of large trace buffers, and time overhead, resulting from the use of compression algorithms [20] or from the time spent transferring trace data to disk.

3.4.2. Dynamic reduction of the amount of traced data. There exist several approaches to reduce dynamically the amount of recorded data, based on more or less elaborate on-line analysis of the data:

1. The first class of techniques implements a dynamic change of the level of detail of the recordings. In PABLO [29], the trace recording frequency is adjusted dynamically and event tracing may be replaced by counting when the occurrence frequency of traced events becomes too high. Such dynamic adjustment of the amount of collected information is also performed by Paradyn [24]: depending on the truth value of some predicates indicating potential performance problems. In such a case, the observed program is dynamically instrumented to collect more performance data related to the problem.
2. Another technique performs dynamic statistical data clustering in order to limit the recording of event traces to representative processors from each cluster [25].
3. Another approach is based on two basic ideas: the use of "averages" to replace recording data for each instance and "formulae" to represent infinitely long sequences of values [36] — formulae representing some sort of temporal patterns, instead of the spatial patterns used in the previous method. Trace files can then be of "fixed" length, that is independent of the number of iterations and of the problem size. Event traces can be reconstructed by post processing the performance data.

3.4.3. Quality of time measurement. In distributed parallel systems, each processor has its own physical clock. Using the physical properties of the quartz oscillators commonly used for computer clocks, it is possible to model the local time $lt_i(t)$ measured on processor i as a linear dependence [21]:

$$lt_i(t) = \alpha_i + \beta_i t + \delta_i, \quad i \in [1, p], \tag{3.1}$$

where t represents the "absolute" or "universal" time, the constant α_i is the offset at time $t = 0$, the constant β_i (close to 1) is the drift of the physical clock, and the random variable δ_i models granularity and other random perturbations. δ_i can be assumed to be independent of the time t. This model is correct only if the physical parameters (e.g. temperature) of the environment (machine room) remain constant and t is sufficiently small to neglect crystal aging. If these constraints are not satisfied, the coefficients α_i and β_i may no longer be constant.

The lack of global clock in a distributed memory parallel system may result in incoherencies between recorded events if they are dated using the local clocks of the processors. For example the date of reception of a message can be lower than its emission date. Such incoherencies make difficult or impossible the analysis of performance traces by performance measurement tools. On hardware or hybrid tracers, this problem is solved by using dedicated hardware [10]. On software tracers, this problem can be addressed by a software implementation of a clock correction algorithm (see Section 4).

3.4.4. Tracing intrusion. As any monitoring technique, tracing perturbates the execution of the observed parallel programs. It is hard to estimate the intrusion of tracing since it depends on the traced program and on the number of traced events. In case of hardware or hybrid tracing, it is assumed that the tracing intrusion remains limited to a few percent of the execution time. Such an intrusion can be assumed to have a negligible effect on the behavior of the observed program execution [10].

The intrusion of tracing cannot be neglected in the case of software tracing. Several proposals were done to model and compensate the tracing intrusion of software tracers [22, 35, 19, 20].

When modeling tracing intrusion, two types of perturbations are generally defined [22]:

Direct perturbations: resulting from the execution of additional event generation instructions – time spent reading the clock and building an event descriptor in memory – and trace storing on files by the instrumented processes.

Indirect perturbations: localized outside of the tracing code but resulting from the execution of the tracing code. Indeed monitoring can affect the way processes are scheduled and memory is referenced (frequency of page faults and cache misses). It can also hinder some compiler optimizations or penalize performances of I/O subsystems, including file system and network access.

Perturbation compensation models do not take indirect perturbations into account since these perturbations cannot be estimated at the application level of abstraction. However a lot of work is devoted to limit the factors influencing indirect perturbations such as the volume of the traced data.

3.5. Some existing software tracing tools.

3.5.1. AIMS. AIMS[35] includes a set of tools for measuring the performances of parallel programs. The application source code is instrumented by a preprocessor performing a syntactic analysis of the program and building a call graph of procedures and loops: using a graphical presentation of this call-graph, the user can select instrumentation points. Instrumented code needs to be recompiled and linked to a library including data formating and storage procedures. AIMS includes a system for synchronizing clocks and correcting tracing intrusion based on the work of Sarukkai-Malony [30]. The amount of recorded information can also be dynamically reduced by using averages and formulae [36].

3.5.2. Annai. Annai[3] provides an instrumented communication library — to observe communications – as well as a compile-time instrumentation — to observe the components of the source code such as procedures, loops, etc. During the execution, users can specify "instrumentation action points", similar to breakpoints during correctness debugging, where it will be possible to change the instrumentation parameters dynamically. Annai can be used in tracing or timing mode. It is also possible to estimate the intrusion of tracing for the various components of the source program.

3.5.3. Pablo. In Pablo[28], the source code is instrumented directly by the user with the help of a graphical interface. In case the trace recording frequency is too high, it is automatically reduced or even tracing is automatically replaced by counting. The self defined trace format of Pablo, SDDF, was adopted for several other tools.

3.5.4. PGPVM. Monitoring data is collected by an instrumented version of the PVM communication library which generates traces at the Paragraph format. Post mortem clock synchronization and intrusion removal can be performed if necessary.

3.5.5. Tape/PVM. Tape/PVM[18, 19, 20] is a monitoring tool for PVM programs. The source code is directly instrumented using a preprocessor. A global clock is implemented by software, using the SBA technique (see Section 4). The tracing intrusion can be compensated post-mortem (see Section 5).

3.5.6. XPVM [12]. XPVM is a graphical interface for PVM which includes an instrumented PVM communication library generating trace data which can be used for on-line or post-mortem visualization. Trace data transmission alters the bandwidth available for the communications of the observed applications.

TABLE 3.1
Some existing tracing tools

Tool	**Instrumentation**	**Dynamic filtering**	**Global clock**	**Intrusion compensation**
AIMS	source	yes	synchro	yes
Annai	source, compiler binary	yes		estimate
Pablo	source	yes	no	no
PGPVM	instrumented library	no	post mortem synchronization	yes
Tape/PVM	source	no	post mortem global clock	post mortem
XPVM	instrumented library	no	no	no

4. Global time implementation on distributed memory parallel systems. As mentioned above, many distributed memory systems such as clusters do not have a hardware global clock. Using local clocks to date events results in errors where the sequencing order, derived from the dates of the events, could contradict the causal relationship between these events [14], which could be established using a logical clock [23]. To avoid these errors, it is possible to implement a global time in a distributed memory system by selecting the clock of one of the processors of the system as a reference clock [21]. Equation 3.1 can be derived into:

$$lt_i(t) = \alpha_{i,ref} + \beta_{i,ref} lt_{ref}(t) + \delta_{i,ref}. \tag{4.1}$$

The corrected global time on processor i, $LC_i(t)$ will therefore be estimated as:

$$LC_i(t) = lt_{ref}(t) \approx \frac{lt_i(t) - \alpha_{i,ref}}{\beta_{i,ref}}, \tag{4.2}$$

the value $lt_{ref}(t)$ in equation 4.2 being the reference clock value at time t, such as it can be computed from $lt_i(t)$, provided that $\alpha_{i,ref}$ and $\beta_{i,ref}$ are known. The coefficients $\alpha_{i,ref}$ and $\beta_{i,ref}$ need to be estimated for each of the processors of the system. The method consists in building a statistical sampling of the dates that some events – occurring on the reference processor and whose dates are measured using the reference clock – *would have* on processor i. The events used for these estimates are the receptions of "ping-pong" messages, exchanged between the reference processor and processor i (event R^k_{ref} on Figure 4.1). The statistical method assumes that the communication delays of messages sent by the reference processor to the other processors of the system are the same as the communication delays of the reply messages.

The expectation of the communication delay of the "ping" message is supposed equal to the expectation of the communication delay of the "pong" message. R^k_{ref} is chosen as reference event and its occurrence date, on the clock of processor i, estimated

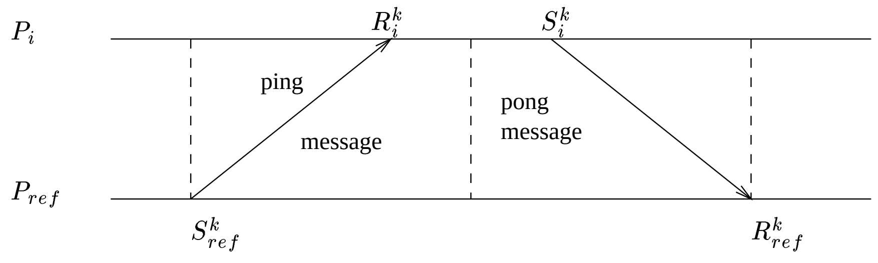


FIG. 4.1. *Data collection process*

by (see Figure 4.1):

$$\widehat{lt_i}(R^k_{ref}) = \frac{lt_i(R^k_i) + lt_i(S^k_i)}{2} + \frac{lt_{ref}(R^k_{ref}) - lt_{ref}(S^k_{ref})}{2}, \tag{4.3}$$

which is identified to $\alpha_{i,ref} + \beta_{i,ref} lt_{ref}(R^k_{ref}) + r^k_{i,ref}$, $r^k_{i,ref}$ being the residu. This method requires a recording of the four dates $lt_i(R^k_i)$, $lt_i(S^k_i)$, $lt_{ref}(R^k_{ref})$, and $lt_{ref}(S^k_{ref})$.

This way a sample of couples $(lt_{ref}(R^k_{ref}), \widehat{lt_i}(R^k_{ref}))$ is obtained and the coefficients $\alpha_{i,ref}$ and $\beta_{i,ref}$ are computed using linear regression techniques.

The SB (*Sample Before*) technique consists in computing the offsets and drifts of the clocks of all the processors of the system, relatively to the reference clock, before the execution of the traced application. It is then possible to give a global date to the monitored events, as soon as they are recorded. However, when an application takes a long time to execute, which is quite frequent for time-demanding parallel applications, the errors arising from approximating the drifts cannot be neglected (when they reach the order of the communication delays).

In this case, it is necessary to use the SBA (*Sample Before and After*) estimation method of $\alpha_{i,ref}$ and $\beta_{i,ref}$, which includes two series of ping-pong messages exchanges, between the reference and all the other processors, *before and after* the execution of the monitored parallel application. The use of two synchronization phases limits the global time extrapolation error. ¿From the values estimated for $\alpha_{i,ref}$ and $\beta_{i,ref}$, it is possible to correct *post mortem* the local dates of each of the events recorded during the execution of the monitored parallel program. The global time estimation error depends on the length of the synchronization phases. By adapting the length of the synchronizations to the duration of the monitored application, it is possible to get rid of all causal incoherencies (see [18] for more details).

5. Modeling and compensation of software tracing intrusion.

Three objects are involved in the modeling and compensation process:

1. the trace file T, reflecting a perturbated application behavior;
2. the "ideal" execution trace T_0, which would be obtained by an ideal, non-intrusive instrumentation;
3. the approximated execution trace T_a, obtained by applying an intrusion compensation model to T.

In case of software tracing, in the absence of a non-intrusive hardware monitor, the only performance index which can be known about T_0 is its execution time. The importance of the intrusion of software tracing in T with respect to T_0 can be evaluated by comparing the respective execution times. Although this intrusion may remain limited to a few percent of the execution time for some program executions tracing only communications, it may become predominant as soon as detailed tracing is required, to trace the most frequently called procedures for example.

The objective of modeling and compensation methods is to transform a trace T, reflecting a perturbated application behavior, into a trace T_a, approximating as much as possible the "ideal" execution trace T_0, which would be obtained by an ideal, non-intrusive instrumentation. Only direct perturbations are taken into account in the trace correction methods. The principle of these methods is to correct post mortem the dates of the traced events to approximate the dates that these events would have if the direct perturbations caused by software tracing were negligible.

Such a correction methods must take into account the causal dependencies between events occurring in different processes, resulting from synchronizations or communications between these processes. In [22], Malony gives a correction method for several synchronization primitives such as barrier, semaphore, etc. In the remaining of this section, we present how this method was adapted by É. Maillet to the asynchronous communications of PVM in the *Tape/PVM* tracer [20].

5.1. Notations. The direct perturbation α is the cost in time to generate a single event and is assumed to be constant and localized at the instrumentation point (see Figure 5.1). α is assumed constant for ease of presentation only. In practice, α is likely to depend on the size of the event (i.e. its number of attributes). Generating an event for an action of interest A consists in reading the clock to get the date $t(e)$ of the start of the action, and in storing the event attributes after the end of the action. We assume that all the overhead α consists of storing the event with its attributes and that it is located *after* $t(e)$. If A is an action which has a measurable duration (e.g. blocking receive primitive), the execution time of A can be either part of the attributes of the event, or two events can be generated, one at start of A the other at end of A. This depends on the implementation of the tracing tool.

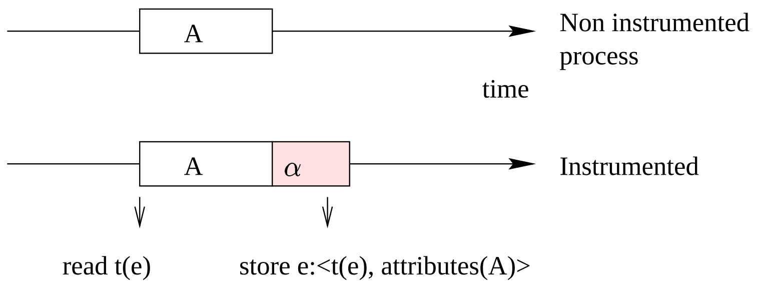


FIG. 5.1. *Model of elementary intrusion cost*

5.2. Case of independent sequential processes. This is the case of a process registering local events, between two consecutive communication events (see Figure 5.2). The approximated date $t_a(e^i_k)$ of the k^{th} registered event of process i can be estimated as:

$$t_a(e^i_k) = tb^i_a + (t(e^i_k) - tb^i) - acc^i,$$

with:

tb^i_a**:** approximated base date of process i. This date is used for each date correction. It is initialized at the date of the first recorded event $t(e^i_0)$. Later on it is recomputed after each communication (see below).

acc^i: accumulated perturbation of process i between the base event and the current event. For an independent sequential process (see Figure 5.2), the value of acc^i when recording the k^{th} event is $(k-1)\alpha$.

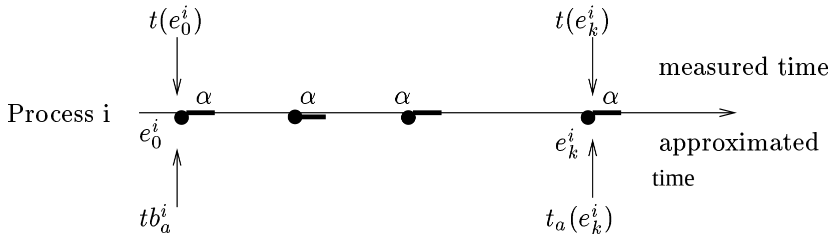
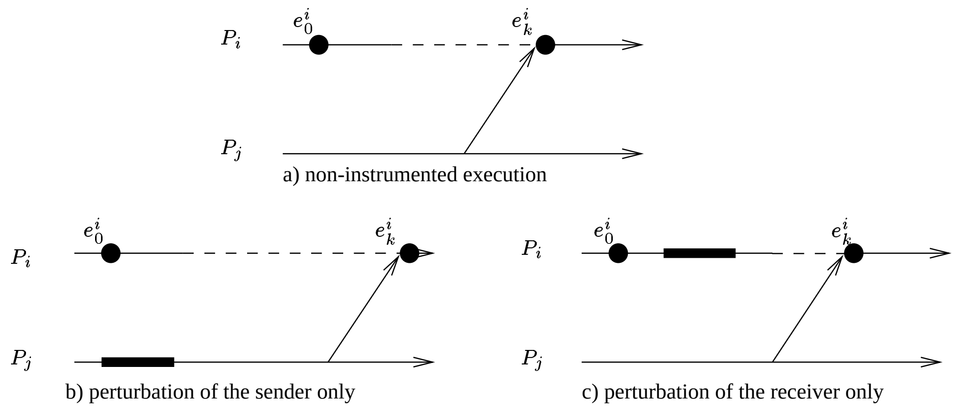


FIG. 5.2. *Perturbation compensation on a sequential process*

5.3. Non-blocking send and blocking receive communication primitives. In presence of communications, it is no longer true that the perturbation at a given event e is the accumulation of all direct instrumentation delays from beginning of the process. The base date needs to be reset after each communication.

5.3.1. Resetting of the base date. In Figure 5.3, process P_i performs a blocking receive primitive of a message sent by P_j. Figure 5.3.a represents non traced execution. In Figure 5.3.b, the perturbations of the sender P_j delays message emission, thus increasing the blocking delay of the receiver P_i. In Figure 5.3.c, perturbations of the receiver P_i delay its posting of the request, thus reducing, or even eliminating the blocking delay. A new base date needs to be recomputed after each communication or synchronization. The base date computation algorithm, in case of non-blocking send and blocking receive primitives, such as the **pvm_send** and **pvm_receive** of the PVM communication library [32], is presented in the following.



P_i
e_0^i
e_k^i
P_j
a) non-instrumented execution
b) perturbation of the sender only
c) perturbation of the receiver only

FIG. 5.3. *Synchronization through a blocking reception primitive: importance of the time base*

5.3.2. Base date computation. The application level reception event can be decomposed into three different "sub-events":

SR: start of blocking receive by the receiving process.

ER: end of blocking receive after message delivery, resuming of the computation.

B: instant at which the message becomes available in the receiver's system buffers, i.e. the soonest possible instant at which the receiver can extract the message.

Information on the **B**, **SR**, and **ER** events is supposed to be stored altogether with the attributes of **ER** (they actually form one single event), which explains why a single overhead α is taken into account in the perturbation compensation algorithm (see Figure 5.4 and Figure 5.5).

Two different cases are possible, depending on whether the message arrives on the receiving node before or after the reception request **SR** of the receiving process:

1. **Before the receive request**

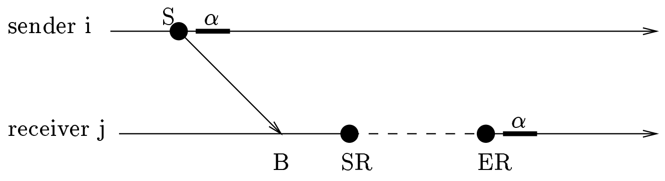


FIG. 5.4. *Message arrived* before *the posting of the receive request*

$$t_a(ER) = t_a(SR) + DC,$$

DC being the delay of processing the incoming data (copying into a new active receive buffer in case of PVM).

2. **After the receive request**

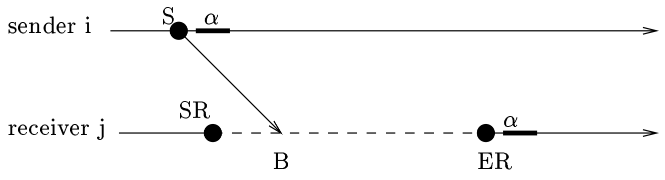


FIG. 5.5. *Message arrived* after *the posting of the receive request*

$$t_a(ER) = t_a(B) + DC.$$

Assuming that neither the message transmission time $(t(B) - t(S))$ nor the message processing delay **DC** are affected by tracing, we get:

$$t_a(B) = t(B) - t(S) + t_a(S).$$

In both cases, the base date tb_a^j of the receiving process is reset to the approximated date of the ER event, after the communication:

$$tb_a^j \leftarrow t_a(ER) \quad acc^j \leftarrow \alpha.$$

The main problem with this algorithm is that the value of $t(B)$ cannot be measured since it is an event at the communication system level and tracing is performed at the application program level of abstraction. É. Maillet describes in [20] an algorithm allowing the correction of the date of event **ER**, depending on the relative positions in time of **B** and **SR**. In some cases, $t_a(ER)$ can be computed without needing to estimate $t_a(B)$. In the few remaining cases, an estimate of the date $t_a(B)$ can be computed from a measure of the date of $t_a(S)$, by applying an approximate cost model of the communications on the monitored parallel system.

5.4. Limits of perturbation compensation methods for programs behaving non deterministically. Perturbation compensation methods do not change the causal dependency relation between events and therefore do not take into account potential behavioral changes of the traced application coming from tracing. Such methods are not applicable to traces of applications using non-deterministic communication primitives. In PVM for instance, a task may request a message of any type from any other task using a **pvm_recv**(-1,-1) function call. The "execution path" of a traced execution of such a program might differ from the "execution path" of a non-traced program execution. In the example of Figure 5.6, the execution of process P_1 was heavily perturbated by tracing. If the emission date of the message sent by process P_1 to process P_2 were naively corrected, its transmission time being unaffected by tracing, the order of reception of messages emitted by processes P_1 and P_3, by process P_2, would be reversed with respect to the actual reception order of the traced execution. However, perturbation compensation methods cannot change the order of reception of messages in the corrected trace since the effects of such a change on the remaining of the traced execution could not be deduced from the traces. The only possibility to correct traces of non deterministic programs is therefore to apply a conservative approximation which keeps the order of message receptions of the corrected trace unchanged with respect to the non corrected trace.

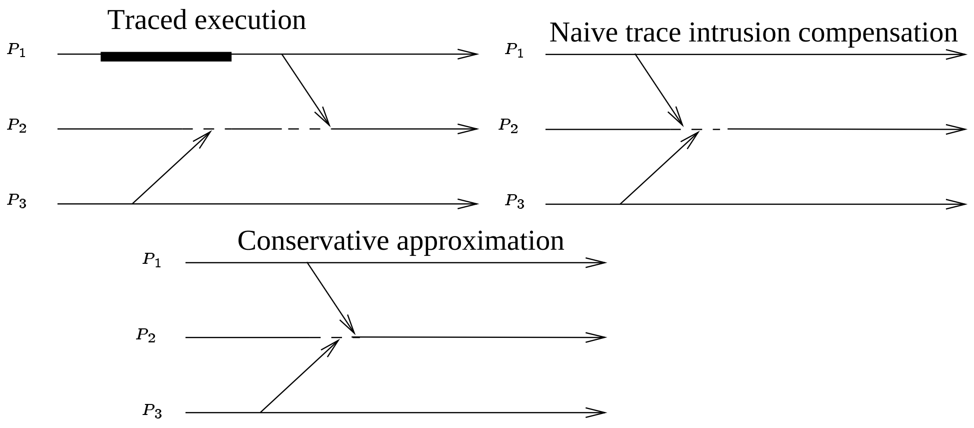


FIG. 5.6. *Order of message reception changed by tracing*

A possible solution to the problem of non-determinism is to use a *deterministic replay* [15] mechanism, when intrusively tracing an application for performance data. Limited control information is recorded during an initial *record* execution, the intrusion caused by this recording being usually very low [5]. This information is used by subsequent *replay* execution to guarantee determinism with respect to the initial *record* execution. If performance traces are collected during a *replay* execution, a perturbation compensation method, similar to the one described above, can be used to correct on line [16] or off-line [33] the intrusion of performance tracing. In the latter case, the method aims at constructing an approximated execution trace T_a, as close as possible from the trace that would be obtained by a non-intrusive tracing of the initial *record* execution.

6. Interaction with the operating system. Although tracing is well suited to capturing phenomena occurring at the application level such as communication delays, it may fail capturing inefficiencies of loaded multi-users systems. The reason is that tracing does not make possible to distinguish between the *ready* and *active* execution states of processes: once a process has been started, only explicit process suspensions at the application level, for example waiting for a message reception, can be detected by tracing at the application level. However, if the observed process is running on top of a multi-users operating system, it may occur that it gets suspended without any relation with an application-level event. This is the case if a higher priority process becomes activable or if the observed process has exhausted its time slice. In such cases the measured duration of the activity of some tasks is likely to exceed the actual duration of these tasks.

Such phenomenon can be captured if tracing at the application level can be coupled with measurement data gathered at the operating system level. One possible approach is to relate the counting information maintained by the operating system during the execution to the information deduced from tracing at the application level[26]. Then it becomes possible to evaluate the percentages of activity and idleness during the periods where tracing can only indicate that a thread was activable.

A similar problem may occur when other monitoring techniques are used and similar solutions may be designed. For example, in Paradyn, some performance data kept by the operating system can be mixed with the counters and timers generated from the monitoring of the observed programs. For example, the cumulative number of page faults is read before and after a procedure call to approximate the number of page faults taken by that procedure [24].

7. Conclusion. This chapter presents the issues of collecting monitoring data for performance debugging of parallel programs. Most monitoring tools for parallel programs are event driven, the most general event-driven monitoring technique being tracing. Among the possible tracing techniques, software tracing is deemed the most portable and widespread, although it requires to solve two difficult problems in order to obtain high quality traces: providing a precise global clock in distributed-memory systems and being able to limit or compensate the intrusion of tracing. The latter problem is specially difficult in case of programs behaving non deterministically. For such programs, a monitoring approach using execution replay techniques seems promising. Tracing alone is not sufficient to detect performance problems arising from the interaction of a parallel application with an underlying software layer such as the operating system. A possible approach to detect such problems is to combine tracing at the application level with time-driven monitoring at the operating system level of abstraction. The investigation of monitoring tools based on this approach seems a promising research track.

REFERENCES

[1] J. Bull, *A hierarchical classification of overheads in parallel programs*, in Proceedings of First IFIP TC10 International Workshop on Software Engineering for Parallel and Distributed Systems, Chapman Hall, 1996, pp. 208–219.

[2] C. Clémençon, A. Endo, J. Fritscher, A. Müller, and V. Wylie, *Annai scalable run-time support for interactive debugging and performance analysis of large-scale programs*, in Proc. Euro-Par'96 Parallel Processing, L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, eds., no. 1123 in LNCS, 1996.

[3] K. M. Decker and B. J. N. Wylie, *Software tools for scalable multi-level application engineering*, International Journal of Supercomputer Applications and High-Performance Computing, 11 (1997), pp. 236–250.

[4] L. DeRose, Y. Zhang, and D. Reed, *Svpablo: A multi-language performance analysis system*, in 10th International Conference on Computer Performance Evaluation - Modelling Techniques and Tools - Performance Tools'98, Palma de Mallorca, Spain, Sept. 1998, pp. 352–355.

[5] A. Fagot and J. Chassin de Kergommeaux, *Systematic assessment of the overhead of tracing parallel programs*, in Proceedings of the 4th Euromicro Workshop on Parallel and Distributed processing, PDP'96, E. Zapata, ed., Braga, Jan. 1996, IEEE/Computer Society Press, pp. 179–186.

[6] G. A. Geist, M. Heath, B. Peyton, and P. Worley, *Picl, a portable instrumented communication library*, TN 37831-8083, Oak Ridge National Laboratory, Oak Ridge, USA, 1991.

[7] S. Graham, P. Kessler, and M. McKusik, *gprof: A call graph execution profiler*, in Proceedings of the SIGPLAN'82 Symposium on Compiler Construction, ACM, 1982, pp. 120–126.

[8] W. Gu, G. Einsenhauer, K. Schwan, and J. Vetter, *Falcon: On-line monitoring for steering parallel programs*, Concurrency: practice and experience, 10 (1998), pp. 699–736.

[9] M. T. Heath and J. A. Etheridge, *Visualizing the Performances of Parallel Programs*, IEEE Trans. Softw. Eng., 8 (1991), pp. 29–39.

[10] R. Hofmann, R. Klar, B. Mohr, A. Quick, and M. Siegle, *Distributed performance monitoring: Methods, tools, and applications*, IEEE Transactions on Parallel and Distributed Systems, 5 (1994), pp. 585–598.

[11] J. Hollingsworth, J. Lumpp, and B. Miller, *Techniques for performance measurements of parallel programs*, in Parallel Computers Theory and Practice, T. C. et al, ed., IEEE Computer Society Press, 1995.

[12] J. Kohl and G. A. Geist, *The pvm 3.4 tracing facility and xpvm 1.1*, in Proc. of the 29th. Hawai International Conference on System Sciences, 1996.

[13] E. Kraemer and J. T. Stasko, *The visualization of parallel systems: An overview*, Journal of Parallel and Distributed Computing, 18 (1993), pp. 105–117.

[14] L. Lamport, *Time, Clocks and Ordering of Events in a Distributed System*, Communications of the ACM, 21 (1978), pp. 558–565.

[15] T. LeBlanc and J. Mellor-Crummey, *Debugging Parallel Programs with Instant Replay*, IEEE Transactions on Computers, C-36 (1987), pp. 471–481.

[16] E. Leu and A. Schiper, *Execution replay : a mechanism for integrating a visualization tool with a symbolic debugger*, in CONPAR 92 - VAPP V. Second Joint International Conference on Vector and Parallel Processing, L. Bougé, M. Cosnard, Y. Robert, and D. Trystram, eds., vol. 634 of Lectures Notes in Computer Science, Springer-Verlag, 1992.

[17] T. Ludwig, M. Oberhuber, and R. Wismueller, *An open monitoring system for parallel and distributed programs*, in Proc. Euro-Par'96 Parallel Processing, L. Bougé and et al., eds., vol. 1123 of Lecture Notes in Computer Science, Springer, 1996.

[18] É. Maillet, *Tape/PVM: An efficient performance monitor for PVM applications*, user guide, LMC-IMAG, B.P. 53, F-38041 Grenoble Cedex 9, France, 1994. Available at ftp://ftp.imag.fr/imag/APACHE/TAPE.

[19] É. Maillet, *Issues in performance tracing with Tape/PVM*, in Proceedings of EuroPVM'95, HERMES (ISBN 2-86601-497-9), 1995, pp. 143–148.

[20] É. Maillet, *Traçage de logiciel d'applications parallèles : conception et ajustement de qualité*, PhD thesis, Institut National Polytechnique de Grenoble, Sept. 1996. In French. Available at ftp://ftp.imag.fr/pub/APACHE/THESES/.

[21] É. Maillet and C. Tron, *On Efficiently Implementing Global Time for Performance Evaluation on Multiprocessor Systems*, Journal of Parallel and Distributed Computing, 28 (1995), pp. 84–93.

[22] A. D. Malony, A. Reed, and H. Wijshoff, *Performance Measurement Intrusion and Per-*

turbation Analysis, IEEE Transactions on parallel and distriuted systems, 3 (1992).

[23] F. Mattern, *Virtual time and global states in distributed systems*, in Proc. Int. workshop on Parallel and Distributed Algorithms, M. Cosnard, P. Quinton, M. Raynal, and Y. Robert, eds., North Holland, 1989.

[24] B. Miller et al., *The paradyn parallel performance measurement tool*, IEEE Computer, (1995).

[25] O. Nickolayev, P. Roth, and D. Reed, *Real-time statistical clustering for event trace reduction*, The International Journal of Supercomputer Applications and High Performance Computing, 11 (1997), pp. 144–159.

[26] F.-G. Ottogali and J.-M. Vincent, *Mise en cohérence et analyse de traces multi-niveaux*, Calculateurs Parallèles, 11 (1999), pp. 211–227. In French.

[27] D. Reed, *Experimental analysis of parallel systems: Techniques and open problems*, in Proc. 7th Int. Conference on Computer Performance Evaluation, G. Haring and G. Kotsis, eds., Vienna, Austria, May 1994, Springer Verlag.

[28] D. Reed, K. Shields, W. Scullin, L. F. Tavera, and C. Elford, *Virtual reality and parallel systems performance analysis*, IEEE Computer, (1995).

[29] D. A. Reed et al., *Scalable Performance Analysis: The Pablo Performance Analysis Environment*, in Proceedings of the Scalable Parallel Libraries Conference, A. Skjellum, ed., IEEE Computer Society, 1993, pp. 104–113.

[30] M. Sarukkai and A. Malony, *Perturbation analysis of high level instrumentation for spmd programs*, ACM SIGPLAN Notices, (1993), pp. 44–53.

[31] W. Stevens, *Unix network programming*, Englewood Cliffs, NJ: Prentice-Hall, 1990.

[32] V. Sunderam, *PVM: A Framework for Parallel Distributed Computing*, Concurrency: Practice and Experience, 2 (1990), pp. 315–339.

[33] F. Teodorescu and J. Chassin de Kergommeaux, *On correcting the intrusion of tracing non deterministic programs by software*, in Euro-Par'97 Parallel Processing, LNCS 1300, Springer-Verlag, Aug. 1997, pp. 94–101.

[34] J. Trocki, *Mon: Service monitoring daemon.* http://www.kernel.org/software/mon/.

[35] J. C. Yan, *Performance tuning with aims — an automated instrumentation and monitoring system for multicomputers*, in Proc. of the Twenty-Seventh Annual Hawai Conference on System Sciences, IEEE Computer Society Press, 1994, pp. 625–633.

[36] J. C. Yan and M. A. Schmidt, *Constructing space-time views from fixed size trace files – getting the best of both worlds*, in Parallel Computing: Fundamentals, Applications and New Directions, E. D'Hollander, G. R. Joubert, F. J. Peters, and U. Trottenberg, eds., vol. 12 of Advances in Parallel Computing, Amsterdam, Feb. 1998, Elsevier, North-Holland, pp. 633–640.