The Journal of Supercomputing, , 1–29 ()

Supporting Scalable Performance Monitoring and Analysis of Parallel Programs

KEI-CHUN LI AND KANG ZHANG {danielli, kang}@mpce.mq.edu.au
Department of Computing, Macquarie University, NSW 2109, Australia

Editor:

Abstract. Tools for performance monitoring and analysis become indispensable parts of programming environments for parallel computers. As the number of processors increases, the conventional techniques for monitoring the performance of parallel programs will produce large amounts of data in the form of event trace files. On the other hand, this wealth of information is a problem for the programmer who is forced to navigate through it, and for the tools that must store and process it. What makes this situation worse is that most of the time, a large amount of the data are irrelevant to understanding the performance of an application. In this paper, we present a new approach for collecting performance data. By tracing all the events but storing only the statistics of the performance, our approach can provide accurate and useful performance information yet require far less data to be stored. In addition, this approach also supports real-time performance monitoring.

Keywords: performance monitoring, program instrumentation, trace data organisation

1. Introduction

Monitoring the performance of a parallel program at run-time is an extremely useful feature for any powerful performance tool. The existing run-time performance monitors can only provide and display limited information for performance tuning. One of the major reasons is that, with run-time monitoring, there is a limited time for data collection and analysis. The longer time the data collection takes, the more intrusive it is to the behaviour of the program. The intrusiveness of monitoring may result in a change of the program behaviour.

Another major difficulty in providing useful performance information at run-time is due to the large amount of performance data. To understand the performance of a parallel program, it is necessary to collect full-sized data sets running on a large number of processors. The conventional techniques for monitoring the performance of parallel programs generate large volumes of performance data. It is common to generate a few megabytes of data per second of program run when collecting basic performance information at the granularity of procedure and system calls. This estimate is based on generating traces of procedure and system call events. For a large scale parallel computer, consisting of say, 1000 nodes, this amount of data would be impractical to collect. Therefore, this creates a dilemma. Identifying a performance bottleneck necessitates collecting detailed information, yet collecting a large amount of data can overload the memory or trace files.

To reduce the volume of data to be collected, a performance tool should collect only the information necessary to explain the performance of the program. However, deciding which data are relevant during a program's execution is difficult. On the other hand, this wealth of data is a problem for the user to navigate through, and for the tool to process during run-time monitoring.

This paper introduces an instrumentation approach for data collection that supports run-time analysis of parallel programs' performance. The approach can maintain the ordering of program events and collect timing information of interesting events. The paper also presents a trace data organisation scheme that provides performance information rapidly during the execution of the program but require far less amount of data to be stored. The collected trace data are organised into different levels, ranging from the program itself to an individual communication statement, so that the user can tune the program in a top-down fashion, focusing on those areas that have the greatest impact on the performance. Furthermore, this technique is also suitable for post-mortem performance analysis.

2. Related Work

We now review the previous work related to the two major concerns mentioned above, i.e. dynamic program instrumentation methods for reducing intrusiveness and trace data organisation methods for reducing the data volume and increasing the scalability.

2.1. Instrumentation

Gupta and Spezialetti [8] present an approach for dynamically minimizing the intrusive effect of monitoring and ensuring that the execution selection of a non-deterministic event is the same for the un-instrumented and instrumented versions of a program. They assume an execution model in which a distributed computation is composed of multiple processes located at different processing sites. The processes' communication is via message passing and pending messages are maintained in a message pool for each process. Non-determinism arises due to the order in which the messages arriving at a process from different sources are handle.

By restoring the message pool and message ordering, this approach can undo the intrusive effects of monitoring at run-time. It estimates the time at which the messages would have been sent and received if no monitoring actions were performed. On the basis of these estimates each process can restore the message pool by waiting for late messages or ignoring messages that have become available early. In addition, the messages can also be appropriately ordered. The major limitations of this system are: (1) it assumes that all the messages are sent in asynchronous mode; (2) it assumes that the messages that have been sent can arrive and be stored in the destination's message pool before the receiving process is ready to receive; and (3) for a non-deterministic receiving event, it requests the monitor to wait until all the messages from the remote sites to arrive.

Cai and Turner [16] introduced a logical clock approach, which monitors the behaviour of occam programs on the transputer system. In this approach, not only the partial ordering of events, but also the communication ordering on each process is preserved. In permitting non-deterministic communication (ALT construct in occam), the monitor has to wait until it is safe to allow the communication to occur. That is, after a process has been granted the permission to receive a message, there will be no other message destined to that process with an earlier logical time. However, this may introduce deadlocks, not normally existing in the programs without monitoring. To avoid the deadlocks introduced by the logical clock approach, Cai and Turner developed a *carrier-null message* algorithm [2], based on the deadlock avoidance algorithm called *null-message* [3] developed by Chandy and Misra.

2.2. Scalability

Performance Metric Predicate Library (PMPL) [5] attempts to solve the problem of performance data size. PMPL can be used with existing trace monitors such as AIMS [18, 19] and PICL [6]. It uses user-defined predicates (PMPs) to control the output of event trace data during the application's execution. These predicates test the state of the program performance; if the performance does not fall into a specified tolerance, the monitor will output the performance history leading up to the point of performance degradation. The major limitations of this system are that it requires the user to manually specify predicates and its additional perturbation to the application's performance imposed by Performance Metric Predicates.

Dynamic Instrumentation [9, 13] controls the amount of data to be collected by deferring the insertion of instrumentation probes until the application is in execution. It can insert or change instrumentation at any time during execution by modifying the application's binary image. Only the instrumentation probes that are required for the currently selected analysis or visualisation are inserted. The approach periodically samples detailed information stored in event counters and timers. These intermediate values provide information useful for making decisions on how to change the instrumentation to meet the need. The volume of data collected is controlled by collecting only the information needed at a given moment, and by controlling the sampling rate. Although this approach provides great flexibility, it requires many decisions to be made on what to collect, and when to collect them. The quality of the collected performance data depends on the sampling rate, the time required to make the decisions, and the correctness of the decisions.

The MPP Apprentice performance tool [17] is designed to help users to tune the performance of their Cray T3D applications. It collects statistics for each section of code, summarises the data on each processor, and displays the summary across all processors. During program execution, the time and pass count for each code block are summed within each processor and kept locally in each processor's memory. This enables the MPP Apprentice tool to handle very long-running programs without increasing the demand for the processor memory. At the end of program execution, or when requested by the user, a global reduction of the statistics for each code object across the processors is done, and a run-time information file is created. The MPP Apprentice tool then post-processes this file and provides performance information to the user. The major limitation of this tool is that it does not support real-time performance analysis.

Quartz [1] is a profiling tool for tuning parallel program performance on shared memory multiprocessors. The philosophy underlying Quartz was inspired by the sequential UNIX tool gprof [7]: to appropriately direct the attention of the programmer by efficiently measuring just those factors that are most responsible for performance and by relating these metrics to one another and to the structure of the program. The primary limitations of the profiling tool are its dependence on an external sampling task and the potential errors inherent in sampling. Another limitation of this tool is that it focuses on finding CPU time bottlenecks only.

3. Design Goals

Having compared the existing approaches and identified their limitations, we now discuss four major goals in the design of our run-time performance monitor.

Goal 1: Providing accurate performance information

This goal includes minimising the intrusiveness of data collection and accurately calculating the performance (e.g. timing) information.

Goal 2: Keeping trace data in a manageable size

As mentioned before, for a large scale parallel computer, it could be impractical to collect the amount of performance data. The problem also happens for small but long running programs which also generate a large amount of trace data. Therefore, the second goal of the monitor is to maintain the performance data set in a manageable size without losing the characteristics and usefulness of the data.

Goal 3: Providing performance information rapidly

Parallel programs incur overhead in many different ways, such as synchronisation, load imbalance, and communication. A prompt assessment of how processing time is spent on each of these aspects is useful during performance tuning of parallel programs. The third goal of the monitor is to store the performance data in a form so that the performance information, including the statistics of CPU time (for load imbalance), transmission time (for communication), and blocking time (for synchronisation), can be easily derived and rapidly provided.

Goal 4: Providing hierarchical data and relating back to the source program

The performance tuning process should be associated with the source program. If a means of focusing on successively detailed regions of the program is provided, one can proceed in a hierarchical manner with respect to the program code, by studying the code at various levels of detail. This helps to systematically focus on the region of code causing the performance problem. A previous informal study [10] shows that users generally start by taking a high level view of the performance of their applications, and isolate the source of their program's poor performance. This process continues until they understand their program's performance well enough to start tuning it. Therefore, the fourth goal of the monitor is to provide the user with different levels of performance statistics, ranging from the whole program to a single communication statement. The performance statistics are organised according to the hierarchical structure of the program.

To achieve these four goals, the key questions to be answered are:

- how to minimize the intrusiveness of data collection,
- what data should be collected, and
- how the collected data are organised.

The next section discusses various approaches to performance data collection that achieve some of the goals, and presents our design decisions that have been made in order to achieve all the goals. Sections 5 and 6 present our approaches to intrusion minimisation and reduction of the trace data volume.

4. Data Collection Methods and Design Decisions

There are three basic approaches to collecting performance data: counting, statistical sampling, and event tracing, each representing a different tradeoff among information volume, potential instrumentation perturbation, accuracy, and implementation complexity.

4.1. Counting

The first approach records the number of times each interesting event occurred. In Unix grof [7], a monitoring routine is called in the prologue of each profiled procedure. This monitoring routine updates the count of calls to the procedure from its caller. Given both the count and total time, one can compute the average execution time. This approach significantly reduces the volume of performance data collected. However, it records the number of times an event occurred, but not where or why the event occurred. It therefore loses the important information about the event context. This approach achieves Goals 2 and 3, but cannot achieve Goal 4. It also achieves Goal 1 to some extent since it collects only the counting information which introduces little intrusion.

4.2. Statistical sampling

Another approach to data collection is statistical sampling that involves periodic sampling of program resources, such as the currently active procedures, to gauge the program performance. Unix grof [7] periodically interrupts the program to sample the program counter, thereby estimating the execution time of each procedure. In Quartz [1], a set of processors execute the program and maintain their states in a shared memory by special code executed during thread operations and at procedure entries and exits. These states are then sampled by a dedicated processor that does not participate in executing the program. Crovella and LeBlanc's predicate profiling tool [4] uses a similar approach.

The primary limitations of these approaches are their dependence on the external sampling task and potential errors inherent in sampling. While sampling procedures provide only an estimate, it is accurate just where it needs to be - for those routines which spend most of the program's time and the accuracy of the result depends on the sampling rate. On the other hand, if the total program execution time is too low, the sampling error may be intolerable. Ponder and Fateman [14] identified a number of other limitations of statistical sampling approaches. The statistical sampling approach achieves Goals 2 and 3, but cannot achieve Goals 1 and 4.

4.3. Event tracing

Event tracing is the most general and flexible approach to data collection. It generates a sequence of event records. Each event record consists of an encoded instance of an action and its attributes, typically including the following:

- what action occurred (i. e., an event identifier),
- the time when the event occurred,
- the location where the event occurred (e.g., a line number), and
- any additional data that define the event circumstances.

Not only does event tracing identify what happened and where it happened, the event timestamps impose an order on the events that defines the control flow and interactions between system components. One of the strengths is that judiciously chosen events can identify most performance problems and some correctness problems. Task schedules may be directly obtained from trace files, and an immense amount of flexibility is available in assigning and interpreting time-stamped events.

Event tracing subsumes counting and sampling; one can compute times and counts from trace data. For example, given a trace of procedure entries and exits, one can compute the total number of calls to each procedure by counting the number of instances of each event type, as well as the total procedure execution times by matching procedure entry and exit events, computing the difference in their event times, and adding the difference to a running sum for that procedure. In addition, dynamic procedure call graphs and space-time diagrams can be drawn for program visualisation using the trace data. With tracing, one can capture components interactions, dynamic behavior, transients, as well as load imbalances due to message waiting.

Since the performance information is retrieved from trace files, this technique is usually used in post-mortem performance tuning tools, such as AIMS [18, 19], PICL [6], and Pablo [15]. These tools present information from the perspective of time, and try to give the user an idea of what was happening in every processor in the system at any given moment. Event traces have great potential to help the user to understand the program behaviour, but also have some serious problems as discussed below.

To understand the performance of parallel programs, it is necessary to collect a large amount of data running on a large number of processors. But the volume of data, which is proportional to the number of processors, the frequency of trace points, and the length of program execution, limit the scalability of the monitoring system.

For example, AIMS generates 3.6 megabytes of trace file for a parallelised version of ARC2D (an NAS parallel benchmark) executed on 64 processors in two seconds [5]. For a massively parallel computer, the amount of trace data is too large to collect and maintain for realistic programs. With continued increases in the number of processors and the sizes of applications, event tracing is not a solution that scales well. On the other hand, the data volume is also difficult to manage and display, and the user is left to interpret all the events happening on thousands of processors.

Therefore, this approach achieves Goal 4, but cannot achieve Goals 2 and 3. With Goal 1, it collects accurately most of the performance information, such as counting and timing, but it generates serious intrusion especially in monitoring large or long-running parallel programs.

4.4. Design decisions

From the above discussion, there is no single approach that can achieve all the four goals. Therefore, we developed an approach which is a hybrid of counting and event tracing that provides a low data volume of counting with detailed tracing. We instrument the most interesting events (e.g. communications) and collect the event traces during the execution of the program. The collected data are not only summarised in counts and times as in the counting approach, but the information such as the locations of the events and the break-down of the timing are also stored as in the event-tracing approach. Moreover, to support the user's cross-referencing, our approach uses the knowledge of the program structure to relate the performance characteristics back to the source program.

Data to be collected

When collecting performance data of parallel programs, interesting events usually include procedures and communications. To support tuning long-running programs, it is reasonable to assume that most of the execution time is spent on loops. Hence loops are treated as interesting events in our approach. To provide more detailed timing information for procedures, procedure calls are also considered in our monitor as interesting events. Thus, the statements of communications, procedure calls, procedures, and loops are instrumented for performance data collection.

Organisation of performance data

After collecting the performance data for each interesting event, the monitor will accumulate the collected execution times and store them in a data structure dynamically generated during program execution. It also updates the count for such an event. The locations of all the monitored events in the source program are also stored in the data structure, and are used to relate the performance characteristics of the interesting events back to the source program. Moreover, the break down of the timing is also recorded in more details. For example, the monitor records not only the total execution time and count of calls to a procedure, but also the break down of the information by different procedure calls. All of these are processed on-the-fly during the execution of the program.

Cross-reference between performance data and source program

To relate the performance characteristics back to the source program, we must maintain a link between the control flow of the computation and the performance data. As a means of representing the original source program and its control flow, we provide an abstraction of the intermediate program representation using program graphs.

The following two sections present our approaches, based on the above design decisions, for program instrumentation that aims at minimising intrusiveness and for data organisation that aims at reducing the volume of trace data.

5. Intrusion Minimisation

A fundamental problem with any performance monitoring tool is the intrusiveness of data collection and its perturbation to the program behaviour. Intrusiveness may alter the timing of events in the program in an arbitrary manner and can lead to a change of ordering of the events. This means that the result of monitoring with an intrusive monitor can only be taken as an approximation of what has happened in an unmonitored program. It may also hide deadlocks existing in the program or even create new deadlock situations [16].

5.1. Perturbation of Instrumentation

Time characteristics

An intrusive monitor may alter the time characteristics of events in a program in an arbitrary manner. The delays introduced by the monitor in the execution of processes can change the relative timings of their executions. Thus not only are the overall execution times of the processes affected, but also are other time characteristics such as waiting for messages, processor idle, and so on.

Figure 1 is a space-time diagram showing the effect of intrusiveness in terms of timing. P_1, P_2, and P_3 are three different processes. Without instrumentation, the sending event E_a in $P1$ is ready to send a message to $P2$ at time T_1. The receiving event E_b in P_2 is ready to receive the message from P_1 at time T_2. Assume that T_2 is earlier than T_1 by t, the ready time of E_b is t earlier than E_a. Assuming there is one probe inserted before and one probe inserted after each event, and there are two instrumented events before E_b. With instrumentation, there is one probe before E_a and five probes before E_b. For the simplicity of calculation, assuming the delay time of every probe is the same and equal to t, the ready time of E_a becomes $3t$ earlier than E_b. Therefore, the instrumentation has altered the ordering of the events, and the timing information, such as synchronisation delay, is no longer the same as that without monitoring.

Execution selection

The second major problem with program instrumentation is that it may result in a change of the execution selection in a non-deterministic communication which has multiple legal executions for a given input. A non-deterministic message reception is supported using the any-way branch. The receiving process may accept a message

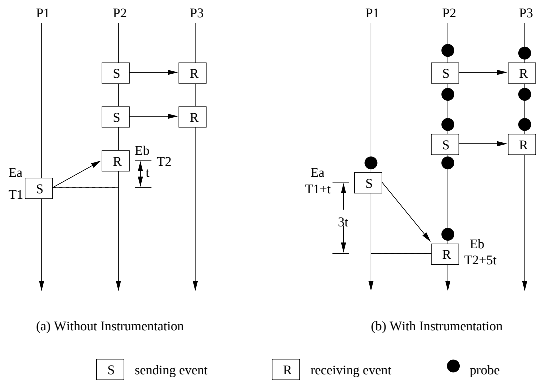

Figure 1. Intrusiveness to the time characteristics

from any other process node (i.e., $P_1, P_2, ..., P_n$) and selects the path along the any-way branch (i.e., $E_1, E_2, ..., E_n$) which corresponds to the sender of the message. The criteria for message selection are not expressed in the application by the user and may differ in different implementations. In distributed systems, this selection is based upon criteria such as the order in which the messages arrive, priorities of the senders, fairness criteria such as selecting a message from a process that has been least recently considered, or a purely random selection may be made, etc.

We assume that the receiver selects the path along the any-way branch which corresponds to the sender whose message is the first ready to send to the receiver. Therefore, the selection of a particular execution is made at run-time and is typically influenced by the timing of various events. Since the monitoring actions alter the execution time, the likelihood of the selection of various legal executions may also be altered.

5.2. Instrumentation Methods

The existing instrumentation methods can be classified into three groups: software, hardware, and hybrid approaches. The software approach is to add a set of instructions (probes) in an application program to enable the runtime collection of all useful information about system behaviour. This approach offers the advantage of being fairly independent of the target architecture. It also provides information at a high level of abstraction, directly useable by designers or by a graphical display tool.

There are many ways of realising software instrumentation. Probes can be inserted manually by editing the source code, automatically by a pre-compiler, by linking instrumented libraries, by modifying the linked executable or by modifying the operating system. According to the programming stages that the probes are inserted, the software instrumentation can be further divided into *source instrumentation*, *library instrumentation*, *object instrumentation*, and *kernel instrumentation*.

The main shortcoming of software instrumentation is related to the time overhead introduced by the information gathering. To limit the effect of intrusion, instrumentation is usually limited to events whose observation is considered essential.

Other instrumentation methods reply on extensive hardware support and do not have a noticeable effect on the behaviour of the program being observed. The hardware approach involves connecting probes to the system hardware in order to observe its behaviour without disturbing it. However, it has a number of limitations. Firstly, they provide very low-level data. Usually considerable processing and complicated mechanisms are required to provide application level monitoring information from machine level data. Secondly, they form the least portable class of monitoring mechanisms. Thirdly, the design of a hardware monitor can be complicated with the use of pipelining and on-chip cache for increasing the throughput of microprocessors and also an increase in the integration of various functional units (e.g., floating point units and memory management units) which makes monitoring difficult.

Hybrid monitoring is a compromise between software and hardware monitoring approaches. It is based on the addition of a few instructions in programs in order to select adequate information useful to explain the behaviour off-line. The information is collected with a specific device and transmitted to the host system which interprets it off-line and then displays the result. The approach has the advantages of the two other approaches because it gives information at a high abstraction level and introduces a low overhead. However, the major limitation of hybrid instrumentation is that it is often dedicated to specific hardware architectures. Therefore, standard interfaces are important for supporting instrumentation portability.

To achieve high portability and flexibility, we choose a source instrumentation method using software monitoring. The following subsection presents our instrumentation method using virtual time for monitoring parallel message-passing programs.

5.3. Virtual Clock Approach

In order to achieve Goal 1 stated in Section 3, i.e. accurately capture the performance information such as synchronous delay, transmission time, and to introduce minimal intrusiveness to the program behaviour, we use a virtual clock approach to collect the performance data. Our approach is based on Cai and Turner's logical clocks approach [16] and able to handle parallel programs that involve nondeterministic communications. However, it differs from Cai and Turner's approach in three key aspects [**?**, 20]:

- A new deadlock avoidance method is used to speed up the realisation of the *global waiting dependency*, and decrease the amount of control messages and time for the monitor to select the earliest available sending process. Moreover, it is more suitable for parallel systems containing nodes connected within a network that provides point-to-point communications.

- Our approach is designed and implemented for monitoring both the synchronous and asynchronous communication functions in CMMD on the CM-5. It is, however, general enough to be applied to other parallel platforms.
- No special monitor process is necessary in our approach. The monitor is implemented as a run-time library, which contains a set of routines (i.e. probes) to be inserted into the source code.

We assume an execution model in which a parallel computation is composed of multiple processes located on different processors. The processes communicate via message passing and non-determinism arises due to the order in which communication requests to a process from different sources are handled at the destination process.

5.3.1. Virtual time In order to keep the execution selection and the collected timing characteristics of events the same as those of the original un-instrumented program, the estimate of time spent on monitoring activities is maintained by a virtual clock in each process. Using the estimates for all the processes involved, we can infer the times at which various actions would have occurred if no monitoring had been performed. Next we define various times used in our approach.
Definition 1 The *local time* at processor P_i during the execution of an instrumented process, denoted by LT_i, is given by (RT_i, I_i), where RT_i is the *real time* of P_i obtained from P_i's internal physical clock and I_i is the current *intrusion time* of P_i. The *virtual time* VT_i, which is the estimate of the real time at P_i during the execution of the original un-instrumented program, is given by $RT_i - I_i$.

For a given application process, the local value of the intrusion time indicates the delay due to monitoring activities in the execution of the process.

5.3.2. Execution control In order to preserve the timing characteristics, and keep the execution selection unchanged, the monitor needs to take over the control of the process which is ready to communicate with other processes and determine the timing and ordering of communication events.

When monitoring a deterministic synchronous communication event, the monitor will postpone the permission of the communication until it detects the arrival of the partner communication event. When monitoring a deterministic asynchronous communication event, the monitor will permit the asynchronous function to execute without waiting for the readiness of its communication partner. In permitting a non-deterministic communication, the monitor has to wait until it is safe to allow the communication to occur. After the receiving process œhas been granted permission to receive a message, there should be no other process that will send a message to that process with an earlier virtual time.

The timing and counting information is collected by subroutine calls (probes) in a Data Collection Library inserted into the source code by a pre-processor. Probes are inserted before and after each procedure call, communication, and loop statement. For the procedure, a probe will be inserted after the heading of the procedure.

When monitoring communication events, the monitoring functionality is divided into two parts, *fore monitoring* activities and *back monitoring* activities. The fore monitoring is responsible for recording the synchronisation time and controlling the inter-process communication. Since the fore monitoring activities are performed before a communication statement, the monitor can control the occurrences of communication events. When the monitor notices that its communication partner is ready to communicate or is able to decide which sender should communicate, the monitor can calculate the time spent on synchronisation and permit the communication to start. After the execution of the communication, the back monitoring simply calculates the time spent on the message transmission.

5.3.3. Preservation of time characteristics Consider again the example shown in Figure 1, the use of virtual time to preserve the ordering and execution times of events are illustrated in Figure 2. RT_1, RT_2, and RT_3 are the real times of processes P_1, P_2, and P_3 respectively. VT is the virtual time. The sending event E_a and receiving event E_b arrive at real times $T_1 + t$ and $T_2 + 5t$ respectively. Since the times spent on monitoring activities before E_a and E_b are t and $5t$ respectively, the intrusion time when E_a arrives is t and that for E_b is $5t$. Therefore, E_a arrives at virtual time $(T_1 + t) - t = T_1$ and E_b arrives at virtual time $(T_2 + 5t) - 5t = T_2$. Based on the virtual time measurement, the correct ordering of events is preserved, that is, E_a arrives after E_b, and synchronous delay which is equal to $T_1 - T_2 = t$ can also be accurately collected.

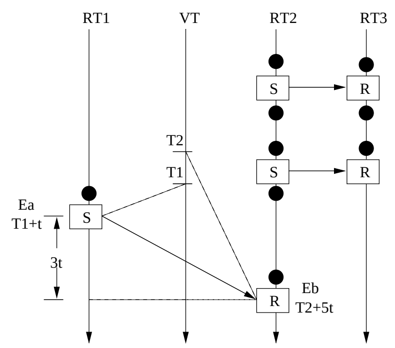

Figure 2. Preservation of time characteristics

5.3.4. Preservation of execution selection The monitor also uses the virtual time to control the occurrence of inter-process communication to keep the execution selection unchanged. To achieve this, when a non-deterministic receiving event arrives, the monitor makes the decision as to which inter-process communication should happen next based on virtual times, rather than real times, of other processes. It delays the occurrence of a communication if it is aware that there is at least one possible candidate process for that communication which is running in an earlier virtual time. In this way, the communication is prevented from occurring either too early in virtual time or too late in real time. Therefore, although the real-time execution of a process is slowed down by the monitoring activities, the behaviour of the program is unchanged.

Figure 3 shows an example how the monitor uses the virtual time to keep the execution selection unchanged, where VT_1, VT_2 and VT_3 are virtual times of processes P_1, P_2 and P_3 respectively. E_b and E_d are the non-deterministic receiving events in P_2. E_a and E_c are the sending events in P_1 and P_3 respectively. Because of the delays introduced by the monitor routines, E_a, E_b, E_c, and E_d arrive at real times q, p, r, and s instead of original times T_1, T_2, T_3, and T_4 when executing the uninstrumented program. The relationships between these times are $s > r > q > p$ and $T_4 > T_1 > T_2 > T_3$.

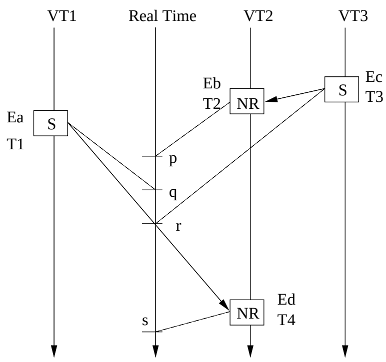

Figure 3. Preservation of execution selection

5.3.5. Deadlock avoidance The above virtual clock approach may introduce a deadlock in a non-deterministic communication. Consider the example shown in Figure 4, in which the W end of the line is connected to a waiting process, and the N end of the line is connected to a non-waiting process. In this example, process P_1 is waiting for input from processes P_0 and P_3. P_0 is ready to output to P_1 at virtual time 30. P_3 is waiting to receive from process P_2 and P_2 is waiting to receive from P_1. It is assumed that the virtual time of P_3 is less than 30, thus, P_3 might have a chance to become ready to output to P_1 before 30. Therefore, according to the virtual clock approach, the monitor cannot permit P_1 to receive from P_0 and no message can be sent out to P_2. Thus a deadlock occurs, as P_1 is waiting for P_3, P_3 is waiting for P_2, and P_2 is waiting for P_1.

Without monitoring, this deadlock situation will not happen, since P_1 will select P_0 for communication. So, the above deadlock is introduced by the monitoring policy of the virtual clock approach. It is clear that this kind of deadlock should be avoided. Cai and Turner have developed a carrier-null message algorithm [2],

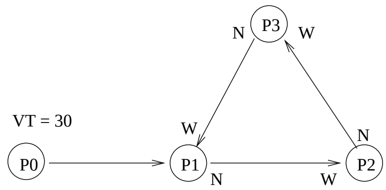

Figure 4. Deadlock creation

which is based on the null-message algorithm [3]. Both of these algorithms avoid deadlocks by transmitting null messages to announce the absence of real messages and to advance the simulation time or logical time of each process in the waiting dependency loop.

However, in parallel systems (e.g. CM5, SP2) with tens or hundreds processors, using the carrier-null message algorithm, the amount of protocol messages will overload the system with message traffic. We have developed a new deadlock avoidance method [**?**, 20] which detects the global waiting dependency and allows the earliest available sending process (P_0 in the above example) to send its message. To obtain the information on the global waiting dependency, the monitor for a non-deterministic receiving event collects the timing and state information from all the other processes. After identifying a dependency loop, the monitor selects the earliest ready process to permit its communication with the non-deterministic receiving event. The approach uses a simple algorithm that introduces little amount of extra control messages.

5.3.6. Performance of instrumentation We compared the times of different stages of a test program executing with monitoring and without monitoring in order to measure the overhead of the monitor and the accuracy of the times collected by the monitor [12]. The test program is a matrix multiplication parallel program written in C and CMMD. In this program, a master process executes on processor 0 and a number of worker processes execute on processors 1 to 31. In the beginning, the master process initialises the matrix A and matrix B. It then sends different portions of matrix A and matrix B to the worker processes. All the worker processes perform their own calculations after receiving the data and send their results back to the master process. The master process then merges the result of the multiplication between matrix A and matrix B to form matrix C.

Figure 5 shows the tolerance of the timings of different stages and indicates that the maximum tolerance is 3.6%. We also obtained the monitoring overhead that is 2% of the total execution time of the program without monitoring. By inserting different delays into the monitoring routines to evaluate the variation of the tolerance of the timings, we observed the maximum tolerances that ranged between 1.8% and 12%. The results shown that such tolerances are not proportional to the increase of delays inserted in the monitor.

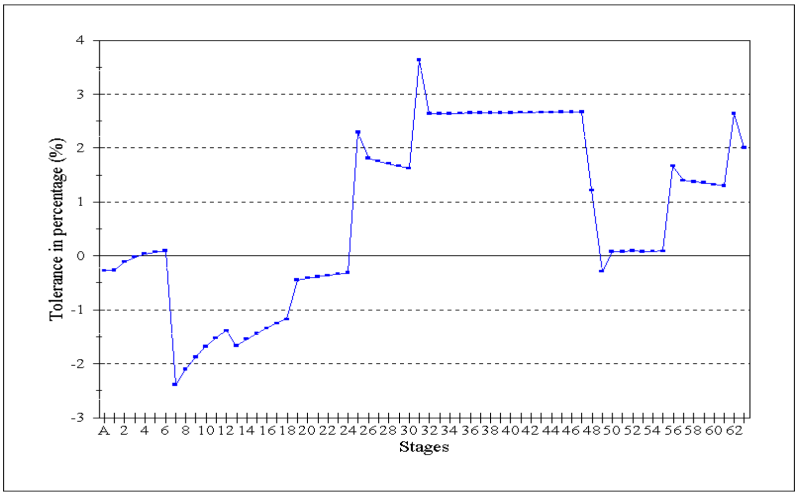

Figure 5. The tolerance of the timings

6. Reducing Trace Data Volume

To reduce the amount of trace data to be maintained, we organise the performance data collected by program instrumentation in two major forms of data structure, i.e. static tables and dynamic records. Figure 6 illustrates how the performance information is organised. During the execution of a parallel program, the performance analyser generates a performance report based on the information in static tables and dynamic records and on the user's input from the user interface. At the end of the program execution, the monitor may also save the dynamic records to a system disk for post-mortem analysis.

The combined use of static tables and dynamic records ensures that the necessary performance information is structurally maintained. The storage required for such data structures is manageable in size and does not increase proportionally as the program size or running time.

6.1. Program graphs

To be able to represent and interpret the performance of a program, and relate the performance back to the source program, we use an abstraction called *program graph*, as depicted in Figure 7. A program graph consists of nodes that represent major control points in the program such as procedure calls, loops, and procedure headers. The nodes are organised according to the hierarchical structure of the parallel program.

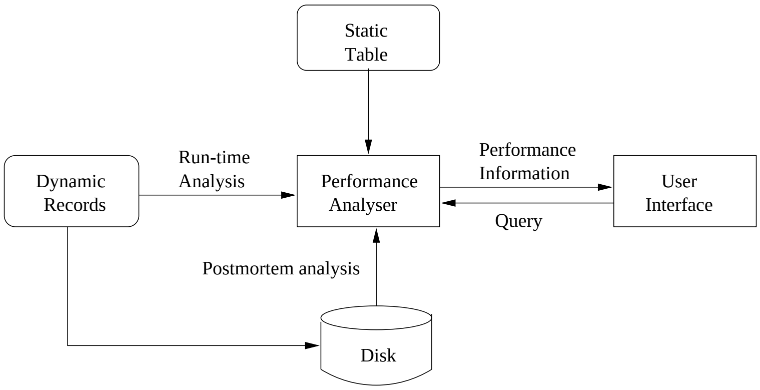

Figure 6. The generation of performance information

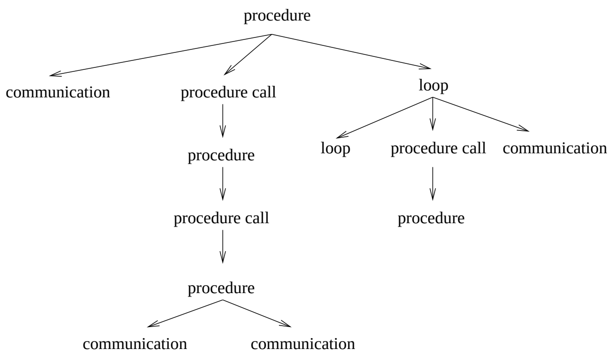

Figure 7. The program graph of a message passing parallel program

The top level of the program graph is the main procedure. The second level consists of the monitored events in the main procedure, such as communication events, procedure calls and loops. For a loop event, if there is a nested loop, procedure call, or communication event within the loop, a lower level description of such an event will be attached to the loop event. For a procedure call event, the called procedure will be attached to the calling event. A program graph is implemented as a number static tables in our monitor. The static tables are constructed during program instrumentation by the pre-processor before the execution of the program since their entries can be determined from the program text. Figure 8 shows the structure of the pre-processor.

6.2. Static tables

Static tables are used for storing the hierarchical relationship of the program structures and statements. An interesting program structure or statement is identified

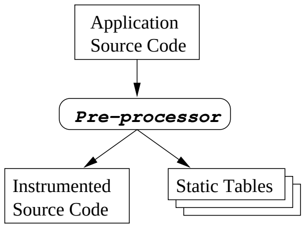

Figure 8. The pre-processor for program instrumentation

by its line number in the program. Tables T_{pl}, T_{pcom}, and T_{pc} store the information of loop structures, communication statements and procedure calls inside a procedure respectively. Tables T_{ll}, T_{lcom}, and T_{lc} store the information of loop structures, communication statements and procedure calls inside a loop structure respectively. Figure 9 shows the format of static tables, each having the following fields:

- l_p is the line number of a procedure statement. Since the data presented in the monitor is primarily based on procedures, this field is the key in Tables T_{pl}, T_{pcom}, and T_{pc}.
- l_l is the line number of a loop statement. This field is the key in Tables T_{ll}, T_{lcom}, and T_{lc}.
- l_c is the line number of a communication statement.
- l_{pc} is the line number of a procedure call statement.
- l_{pi} is the line number of a procedure statement which is invoked in l_{pc}.
- l_{il} is the line number of a loop statement which is nested in another loop.

	Fields					
Table	l_p	l_l	l_c	l_{pc}	l_{pi}	l_{il}
T_{pl}	$\surd$	$\surd$				
T_{pcom}	$\surd$		$\surd$			
T_{pc}	$\surd$			$\surd$	$\surd$	
T_{ll}		$\surd$				$\surd$
T_{lcom}		$\surd$	$\surd$			
T_{lc}		$\surd$		$\surd$	$\surd$	

Figure 9. The format of static tables

6.3. Dynamic records

Dynamic records are generated by the monitor during the execution of a parallel program. They store the statistics of interesting events. In our approach, interesting events include procedures, loops, and communications. A dynamic record is allocated for each procedure, loop and communication statement, whose structure is shown in Figure 10.

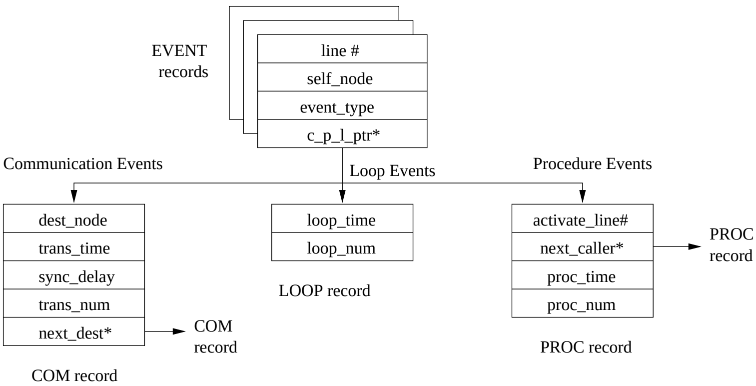

Figure 10. The structure of dynamic records for storing interesting events

A list of dynamic records named EVENT records are allocated at run-time to different interesting program events of a parallel program. Each EVENT record stores the statistics of an interesting program event, and is a four tuple,

$$(line\#, self\_node, event\_type, c\_p\_l\_tpr*),$$

where

- $line\#$ is the line number of the first statement of the program event.
- $self\_node$ is the identifier of the process which executes the program event.
- $event\_type$ is the type of the program event, which can be a communication, a loop, or a procedure.
- $c\_p\_l\_ptr*$ is a pointer pointing to the corresponding records of the type $event\_type$.

To relate the performance characteristics back to the source program, an interesting program event is identified by its line number in the program. For a communication event, $line\#$ stores the line number of the communication statement. For a loop or a procedure event, $line\#$ stores the line number of the first statement of the event. Since the performance data may be collected from different processes,

$self\_node$ identifies the process in which the program event is executed. We use multiple EVENT records to store performance data of different program events. According to the type of a program event stored in $event\_type$, $c\_p\_l\_ptr*$ points to the corresponding records. For a communication event, $event\_type = COM$, $c\_p\_l\_ptr*$ points to a list of COM records. For a loop, $event\_type = LOOP$, $c\_p\_l\_ptr*$ points to a LOOP record. For a procedure, $event\_type = PROC$, $c\_p\_l\_ptr*$ points to a list of PROC records. We now describe the design of these records for respective program events.

Procedure events

A large complex program is usually composed of many small procedures that implement abstractions of the program. To direct the user's attention to the location that is causing the performance problem, the statistics of a procedure such as the count of the procedure's invocations and execution time should be provided. On the other hand, it is useful to know in each procedure the statistics of the other procedures it invokes. Consider the following example:

```
procedure X
{
     ...
     call Y
     ...
}
```

Assume that procedure X is responsible for 90% of the execution time in a program, and the call Y statement is responsible for 80% of the execution time. We should obviously focus our attention on procedure Y rather than procedure X. For this reason, it is necessary to detect a situation in which the time is dispersed among several procedures.

Unix gprof [7] can provide such information. It gathers three pieces of information during the program execution: call counts, execution time of each profiled routine, and the arcs of the dynamic call graph traversed during the program execution. By post-processing these data, gprof builds a dynamic call graph where nodes represent the routines and directed arcs represent the calls to the routines. It then propagates the times along the arcs of the graph to the routines that invoke them. Since it assumes that each call to a routine takes the average amount of time for all calls to that routine, the caller r is accountable for $T_s \times C_{rs}/C_s$, where T_s is the total time spent by the callee s, C_{rs} is the number of calls from a caller r to s, and C_s is the number of calls to routine s. This assumption is not always true since the execution time of a routine may depend on the parameters passed when the procedure is invoked.

To provide accurate timing in each routine and the times spent by the routines that it invokes, we separately store the execution information of a procedure for different callers. Since our approach provides run-time performance information during the execution of a parallel program, rather than building the dynamic call graph after the execution of the program, a list of PROC records are dynamically allocated for each procedure during the execution to store the statistics of different procedure call statements. Each PROC record in the list stores the statistics of a particular call statement, and is a four tuple,

$$(activate\_line\#, next\_caller*, proc\_time, proc\_num),$$

where

- *activate_line#* is the line number of the statement that calls the procedure.
- *next_caller** is a pointer linking the PROC records in the list.
- *proc_time* is the accumulated execution time of the procedure called by the statement whose line number is stored in *activate_line#*.
- *proc_num* counts the number of executions of the procedure called by the statement whose line number is stored in *activate_line#*.

However, the above arrangement would not work for programs containing recursive calls. Figure 11 shows two simple examples. Because the relationship between the caller and callee is mixed up, the time is propagated from one statement to another in a cycle. Our solution is not to allocate any PROC record to a recursive call. For example, in Figure 11a, no PROC record is allocated to the call X statement. We only provide the counting and timing information of procedure X, just like that the call X statement does not exist. This information is stored in *proc_time* and *proc_num* of PROC record of procedure X with external call statements. In the case of Figure 11b, we treat procedures X and Y as a single procedure Z. All the callers to procedures X and Y become callers to procedure Z. This arrangement results in the situation like in Figure 11a, so that no PROC record will be allocated to either call X or call Y statement. The counting and timing information of procedure Z is stored in the list of PROC record of procedure Z EVENT record with external call statements.

```
procedure X
{
     ...
     call X
     ...
}

(a) A recursive routine
```

```
procedure X             procedure Y
{                       {
     ...                     ...
     call Y                  call X
     ...                     ...
}                       }

(b) Mutually recursive routines
```

Figure 11. Program examples containing recursive calls

Therefore, we provide the user with simple and accurate performance information for recursive procedures instead of presenting overlapping and complicated timing information.

Communication events

Communication is one of the most important activities that may be optimised to improve the performance of parallel programs. In order to observe the communication behaviour, we collect the measurements such as transmission time, synchronous delay and the count of the executions for each communication statement. A communication statement may change its destination during the execution of a parallel program. The following is an example of the CM-5's CMMD send statement whose destination changes from 1 to numworkers.

```
for (dest = 1; dest <= numworkers; dest++)
     CMMD_send_block(dest, 1, &buf, bsize)
```

To provide more detailed timing information of a communication event, such as the time spent on communication between two processes, a list of COM records are dynamically allocated to store the statistical information for all different destinations of each communication statement in the program. A COM record is a five tuple,

$$(dest\_node, trans\_time, trans\_num, sync\_delay, next\_dest*),$$

where

- *dest_node* is the identifier of the process which is the destination of the communication.
- *trans_time* is the accumulated message transmission time of the communication event.
- *trans_num* counts the number of executions of the communication event.
- *sync_delay* is the accumulated synchronisation time of the event.
- *next_dest∗* is the pointer linking to the next COM record in the list.

Loop events

For many long-running programs, most of the execution time is spent on loops. We collect the time and count of executions for every loop event in the program. A LOOP record is dynamically allocated to store the statistical information for each loop event in the program. It is a two tuple, $(loop\_time, loop\_num)$, where

- *loop_time* is the accumulated execution time of the loop event.
- *loop_num* counts the number of executions of the loop event.

6.4. Results

The following are two case studies that indicate some results of the evaluation of the effectiveness of the above data structure in performance interpretation and reduction of trace volumes.

6.4.1. Performance Interpretation To describe and explain the data structures used in our approach and demonstrate how they can provide performance statistics, an example parallel program is used and shown in Figure 12.

In the example program, L1 to L15 are the line numbers of the program statements. The main procedure main in L1 consists of two procedure call statements in L2 and L3 which call procedures P1 and P2 in L4 and L6 respectively. Therefore, two records [L1 L2 L4] and [L1 L3 L6] are included in Table T_{pl}. The loop event loop1 in L11 has a nested loop loop2 in L13, and record [L11 L13] is included in Table T_{ll}. Since the communication statement E1 in L9 is inside procedure P3 in L8, record [L8 L9] is in Table T_{pcom}. On the other hand, the communication statement E2 in L12 is inside loop1 in L11, record [L11 L12] is in Table T_{lcom}. Figure 13 shows the static tables generated for the example program.

Since procedure P3 can be called by the statements in L7 and L5, two dynamic records, i.e. a PROC record with $activate\_line\# = L5$ and a PROC record with $activate\_line\# = L7$, are in the list referenced by $c\_p\_l\_ptr*$ of the EVENT record of procedure P3. One LOOP record is also allocated to each of loop1 and loop2.

The following performance information is derived from the static tables and dynamic records.

Total execution time of the whole program

Since the line number of the main procedure is L1, the record with $line\# = L1$ is searched in EVENT records. The main procedure is the first procedure to execute, so only a PROC record with $activate\_line\# = 0$ is in the list referenced by $c\_p\_l\_ptr*$ of the matching EVENT record. The value in *proc_time* of the PROC record is the total execution time of the whole program.

Total message transmission time of the whole program

All the records with *event_type* equal to communication events are searched in EVENT records. For each of the matching records, the values in *trans_time* of all COM records referenced by the record's $c\_p\_l\_ptr*$ are accumulated. The sum of these accumulated values is the total message transmission time of the whole program.

Total communication synchronisation time of the whole program

The calculation is the same as for the total message transmission time of the whole program except that the values in *sync_delay* rather than in *trans_time* are accumulated.

Total execution time of all the procedure calls in the main procedure

Using $activate\_line\# = L1$ as the key, two records [L1 L2 L4] and [L1 L3 L6] are found in Table T_{pc}. This means that there are two procedure call statements in the main procedure. The execution time of the first call statement is *proc_time* of the PROC record whose $activate\_line\# = L2$ referenced by $c\_p\_l\_ptr*$ of the record whose $line\# = L4$. The method to find the execution time of the second procedure call statement is the same as to find the first. The sum of these two values is the total execution time of all the procedure calls in the main procedure.

Total execution time of the procedure call in L5

The calculation is the same as the above without searching in Table T_{pc} and only one procedure call statement is searched in the dynamic records.

```
            ...
L1          main()
            {
                ...
L2              call P1;
L3              call P2;
                ...
            }
L4          procedure P1
            {
                ...
L5              call P3;
                ...
            }
L6          procedure P2
            {
                ...
L7              call P3;
                ...
            }
L8          procedure P3
            {
                ...
L9              E1;
L10             call P4;
L11             loop1{
                       E2;
L13                    loop2{
                              ...
L14                           call P4;
                           }
                    }
                ...
            }
L15         procedure P4
            {
                ...
            }
```

Figure 12. An example program

Table T_{pl}		Table T_{pcom}	
l_p	l_l	l_p	l_c
L8	L11	L8	L9

Table T_{pc}		
l_p	l_{pc}	l_{pi}
L1	L2	L4
L1	L3	L6
L4	L5	L8
L6	L7	L8
L8	L10	L15

Table T_{ll}		Table T_{lcom}	
l_l	l_{il}	l_l	l_c
L11	L13	L11	L12

Table T_{lc}		
l_l	l_{pc}	l_{pi}
L13	L14	L15

Figure 13. The static tables generated for the example program

Total time and count of procedure P3 executions

Since the line number of procedure P3 is L8, the record with $line\# = L8$ is searched in EVENT records. The sum of the *proc_time* values in all the PROC records that are referenced by $c\_p\_l\_ptr*$ in the matching EVENT record is the total execution time of procedure P3. The sum of the values in *proc_num* is the count of procedure P3 executions.

Total execution time of the communication event in L12

The record with $line\# = L12$ is searched in EVENT records. The sum of the values in *trans_time* and *sync_delay* of all COM records in the list referenced by $c\_p\_l\_ptr*$ in the matching EVENT record is the total execution time of the communication event in L12.

Total execution time of loop1 and the timing information of the events in the loop

The record with $line\# = L11$ is searched in EVENT records. The value in *loop_time* of the LOOP record referenced by $c\_p\_l\_ptr*$ in the matching EVENT record is the total execution time of loop1. Using $line\# = L11$ as the search key, record [L11 L13] is found in Table T_{ll}. This means that there is a loop at L13 nested inside loop1. The method to find the total execution time of loop2 is the same as for loop1. With the same key, record [L11 L12] is found in Table T_{lcom}. This means that there is a communication event at L12 in loop1. The way to calculate the time spent on this communication event was demonstrated before. Using the same key, no record is found in Table T_{pc}, so there is no procedure call statement in loop1.

From the above discussion, we can see that the collected performance data are organised in such a way that they can be related back to the structure of the application program. The performance information can be provided with different levels of performance statistics, ranging from the whole program to a single communication statement. With such information, the user can tune the program in a top-down fashion, focusing effort on those areas that have the greatest impact on the performance.

We now use the program in Figure 12 as an example and assume that procedure P4 consumes most of the program's execution time. Firstly, the user checks the performance of the main procedure and finds that most of the execution time was spent on call P1 and call P2 statements. The performance of procedures P1 and P2 is checked and it indicates that call P3 was the most time-consuming statement. The user then checks the performance of procedure P3. It can be found that most execution time of procedure P3 was spent on the call P4 statement. The user can then conclude that procedure P4 has the greatest impact on the whole performance and should focus his/her attention on procedure P4.

6.4.2. Comparison of Data Sizes The size of the performance data needed in our approach is much smaller than those generated by traditional trace-based systems. Let us consider the following portion of a parallel program written in CMMD and executed on CM-5.

```
47      if (my_address = 5)
48      for (i = 0; i <200; i++) {
49          CMMD_receive_block(4, stype, &offset, intsize);
50          CMMD_receive_block(4, stype, &c[offset][0], Bsize)
51          CAL(offset);
52          CMMD_send_block(6, mtype, &offset, intsize);
53          CMMD_send_block(6, mtype, &b, NCA*dbsize); }
```

Processor 5 repeatedly receives data from processor 4, calls the subroutine CAL for calculation, and sends the result to processor 6. Assuming that the user is interested in the time spent in communication, we compare the volume of data collected by traditional event trace tools and by our approach.

In a traditional event trace tool, monitoring routines are inserted before and after the four communication statements in lines 49, 50, 52, and 53 for collecting the timing information and generating the trace event records. Assume the following format of each event trace record

$$(line, type, source, destination, timestamp)$$

where *line* is the line number of the event in the source program, *type* is the event type, *source* is the process which executes the event, *destination* is the receiving process identifier of the communication event, and *timestamp* is the start time of the event execution. Given that the sizes of the five fields are 2, 2, 4, 4, and 8 bytes respectively, each record occupies 20 bytes. The following event records are created in the first iteration of the for loop in line 48.

(49, 2, 5, 4, 0.308713)
(49, 2, 5, 4, 0.308880)
(50, 2, 5, 4, 0.308905)
(50, 2, 5, 4, 0.484081)
(52, 2, 5, 6, 1.423808)
(52, 2, 5, 6, 1.424025)
(53, 2, 5, 6, 1.424053)
(53, 2, 5, 6, 1.583002)

Since the loop will be executed for 200 times, 1600 ($= 8 \times 200$) event records will be created. That means that the size of the trace data for storing the timing information of these four communication statements will be 31.25 kbytes ($= 1600 \times 20/1024$).

fields	size (in bytes)
$line\#$	2
$self\_node$	4
$event\_type$	2
$c\_p\_l\_ptr*$	4
$dest\_node$	4
$trans\_time$	8
$sync\_delay$	8
$trans\_num$	2
$next\_dest*$	2

Table 1: The sizes of the fields in the EVENT and COM records

In our approach, four EVENT records are allocated for these four communication statements. Since the destinations of these four communication statements are fixed, only one COM record is allocated in each of the lists pointed to by $c\_p\_l\_ptr*$ in each of these EVENT records. Table 1 shows the sizes of the fields in the EVENT and COM records. From the table, we can calculate that each EVENT record occupies 12 bytes (=2+4+2+4) and each COM record occupies 24 bytes (=4+8+8+2+2). The total size of the four EVENT records and four COM records is 144 bytes ($= 4 \times (12 + 24)$). This volume of trace data is significantly less than 31.25 kbytes generated using the traditional approach.

In the traditional event trace tools, the trace data size is increased proportionally to the number of iterations in a loop structure. In our approach, there is no increase in trace data size with increasing number of iterations, so this approach is especially suitable for handling long running programs. On the other hand, the above set of event trace records generated by the traditional event trace tools do not provide the breakdown timing information of the communication, e.g., transmission time and synchronous delay. However, this detailed information can be provided by our approach with a small size of performance data set.

7. Summary: Achieving the Four Goals

We summarise the above approaches by matching them against our original design goals stated in Section 3. We claim that the four goals have all been achieved to a large extent.

As the run-time monitor uses the virtual time instrumentation approach to timestamp and maintain the order of interesting events, it allows the monitor to collect accurate performance data, transfer data for central processing, and perform all the other activities for supporting run-time performance analysis. This achieves the first goal.

Since the monitor traces all the events but stores only the statistics of the performance, it can obtain sufficient performance information yet require far less data to be stored and thus the amount of trace data is controlled in a manageable size. This achieves the second goal.

Stored in a statistical form, the trace data can provide performance information rapidly for run-time monitoring and this achieves the third goal.

Combining the information in dynamic records and static tables, the monitor can provide the user with different levels of performance statistics, from the whole program to a single communication statement. It can appropriately direct the user's attention by efficiently measuring the factors that are most responsible for the performance and by relating these metrics to one another and to the structure of the program. With such information, the user can tune the program in a top-down fashion, focusing effort on those areas that have the greatest impact on the performance. This achieves the fourth goal.

8. Conclusion

The paper has introduced a monitor for supporting run-time data collection and analysis of parallel program performance with high scalability. It assists the user in the location, identification, and resolution of performance problems in parallel programs. The instrumentation approach preserves the ordering of events without being affected by the speed of monitoring activities, and thus supports run-time performance monitoring. Since the raw performance data in each process is summarised as statistics in the process' dynamic records during the program execution, the monitor requires far less data to be stored and maintained. The performance information can be provided easily and rapidly for run-time monitoring. Furthermore, the reduced data volume requires fewer resources and causes less intrusion to the program behaviour. Because the data volume is not proportional to the execution time of the program, the approach can scale well to a growing number of processes and handle long running programs. The lower data volume is manageable for a tool to process and easy for the user to understand.

Performance characteristics are referred back to the source code. The monitor provides users with different levels of performance statistics, relating to the entire program, a procedure, a loop block, a procedure call statement or a communication statement. These statistics are organised according the hierarchical structure of the program. This approach can appropriately direct the attention of the user by efficiently measuring the factors that are most responsible for the performance and by relating these metrics to one another and to the structure of the program. Therefore, users can tune their programs in a top-down fashion, focusing their effort on the areas that have the greatest impact on the performance.

Acknowledgments

The authors would like to thank the anonymous referees for their comments which were very useful in improving the paper.

References

1. T. E. Anderson and E. D. Lazowska, Quartz: A Tool for Tuning Parallel Program Performance, In *Proceedings of the 1990 SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, Boston, May 1990, pp. 115-125.
2. W. Cai and S. J. Turner, An Algorithm for Distributed Discrete-event Simulation - the Carrier-Null Message Approach, In D. Nicol(ed.), *Proceedings of SCS Multiconference on Distributed Simulation*, San Diego, USA, 1990, pp. 3-8.
3. K. M. Chandy and J. Misra, Distributed Simulation: A Case Study in Design and Verification of Distributed Programs, *IEEE Transactions on Software Engineering*, SE-5(5), 1979, pp. 440-452.
4. M. E. Crovella and T. J. LeBlanc, Performance Debugging Using Parallel Performance Predicates, In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, San Diego, California, May 1993, pp. 140-150.
5. C. E. Fineman and P. J. Hontalas, Selective Monitoring Using Performance Metric Predicates, In *Proceedings of the 1992 Scalable High Performance Computing Conference*, Williamsburg, Virginia, April 26-29, 1992, pp. 162-165.
6. G. A. Geist, M. T. Heath, B. W. Petyton and P. H. Worley, A User's Guide to PICL - A Portable Instrumented Communication Library, Technical Report ORNL/TM-11616, Oak Ridge National Laboratory, Aug. 1992.
7. S. L. Graham, P. B. Kessler and M. K. McKusick, gprof: a Call Graph Execution Profiler, In *Proceedings of the SIGPLAN'82 Symposium on Compiler Construction*, ACM SIGPLAN Notices, Vol. 17, June 1982, pp. 120-126.
8. R. Gupta and M. Spezialetti, Dynamic Techniques for Minimizing the Intrusive Effect of Monitoring Actions, In *Proceedings of the IEEE-CS 15th International Conference on Distributed Computing Systems*, Vancouver, Canada, June 1995, pp. 368-376.
9. J. K. Hollingsworth, B. P Miller and J. Cargille, "Dynamic Program Instrumentation for Scalable Performance Tools," In Proceedings of Scalable High-Performance Computer Conference, May 23-25, 1994, pp. 841-850.
10. J. K. Hollingsworth, Finding Bottlenecks In Large Scale Parallel Programs, PhD Thesis, University of Wisconsin-Madison, 1994.
11. K. C. Li and K. Zhang, Instrumenting Parallel Programs Based on a Virtual Clock Approach," In *Proceedings of PDPTA'97 - International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, USA, 30 June - 3 July, 1997, pp. 31-40.
12. K-C. Li, Performance Tuning Support for Message-Passing Parallel Programming, PhD Thesis, Macquarie University, Sydney, Australia, May, 1997.
13. B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall, The Paradyn Parallel Performance Measurement Tools, *IEEE Computer*, Vol. 28(11), November 1995.
14. C. Ponder and R. Fateman, Inaccuracies in Program Profiles, *Software- Practice & Experience*, Vol. 18(5), May 1988, pp. 459-476.

15. D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz and L. F. Tavera, Scalable Performance Analysis: The Pablo Performance Analysis Environment, In A. Skjellum (ed.), *Proceedings of Scalable Parallel Libraries Conference*, IEEE Computer Society Press, Los Alamitos, calif., 1993, pp. 104-113.
16. S. J. Turner and W. Cai., The Logical Clocks Approach to the Visualization of Parallel Programs, In G. Haring and G. Kotsis (eds.), *Performance Measurement and Visualization of Parallel Systems*, Elsevier Science Publishers B.V., 1993, pp. 45-66.
17. W. Williams, T. Hoel and D. Pase, The MPP Apprentice Performance Tool: Delivering the Performance of the Cray T3D, *Programming Environments for Massively Parallel Distributed Systems*, North-Holland, 1994, pp. 333-345.
18. J. Yan, Performance Tuning with AIMS - an Automated Instrumentation and Monitoring System for Multicomputers, In Hesham El-Rewini and B. D. Shriver (eds.), *Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences, Vol. II: Software Technology*, IEEE Computer Society Press, USA, Jan. 1994, pp. 625-633.
19. J. Yan, S. Sarukai, and P. Mehra, Performance Measurement, Visualisation and Modelling of Parallel and Distributed Programs using the AIMS Toolkit, *Software - Practice and Experience*, 25(4), April 1995, 429-461.
20. K. Zhang, C. Sun, and K-C. Li, Collecting Timing Information While Preserving Events Ordering in Parallel Program Instrumentation, Technical Report C/TR95-07, Macquarie University, Dec. 1995.