THE TAPE/PVM MONITOR AND THE PROVE VISUALIZATION TOOL

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

1. Introduction. Performance visualization is a new branch of program development not used in the case of sequential programs. Performance visualization aims at discovering performance bottle-necks in logically correct parallel programs. Such bottle-necks can lead back to previous stages of the parallel program development according to the nature of the bottle-neck. Performance visualization is based on intensive run-time monitoring. In the GRADE parallel program development environment two tools have been integrated in order to realize performance visualization support. These tools are:

- Tape/PVM monitor
- PROVE visualization tool

The current chapter describes these tools and their usage in the GRADE program development environment.

2. Structure of performance visualization systems. Performance visualization systems typically consist of four stages as shown in Figure 2.1. The first stage, the source code instrumentation stage, serves for instrumenting the code with the necessary calls to the operating system or to the underlying extended communication library. The second stage serves to collect trace events during the execution of the parallel program. These collected events are typically stored in one or several log files that are analysed after the execution of the program. This third stage, called trace analysis stage is important in order to establish the physical or logical timing order of the collected events. Finally, the ordered events are visualised by several display views in order to give easily conceivable explanation of the nature of parallel program execution.

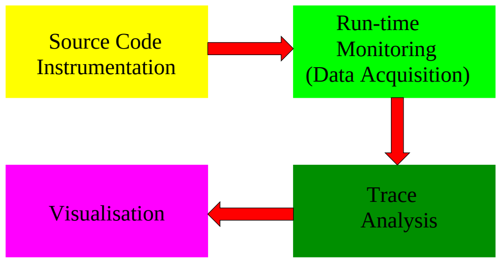

Source Code Instrumentation
Run-time Monitoring (Data Acquisition)
Trace Analysis
Visualisation

Fig. 2.1. *Stages of performance visualization*

Performance visualization systems can be classified according to how they support the four stages of performance measurement. Source code instrumentation is decisive concerning the convenient use of the system or simply from the point of view of usability. Scalability is another important aspect of performance visualization systems.

Scalability is strongly related to the second and third stages of performance visualization. A tool is scalable if it enables the analysis of large, long running parallel programs. It requires special techniques to avoid the generation of too large trace files at run-time. Finally, versatility is another key issue that defines the various display views that the performance visualization system can provide as well as the interoperability with other visualization tools. In the next sections we give a detailed description of all these three aspects and show how they are supported by the Tape/PVM monitor and the PROVE visualization tool in the GRADE programming environment.

3. Source code instrumentation. Source code instrumentation has four major components that should be considered in classifying performance visualization systems:

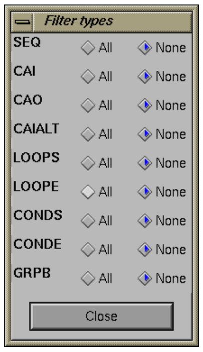
1. Instrumentation mode
2. Filtering
3. Support for monitoring modes
4. Support for click-back facility

The instrumentation mode can be manual or automatic. All the state-of-the-art performance visualization systems provide automatic instrumentation. It means that the user has not to touch the source code, it is the task of the compiling/linking system to transform the original source code or to call extended instrumented communication libraries that support run-time monitoring. In the case of GRADE it is the GRP2C pre-compiler and the GRAPNEL Library that are responsible for supporting automatic code instrumentation. The GRAPNEL Library can call either instrumented PVM or MPI library calls for tracing communication events. It also provides instrumented calls for the graphical blocks of GRAPNEL enabling the GRAPNEL graphical block level event generation and visualization.

Filtering means that the user can specify for the compiling/linking system the interesting program components for which the run-time events should be generated and collected. The lack of such a facility makes the trace file unnecessarily big. Oppositely, filtering makes the trace file customisable to the particular interest of the programmer. The size of the trace file is one of the most crucial problem of performance visualization systems and hence all facilities that can reduce its size are worth supporting. In GRADE, filtering is supported at the level of GRAPNEL as a built-in feature of GRED. In a pull-down menu all the GRAPNEL graphical block types can be filtered. In default, PROVE will collect events on the entry and exit point of each GRAPNEL graphical block. However, if the user is interested for example, only in the SEQ, CAI, CAO and CAIALT blocks, he can filter out all the other graphical blocks (LOOPS, LOOPE, etc.) by the Filter Types pull-down menu as shown in Figure 3.1. Moreover there is a possibility to individually turn on or off filtering on each graphical block of the GRAPNEL program. In this way, the programmer is able to customise the monitoring system to his particular interest and to focus on the events most interesting for him.

Basically two monitoring modes are supported in performance visualization systems. The first one is the collection of individual events, the second one is the collection of statistical information. The former one is supported by Tape/PVM. The current version of PROVE cannot provide statistical information. However, in the new version of GRADE, called P-GRADE (Professional GRADE) both the monitoring system and PROVE will support the collection and visualization of statistical information. The application of statistical information helps in reducing the size of the trace file and hence its usage is highly advantageous.

Although, the click-back facility is one of the most important facilities of performance visualization systems, there are only very few systems that support this


Filter types

SEQ	All	[✓ None]
CAI	All	[✓ None]
CAO	All	[✓ None]
CAIALT	All	[✓ None]
LOOPS	All	[✓ None]
LOOPE	All	[✓ None]
CONDS	All	[✓ None]
CONDE	All	[✓ None]
GRPB	All	[✓ None]

FIG. 3.1. *Filter Types pull-down menu*

feature. The general problem with performance visualization systems is that they provide various graphical views on the program execution based on collected events but they cannot explain which part of the source code is responsible for the generation of the visualised events. The click-back facility applied in advanced tools is a remedy for the problem. It means that when clicking on a visualised event, the system can highlight the part of the source code that is responsible for the generation of the event.

The click-forward facility is the opposite of the click-back facility and it means that when clicking on a source code line, the visualization tool can indicate on its graphical views which events were generated by the selected source code line.

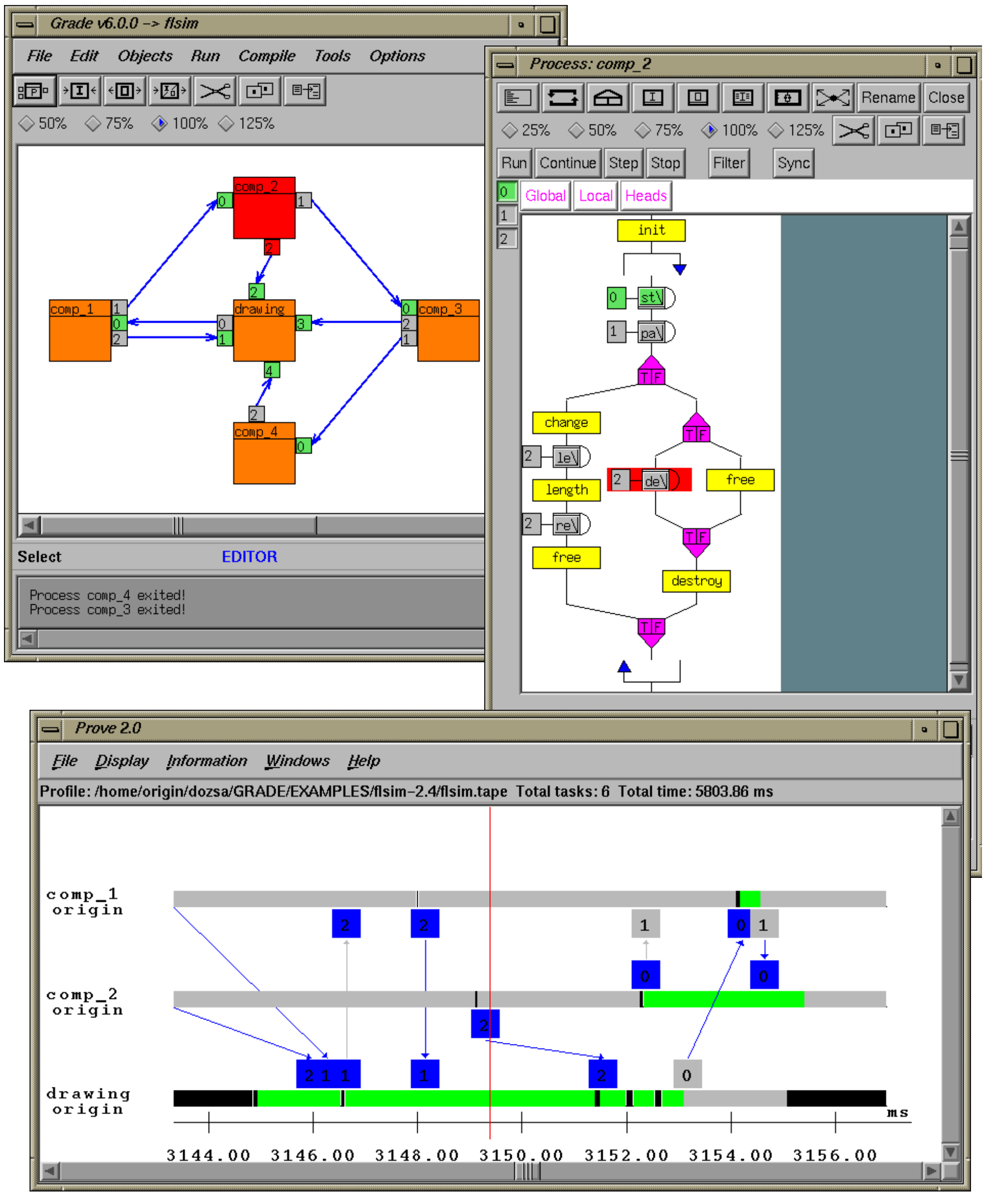
The pair-wise use of click-back and click-forward facilities ensure the perfect identification of the role of program components during the parallel program execution.

The click-back facility of GRADE is illustrated in Figure 3.2. The vertical time bar in the space-time diagram of PROVE in Figure 3.2 is used to realize the click-back facility. The time bar selects the interesting or relevant moment of the execution time. Clicking on the cross point of any process line and the time bar will result in highlighting (making red) the corresponding process in the application window and the corresponding graphical block in the process window. Vice versa, clicking on a graphical block in the process window, the time bar will move in the space-time diagram to the next event that was generated by the selected graphical block.

The click-back facility of PROVE is strongly supported by the Tape/PVM monitor. In order to allow users to quickly find the statement in their source code that generated a particular event, Tape/PVM's events contain the line number of that statement and the identifier of the source code file. In fact, the user's source code is instrumented by Tape/PVM's pre-processor (*tapepp, tapeppf*) which knows the name of the file it processes and the current line number. Each time a probe is inserted into the user's code (at a call of a PVM library function, for instance) the information about file name and line number is given to that probe (in a way similar to Aims [8]). Thus, a visualization tool, like PROVE, can feature source code click-back based on Tape/PVM traces.

4. Data acquisition. Data acquisition is realized by the Tape/PVM run-time monitoring system. Tape/PVM[1] is a tool to generate event traces of PVM applications for post-mortem performance analysis, e.g. discrete event simulation and

[1]The manual and Tape/PVM's distribution are available at ftp://ftp.imag.fr/imag/APACHE/TAPE



```
Process comp_4 exited!
Process comp_3 exited!
```

Profile: /home/origin/dozsa/GRADE/EXAMPLES/flsim-2.4/flsim.tape Total tasks: 6 Total time: 5803.86 ms

FIG. 3.2. *Click-back facility in PROVE*

visualization. It comprises the tool to generate the traces, as well as a utility to transform the traces into the PICL format. It also contains a library of C functions which allows to easily read the generated traces.

Trace generation and post-mortem analysis of traces are two different research areas, each with its own specific problems. The main problem of trace analysis is the design of an appropriate model and a simulator based on that model. The simulator takes a trace file (set of events) as input and reconstructs the successive global states of the system on which the traces were generated. Such a simulator can be coupled with a visualization tool to give a global view of the system under study. However, the simulation is only as accurate as its input - the trace file. Such a trace file has to be representative of what really happened in the parallel system under study. Thus, the main problem in designing a tracing tool is to guarantee the representative quality of the generated traces. The design of Tape/PVM particularly focused on the following two points:

1. Precise, causally coherent event dating,
2. Minimal perturbation of analyzed applications.

Some existing tracing tools for PVM focus on trace visualization and "real-time" interaction rather than on the representative quality of the generated traces. XPVM [3] for example, is a graphical console and monitor for PVM. It uses the event collection mechanism integrated in PVM V3.3.0 or later. Events are routed to XPVM by the PVM kernel during run-time of the instrumented application. Thus, XPVM can update its views in "real time". XPVM can also be used for post-mortem trace analysis using the events of previous executions saved into a file. However, whatever the mode in which XPVM is used, real-time or post-mortem, its traces represent potentially perturbed applications due to on-line event message routing. These messages increase the load of the network which can infer a change in behaviour of the observed application (in fact, many parallel applications are non-deterministic). In addition, the tracing mechanism of the PVM kernel relies on a globally synchronized system clock. Not many systems have a global time reference which is sufficiently accurate to avoid dating anomalies.

In Tape/PVM a non-intrusive, statistical method is used to estimate a precise global time reference [5] (see Chapter 6 for more information on global time implementation in Tape/PVM). Rather than doing post-mortem tachyon removal, an a priori tachyon prevention is achieved through the use of a global time reference. Dated events are causally coherent. However, the estimated global time is only available at the end of the instrumented application which prohibits on-line dating. This is not a drawback because Tape/PVM is intended for post-mortem trace analysis only. In addition to this, at generation, an event is not routed to a central collector task, like in XPVM, in order to avoid additional network load. Instead, the events are stored in local event buffers, which are flushed to local event files. The collection of events into a single file is only done at the end of the user's application to avoid interfering with it.

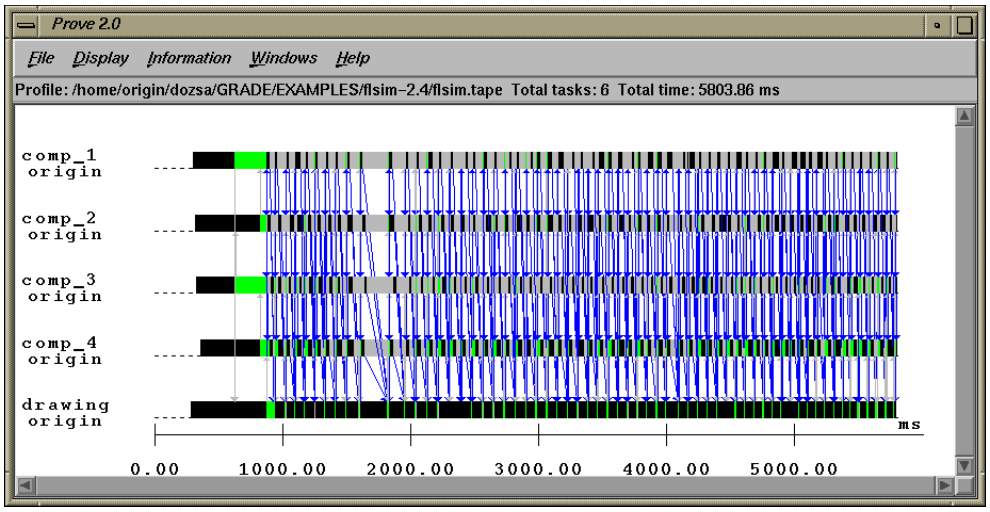
The problem of perturbation of parallel applications due to the presence of a tracing tool is a difficult one. The approach of Tape/PVM is similar to the one adopted in the Aims environment [8]. Although intrusion can be reduced by careful implementation of the tracing tool, it can not be eliminated. The main causes of intrusion are the flushing of local event buffers, the accumulation of the delays of each individual event generation, as well as the additional messages exchanged by the tracing tool. To limit the intrusion due to Tape/PVM the following techniques are used:

- On-line compacting of events. This allows a gain of about 50% with respect to a non-compacted text representation of events. The number of buffer flushes is significantly reduced and so is the perturbation of the application.
- The number of messages exchanged by Tape/PVM is reduced to a minimum. Only events like *PVM_addhost* and *PVM_ kill* which change the configuration of the parallel virtual machine need such additional messages.
- The additional tasks used by Tape/PVM (for global control, for clock synchronisation) are not active while the instrumented user application is running.

5. Trace analysis.

The third stage of performance visualization is devoted to trace analysis. The physical clocks of the processors in a distributed system are usually non synchronized or even in the case of synchronisation they can be drifted to each other. Hence the data collected at run time and time-stamped by the ticks of the physical clocks cannot be considered as strongly and precisely ordered. The first task of the data analysis is to create an at least logical ordering among the collected events. The most frequently used ordering criteria is based on the happened-before relation introduced by [4]. In the GRADE system the Tape/PVM monitor is applied which guarantees the physical ordering of events in the trace file according to a non-intrusive, statistical clock synchronisation algorithm [5].

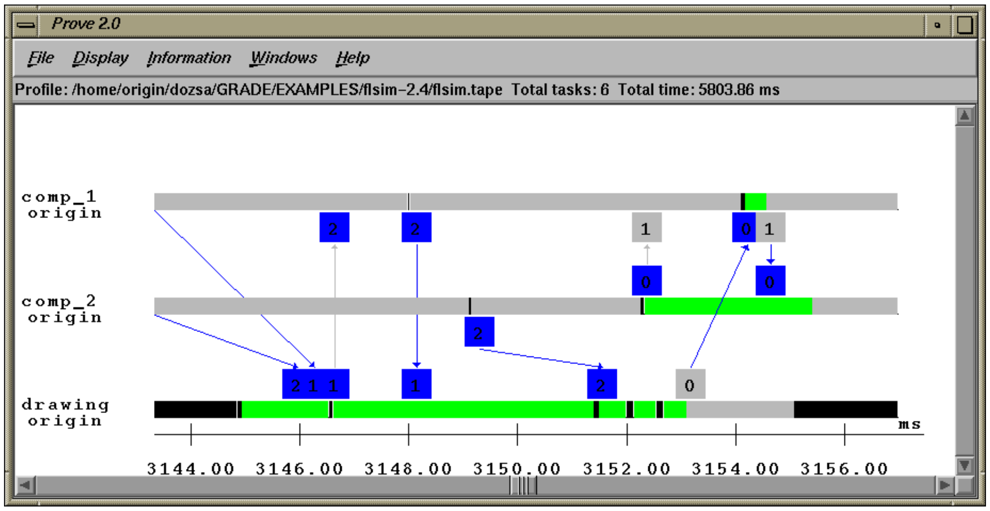
The trace analysis phase should also support some displaying features that are most relevant for the user. Such facilities are zooming and filtering. Zooming means that the user can focus on any part of the whole execution and the visualization view shows the selected part in a much more detailed way. The zooming facility of PROVE is shown in Figure 5.1 and Figure 5.2 for the same program that is shown in Figure 3.2. Total view of the complete program is given in Figure 5.1 but in such a condensed figure the details of communication and other events cannot be observed. A zoomed version of Figure 5.1 is shown in Figure 5.2 where only three processes were selected in the time interval of 3144-3156. Notice that such a zoomed figure can give details on the ports applied in the communication events as well as on the change of state of processes during and among communications. The different colours in the horizontal process bars represent different process states like idle, waiting for communication and busy.



Profile: /home/origin/dozsa/GRADE/EXAMPLES/flsim-2.4/flsim.tape Total tasks: 6 Total time: 5803.86 ms

FIG. 5.1. *The complete space-time diagram of the flight simulation program*

The role of post-mortem filtering is different from the role of the filtering during code instrumentation. Post-mortem filtering helps in selecting relevant information from the collected data similarly to the zooming feature. However, filtering is more selective than zooming and hence it can help in selecting the required processes, processors, communication events, etc. and to visualise only these selected events and units. In order to help the user in selecting post-mortem filters and to rearrange the order of processes and processors in the space-time diagram PROVE provides the



Profile: /home/origin/dozsa/GRADE/EXAMPLES/flsim-2.4/flsim.tape Total tasks: 6 Total time: 5803.86 ms

FIG. 5.2. *The zoomed space-time diagram of the flight simulation program*

dialog window shown in Figure 5.3.

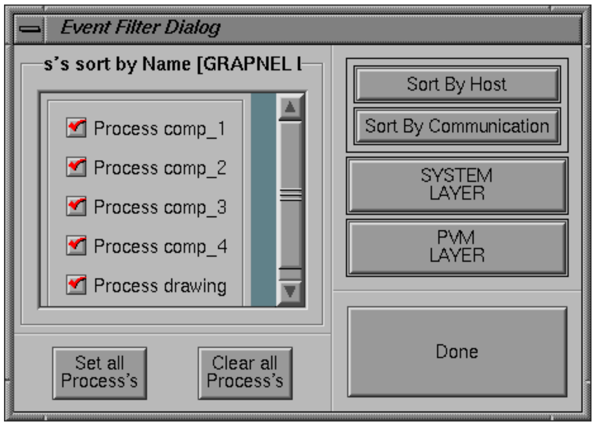


FIG. 5.3. *Event filter dialog window in* PROVE

6. Visualization. Most performance visualization tools (Paragraph [2] Pablo [7] VAMPIR [6]) provide a significant number of various display views to visualise the various aspects of program execution. The current version of PROVE gives detailed space-time diagram which describes the communication aspects of parallel processes as well as the change of their state in time. It also shows on which processor the processes were executed and when they were created on the processor. The space-time diagram of PROVE is shown in Figure 5.1 and Figure 5.2.

PROVE provides three additional windows for statistical purposes. One of them shows the processor utilization by representing process states in a common window. When all the processes that were executed on a particular processor are shown by the Process State Window, the utilisation of the selected processor is well demonstrated. The other two statistical windows are related to communication. The Process Communication window shows the amount of process communication as function of time. The Host Communication window displays the amount of communication among selected hosts in the communication network or among selected processors in a parallel computer. The time range of the three windows are jointly synchronized together with the space-time diagram. The statistical windows are shown in Figure 6.1.

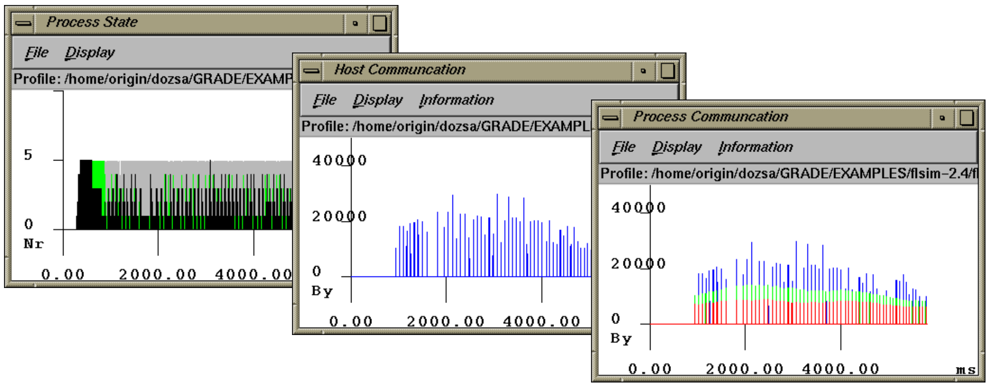


FIG. 6.1. *Statistical windows of PROVE*

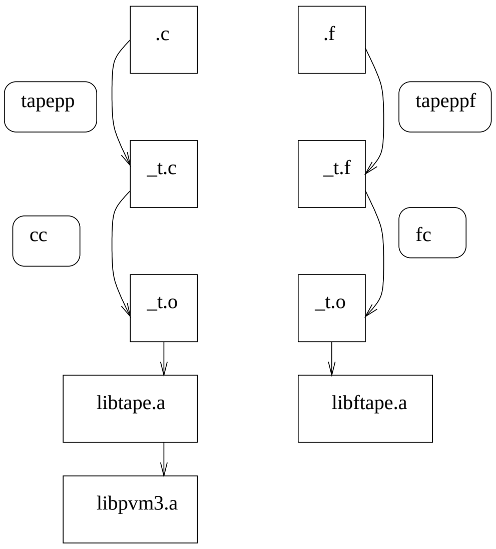


FIG. 7.1. *Tape/PVM system architecture*

7. Tape/PVM instrumentation architecture. So far we have described Tape/PVM and PROVE from the user's point of view. In the current section we give some insight into the Tape/PVM instrumentation of GRAPNEL programs which is practically hidden from the user. The only feature which is important for the user is the way how to set the Tape/PVM instrumentation option when he/she starts the GRADE system.

In Chapter 11, it is explained how to generate C source code from GRAPNEL programs and how to extend them with the necessary PVM or MPI function calls through GRAPNEL Library functions. In the current section we show how to create the necessary instrumentation for the Tape/PVM trace generation system. The main idea of the instrumentation is that every PVM call is replaced in a pre-processing phase with its instrumented version taken from the Tape/PVM library. Instrumenting a parallel application for Tape/PVM comprises three phases which will be discussed in the following subsections.

7.1. Pre-processing phase. Tape/PVM proposes a trace format along with a series of tools operating on this format. Users are also allowed to define their own trace format. In this section we assume pre-processing is done in order to generate traces in the Tape/PVM format.

The Tape/PVM software distribution contains special pre-processing tools which can automatically insert instrumentation points (*probes*) in C and Fortran application source files[2]. The pre-processing phase consists essentially of inserting a call to the Tape/PVM initialization function (*tapestart* or *tapefstart*) and in *intercepting* calls to the PVM library. For each PVM library function there is an associated intercepting function which records the trace information before passing control to the actual PVM function.

The Tape/PVM pre-processor is called *tapepp* or *tapeppf* depending on whether you want to instrument C or Fortran code. Use

$$tapepp[f] \ \ [options] \ \ source.(f \mid c)$$

to create an instrumented source code. The resulting instrumented source file is called *source.t.*$(f \mid c)$[3]. The *tapepp* tools associate a unique source file identifier to each source file they processed and keep these identifiers in a database. The generated Tape/PVM events contain pointers to the line number and file identifier which contain the statement that generated the event. Thus, analysis tools based on Tape/PVM traces can feature source code click-back.

7.2. Compiling phase. The instrumented source files ($\_t.(f \mid c)$) are compiled like the non instrumented files with few exceptions:

- The _t.c files need a special include file.
- Due to instrumentation insertion, the _t.f file may contain lines longer than the 72 characters allowed by standard Fortran (a special option has to be used in order to permit longer lines - unfortunately, there is no standard way in Fortran to do so).

7.3. Linking phase. Like the PVM library, the Tape/PVM library comprises two modules: a main library *libtape.a* and the associated Fortran interface library *libftape.a*. The dependencies between the different modules are shown in Figure 7.1. The name of the instrumented executable has to be the same as the name of the corresponding non-instrumented executable suffixed by _t. When intercepting *PVM spawn* calls, Tape/PVM automatically suffixes the task's name by _t. If this naming convention is not respected, all the *spawns* in the instrumented application will fail.

8. Tape/PVM as a stand-alone tool. The Tape/PVM monitor can be used independently from GRADE as a stand-alone tool for monitoring PVM programs and its output can be connected to stand-alone visualization tools like Paragraph. The

[2]User code pre-processing is required because Tape/PVM does not use PVM's run-time event collection mechanism.

[3]$(f \mid c)$ means that the extension is either .f or .c.

trace format output by Tape/PVM is close to the PICL format [1]. A tool (*t2p* , *t2np*) can be used to transform the traces to the PICL format so that they can be visualised with Paragraph [2]. A special feature of *t2p* is that it models the overhead due to buffer flushes by the "overhead" state. Thus, with Paragraph, the overhead due to buffer flushes is clearly outlined on the "Task Gantt Chart" so that users can study the intrusion by comparing different executions using different buffer sizes (which can be parameterised in Tape/PVM). *t2p* also takes into account the overhead due to packing (unpacking) data in (from) messages. Visualization of group operations in Tape/PVM is fully supported.

9. Conclusions. The Tape/PVM monitor proved to be easily integrated into the GRADE programming environment. Besides, it can be used as a stand-alone monitoring tool for PVM programs. The main features of Tape/PVM are as follows:

- Trace of events at user application level (PVM library calls) through function call interception.
- Pre-processor to instrument user source code (C or Fortran) automatically (instrumented source code has to be recompiled).
- User defined events (like *printf*).
- An event contains the line and file number of the instruction which generated the event (source code feed-back).
- Selective tracing using *source code module groups* and *event types.*
- Precise, causally coherent global time reference.
- On-line event compacting (gain up to 50% with respect to text storage) to limit event buffer flushes.
- Includes a C library which allows to read Tape/PVM traces easily.
- Can generate PICL traces for use with Paragraph.

The PROVE visualization tool is strongly integrated with the Tape/PVM monitor and also with other tools of the GRADE program development environment. Such a strong integration enables the unique click-back and click-forward facilities of PROVE.

REFERENCES

[1] G. A. Geist, M. T. Heath, P. B. W., and P. H. Worley, *PICL, a portable instrumented communication library*, TN 37831-8083, Oak Ridge National Laboratory, Oak Ridge, USA, 1991.

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

[3] 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.

[4] L. Lamport, *Time, clocks, and the ordering of events in a distributed system*, CACM, 21 (1978), pp. 558–565.

[5] 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.

[6] W. E. Nagel, A. Arnold, M. Weber, H. Hoppe, and K. Solchenbach, *VAMPIR: Visualization and analysis of MPI resources*, Supercomputer 63, 12 (1996), pp. 69–80.

[7] D. A. Reed, *Performance analysis of parallel systems: Approaches and open problems*, in Proceedings of JSPP'98, 1998, pp. 239–256.

[8] 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.