TAPE/PVM an efficient performance monitor for PVM applications – User guide*

Eric Maillet

LMC-IMAG / 46, av. Félix Viallet / F-38031 Grenoble CEDEX 1 (France)

e-mail: *Eric.Maillet@imag.fr*

June 27, 1995

1 Introduction

1.1 General overview

TAPE/PVM is a tool to generate event traces of PVM applications for post-mortem performance analysis, e.g. discrete event simulation and 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.

We believe that 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. In the design of TAPE/PVM we 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 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

*This manual and TAPE/PVM's distribution are available by anonymous ftp from *ftp.imag.fr* in */pub/APACHE/TAPE*. TAPE/PVM version at date of writing is V0.9pl4.

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 behavior 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 we use a non-intrusive, statistical method which estimates a precise global time reference [5]. Rather than doing post-mortem tachyon removal, we achieve *a priori* tachyon prevention 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. Our approach is similar to the one adopted in the AIMS environment [7]. We believe that, 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 we use the following techniques:

- 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 signficantly 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 synchronization) are not active while the instrumented user application is running.

In order to facilitate subsequent intrusion compensation, TAPE/PVM generates special events which contain the delay induced by buffer flushes. The complexity of the event storage routine used in TAPE/PVM only depends on the length of the stored events (compacting included). Thus, it is possible to account for the accumulation of event storage overhead knowing the length of the stored events. We have already experimented such a compensation technique on a CSP programming model (Transputer Network). The present version of TAPE/PVM does not include an intrusion compensation utility. Work on such a utility is still in progress.

The trace format output by TAPE/PVM is close to the PICL format [3]. We provide a tool (*t2p*, *t2np*) to transform the traces to the PICL format so that they can be visualized with Paragraph [4]. 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 parameterized in TAPE/PVM). *t2p* also takes into account the overhead due to packing (unpacking) data in (from) messages. Visualization of group operations in PVM is fully supported.

Finally, to allow users to quickly find the statement in their source code which 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 preprocessor (*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). Thus, a visualization tool based on TAPE/PVM traces can feature source code click-back.

1.2 Related work

Up till now, TAPE/PVM can only be used with the visualization tool Paragraph. However, another team of our departement of performance analysis at LMC-IMAG is working on a simulator coupled with a visualization tool (SCOPE [2]) using the NeXT user interface, which has proven to be very flexible due to its object oriented design. The visualization tool features 3D displays of task graphs outlining the successive states of the different tasks and communication channels. Another view displays a 3D Kiviat. At present, the simulator of SCOPE only supports the CSP communication model and can be used with traces taken on Transputer networks (a earlier version of TAPE is also available for Transputer networks [1]). However, it will be adapted soon to support PVM's communication model - it will then directly read TAPE/PVM traces.

1.3 Summary of TAPE/PVM's features

- trace of events at user application level (PVM library calls) through function call interception
- preprocessor 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

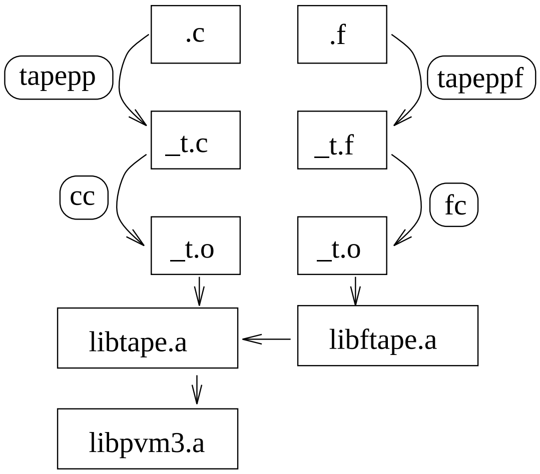

Figure 1: TAPE/PVM system architecture

- can generate PICL traces for use with Paragraph (including the new PICL format [6])

2 TAPE/PVM instrumentation architecture

Figure 1 shows the general architecture of the TAPE/PVM trace generation system. Instrumenting a parallel application comprises three phases which will be discussed in the following subsections.

2.1 Preprocessing phase

TAPE/PVM proposes a trace format along with a series of tools operating on this format. Later, we will see how users can define their own trace format. In this section we assume preprocessing is done in order to generate traces in the TAPE/PVM format.

The TAPE/PVM distribution contains special preprocessing tools which can automatically insert instrumentation points (*probes*) in your C and Fortran application source files[1]. The preprocessing phase consists essentially in inserting a call to the TAPE/PVM *initialization* function (*tape_start* 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 preprocessor is called *tapepp* or *tapeppf* depending on whether you want to instrument C or Fortran code. Use

```
tapepp[f] [options] source.(f|c)
```

to create an instrumented source code. The resulting instrumented source file is called `source_t.(f|c)`[2]. The *tapepp* tools associate a unique source file identifier to each

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

[2] `.(f|c)` means that the extension is either `.f` or `.c`.

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*.

2.2 Compiling phase

The instrumented source files (`_t.(f|c)`) are compiled like the non instrumented files with few exceptions:

- the `_t.c` files need a special include file (an extra `-I$(TAPEINCDIR)` compile option has to be added so that the include file can be found),
- due to instrumentation insertion, the `_t.f` file may contain lines longer than the 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).

2.3 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*. Figure 1 shows the dependencies between the different modules. The **name of your instrumented executable** has to be the same as the name of the corresponding non-instrumented executable suffixed by `_t`. When intercepting your *pvm_spawn* calls, TAPE/PVM automatically suffixes the task's name by `_t`. If this naming convention is not respected, all the spawns in your instrumented application will fail.

3 Instrumenting your program: a case study

This section shows how to create or change your *Makefiles* in order to instrument your applications. We use the *gexample* application that comes with the PVM distribution (there are both C and Fortran versions of *gexample*) to illustrate the procedure. All the files mentioned in this section can be found in the `./examples/pvm3` directory of the TAPE/PVM source tree. In this section we only discuss the C version (*gexample.c*) whose makefile is *Makefile* (see *gexample.f* and *Makefile.f* for the Fortran version).

The *Makefile* builds both the non-intrumented and instrumented executable tasks – *gexample* and *gexample_t* respectively. I assume your PVM environment variables *PVM_ROOT* and *PVM_ARCH* are correctly defined (refer to the PVM
-User Manual). For TAPE/PVM we introduce another environment variable called *TAPE_ROOT* which points to the directory where its libraries and include files are installed. This is usually *$PVM_ROOT/tapelib*. Lines 11–16 of the *Makefile* define the include and library directories for both PVM and TAPE/PVM. The actual libraries are defined in lines 20–21. Line 23 tells *make* to build both the non-instrumented and instrumented versions of *gexample*. Lines 31–43 compile and link the non instrumented version: you should be familiar with this. The rules to build the instrumented version are in lines 45–63. Apart from the additional libraries and includes, compiling and linking are not changed. The rule in line 62–63 tells make how to obtain the instrumented

```
#####################################################
# Sample Makefile to instrument 'gexample.c' from the
# PVM3 distribution (C version)
# ---------------------------------------------------
#                      TAPE/PVM
# E.Maillet, LMC-IMAG, University of Grenoble, France
#####################################################

# include and library directories

PVMLIBDIR = -L$(PVM_ROOT)/lib/$(PVM_ARCH)
PVMINCDIR = -I$(PVM_ROOT)/include
PVMBINDIR = $(PVM_ROOT)/bin/$(PVM_ARCH)

TAPELIBDIR = -L$(TAPE_ROOT)/$(PVM_ARCH)
TAPEINCDIR = -I$(TAPE_ROOT)/include

# libraries

PVMLIB = -lpvm3 -lgpvm3
TAPELIB = -ltape

all: gexample gexample_t

clean:
	rm -f gexample gexample_t *.o gexample_t.c

install: gexample gexample_t
	cp -f gexample gexample_t $(PVMBINDIR)

####################################################
# build non instrumented version (as usual)
####################################################

# link

gexample: gexample.o
	$(CC) $(PVMLIBDIR) $(PVMLIB) gexample.o -o gexample

# compile

gexample.o: gexample.c
	$(CC) $(PVMINCDIR) -c gexample.c

####################################################
# build instrumented version
####################################################

# link

gexample_t: gexample_t.o
	$(CC) $(TAPELIBDIR) $(PVMLIBDIR) gexample_t.o -o gexample_t \
		$(TAPELIB) $(PVMLIB)

# compile instrumented source

gexample_t.o: gexample_t.c
	$(CC) $(PVMINCDIR) $(TAPEINCDIR) -c gexample_t.c

# build instrumented source from normal source

gexample_t.c: gexample.c
	tapepp -fDB gexample.c
```

Figure 2: Sample makefile

source from the non-instrumented source. Here we have to call *tapepp*. The `-f` switch tells *tapepp* to use a file named *DB* in the current directory as its source file data base.

> **Note:** if your version of *make* supports pattern matching rules, like *GNU Make version 3.70*, generic rules can be defined to build a *%_t.c* from a *%.c*. This is very comfortable in case your program has many different source modules of which each has to be preprocessed separately. Variable substitution is also useful in case you have long *SRC* and *OBJ* variables in your existing *Makefile* and want to build the corresponding *_t* variables.

To build your tasks (non-instrumented and instrumented), type *make* (or *make -f Makefile.f* if you use Fortran). Do not forget to copy your tasks in your *pvm3/bin/...* directory – *make install* does this for you.

If your PVM application has more than one task (e.g. a master and a slave) and/or tasks are made up of several source modules, each source module has to be preprocessed with *tapepp(f)* (see also section 5).

4 Running the instrumented program

In the previous section we have seen how to build an instrumented executable task (SPMD style program) or a set of instrumented executable tasks (MIMD style program) suffixed by `_t`. Running the instrumented program generates a trace of the parallel execution. The instrumented program (master task in case of an MIMD style program) reads the *tape.cfg* configuration file in the current directory. In this file, a series of parameters can be set, which determine the behavior of TAPE/PVM: length of sample period for clock synchronization, types of events to trace...If no configuration file is found, default values are used for the different parameters (some of these default values can be changed during the preprocessing phase by means of the *tapepp* command line switches, see section 7).

Once the instrumented program has been launched, the TAPE/PVM welcome message appears[3]. Then TAPE/PVM starts the first (of two) clock synchronization phase (we use the *SBA* method discussed in detail in [5]) which lasts about 30 seconds (unless you changed the length of the sample period in *tape.cfg*). When the synchronization phase has finished, control returns to your program which starts execution as usual. Once your program has finished, TAPE/PVM starts the second phase of *SBA* and collects the events from the different tasks into a single file.

TAPE/PVM creates a direcory named `tape.<tid>`[4] in which it writes the following two files:

1. `clockstats`, which contains the statistics on clock synchronization (cf. figure 3). The *Slope* column represents the relative speed of the clocks. The *Offset* column represents the value of the clocks at the origin of reference time. Standard errors (*StdErr*) and relative errors (*Err*) of the estimation are given for both slope and offset. The precision of the estimation of the coefficients depends on the length

[3]When launched in verbose mode (`V` parameter), the instrumented application writes a series of messages to your `/tmp/pvml` file, giving information about TAPE/PVM actions.

[4]`<tid>` is the task identifier of the TAPE/PVM control task.

```
******************* TAPE/PVM - SBA Clock Statistics ****************

Reference machine         : apache2
Length of sample period   : 36 secs
Size of sample            : 15
Window size for r. median : 5

Machine    |    Slope     |   Std Err    |  Err   |Offset|Std Err|  Err   |
-----------------------------------------------------------------------
geronimo   |  0.99998652 |  0.00001025 |  0.00 | 1.42 |   0.00 |  0.03 |
apache1    |  1.00000445 |  0.00000039 |  0.00 | 1.82 |   0.00 |  0.00 |
apache2    |  1.00000000 |  0.00000000 |  0.00 | 0.00 |   0.00 |  0.00 |
apache3    |  1.00000416 |  0.00000028 |  0.00 | 0.91 |   0.00 |  0.00 |
apache4    |  1.00000257 |  0.00000025 |  0.00 | 0.77 |   0.00 |  0.00 |
```

Figure 3: The *clockstats* file (in this example, time reference is on node *apache2*)

of the sample periods. The default length (30 seconds) gives a good precision on an unloaded Ethernet network. Depending on the characteristics of your network or interconnection switch, you can increase or decrease the length of the sample periods (cf. parameters `A`, `E`);

2. `events`, which contains the execution trace of your program (readable text file). For information about how to read the trace from your own programs, take a look at the documentation provided with the *tapereader* library in *./lib/tapereader*.

If you use *pvm_kill* to kill a task of your application, all the events generated on that task are lost. Thus, if your application's master task kills all its slaves at the end of the computation, all the slaves' traces are lost. This will be fixed in a future version of TAPE/PVM.

The TAPE/PVM distribution includes a tool which transforms your `events` file into the PICL trace format so that they can be viewed with *Paragraph*. Use

```
t2np events > picl.trf
```

to generate a PICL trace in the `picl.trf` file[5].

However, there is more information in a TAPE/PVM trace than is actually used by Paragraph: *source code click back* for example, is not supported by Paragraph. We intend to use the visualization tool SCOPE under NEXTSTEP, developed at LMC-IMAG, which will allow to pop up the related source code when clicking on an object in a view.

For those who intend to develop their own trace analysis tools, TAPE/PVM includes a library (*tapereader*) to easily read the traces, allowing you to extract the event fields by their names (`t2np` is based on that library)[6].

[5] `t2np` generates a trace in the *new PICL format* for use with the new version of Paragraph. Use `t2p` to generate a trace in the old PICL format.

[6] The *tapereader* library and the related documentation can be found in *./lib/tapereader*.

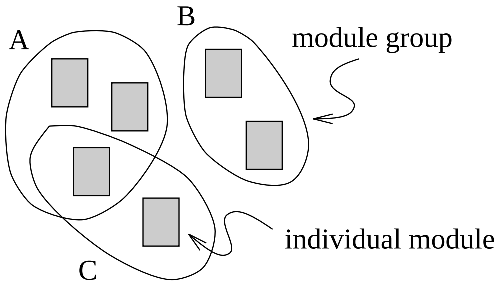

Figure 4: Example of module groups

5 How to select events

5.1 Definitions

TAPE/PVM allows you to do selective tracing. For this, you need a mechanism to identify which of the events are traced. The following two notions are used to do so:

1. *Source code module groups*
   If your application is large, your source code is probably subdivided into a set of *modules* (source files). In turn, these modules can be classified into different *module groups*: for instance, you could have one module group dedicated to communication code (A), another one for computations (B) and a last one for load balancing (C). This example is depicted in figure 4. You can define up to 26 module groups named A,B,...Z (capital letters).

2. *Event types*
   TAPE/PVM subdivides the PVM functions into several classes:

   - class **c**: control functions,
   - class **m**: message communication functions,
   - class **p**: message packing and unpacking functions,
   - class **i**: information functions,
   - class **s**: system functions,
   - class **g**: group functions,
   - class **u**: unclassified functions.

   An event's *type* is the name of the function's class which generated the event. The event generated by *pvm_send*, for instance, is in class **m** – its type is **m**. See section 7 for more details about event types.

5.2 Usage of module groups and event types

With *tapepp* and *tapeppf*, the assignment of a module (source file) to one or several module groups is defined.

Examples:

```
tapepp -mB file.c        { assign file.c to module group B }
tapepp -mAC file.c       { assign file.c to module groups A and C }
```

Note that if no `-m` switch is given, the pre-processed file belongs to *all* modules (A,B,...Z).

Once all your modules have been pre-processed, and your instrumented application has been built, you can specify, using the `M` parameter in *tape.cfg*, which events to trace. For example, giving `M` the value of *Cgp* means to only trace events generated by modules in group C which are *not* of type **g** and *not* of type **p**. See section 7 for more details on how to specify the `M` parameter.

> **Important note:** when selective tracing is used, you will get a sub-trace of your program. While such a sub-trace allows you to focus on a particular functional component of your application, there may be not enough information in such a sub-trace for a graphical analysis with Paragraph. See section 7.

6 Defining your own trace format

> **Note:** This section refers to TAPE/PVM functions which are not fully implemented yet, and which are still in an experimental stage. We include it in this manual to give an idea about what will be available in future releases of TAPE/PVM.

The preprocessing phase discussed in section 2 is automated by the TAPE/PVM preprocessing tools *tapepp* and *tapeppf*. Instrumenting your sources with these tools will give you a trace in the predefined format. Defining your own trace format implies doing the preprocessing phase yourself. This section shows how to do so.

As we have already stated in section 2, the *tapepp* tools change two things in your source code: first, they insert a call to the *tape_start* function to initialize TAPE/PVM's internal data structures; second, they intercept the call to PVM library functions to generate the appropriate trace data (*event*) before calling the actual PVM function. To generate an event, all intercepting functions call the *tape_event* function, which works much like the C *printf* function:

```
tape_event(format_string, arg1, arg2, ...)
```

This allows you to create an arbitrary event with a series of alphanumeric (`%s`), integer (`%d`) and floating point (`%f`) fields. The event is automatically compacted[7] and stored into TAPE/PVM's internal event buffer. If you want to generate events for performance analysis, your events probably contain a *physical time stamp*. On many systems, there is no global time reference and your time stamps are defined with respect to the local clock of the node on which the event is generated. In such a case, you can use TAPE/PVM's global time estimation mechanism to transform your local time stamps into global time stamps. Thus, the time stamps of events from different nodes

[7] String fields are not compacted

become comparable. TAPE/PVM's global time reference is only computed during event collection at the end of your application's execution (see [5] for more details). Therefore, you have to tell TAPE/PVM in which of your events' fields it can find your time stamp so that it can globalize it for you. To do so, you have to respect the following two points:

1. your time stamp has to consist of *two* fields, one for seconds, one for microseconds (these two fields are directly returned by the *gettimeofday* and *tape_clock* functions[8]);

2. the second and microsecond field numbers have to be set with the *tape_set_s_field* and *tape_set_us_field* functions; time stamp field numbers have to be identical for all events.

   Note: if your parallel machine already has a precise global time reference you do not need TAPE/PVM's global time mechanism and you can switch it off using the `E` parameter (see *tapecfg* in section 7).

Finally, at the end of your tasks, just before the *pvm_exit* call, you have to call the *tape_exit* function.

All the points mentioned above are summarized on figure **??** which shows a user defined preprocessing phase. Note the additional `#include` with the TAPE/PVM library function prototypes. The time stamp fields have to be defined before the call to *tape_start*. Arguments to *tape_start* specify default operating parameters which can be overridden by the parameter definitions in *tape.cfg* (see *tapecfg* in section 7). Event generation is illustrated by recording some information relative to a send operation. The *tv.sec* and *tv.u_sec* fields have been obtained by a previous call to the *gettimeofday* or *tape_clock* function (*tv* is a *timeval* structure). Notice that the position of the time stamp fields in the events have to be compatible with the field settings done before.

[8] On most systems, *tape_clock* directly uses *gettimeofday*. On IBM SP1 or SP2 systems, *tape_clock* uses the nano-second resolution clock of the high performance switch.

7 Reference

NAME

tapepp, tapeppf	instrument C or Fortran PVM3 source code for use with Tape/Pvm

SYNOPSIS

```
tapepp[f] [options] file
```

The *tapepp* (*tapeppf*) program takes a PVM3 C (Fortran) source code file as input, and produces an instrumented source file to be compiled and linked with the TAPE/PVM libraries. *tapepp(f)* substitutes the calls to PVM3 library functions by calls to the corresponding interception functions of the TAPE/PVM library. Intercepting functions do the same job as the corresponding PVM3 functions, but execute some additional code to record trace data. *tapepp(f)* also inserts a call to the *tape_start* (*tapefstart*) function which initializes TAPE/PVM and dictates the default behavior of the resulting executable instrumented tasks (once the instrumented source code has been compiled and linked). The default behavior of the instrumented executable can be set using the *tapepp(f)* options.

OPTIONS

-f*file*

file is used as source code data base. For each file processed, tapepp(f) finds an identifier which is stored, together with the absolute file name, in the data base. TAPE/PVM events contain the source code identifier. This allows you to find the source file where an event came from (the line numbers are also part of all the events' fields)

-m*group*

Up to 26 module groups can be used. These groups are named A,B,..Z. The **-m** option is used to specify the module group to which belongs the source module processed by *tapepp(f)*. A module can belong to more than one group. By default, a module belongs to all groups A-Z. Examples of usage are:

Specify that file.c (file.f) belongs to module group A

```
tapepp  -mA file.c
tapeppf -mA file.f
```

Specify that file.c (file.f) belongs to both module groups A and D

```
tapepp  -mAD file.c
tapeppf -mAD file.f
```

-o*myfile*

Instrumented source code will be written to *myfile*. Default is to use *file_t.c* (*file_t.f*) for input file *file.c* (*file.f*).

-T*size*

Use *size* (in bytes) as the default buffer size. size must be the same for all source modules of the tasks of the instrumented PVM application (default is 10000 bytes). The **T** default parameter can be overridden in the *tape.cfg* file

-E*nbxchg*
-A*delay*

Set default parameters for clock synchronization. **-E** sets the default number of message exchanges, **-A** specifies the default delay (in secs) between two successive exchanges. The **A** and **E** default parameters can be overridden in the *tape.cfg* file. Defaults are *nbxchg=15*, *delay=2*. You should experiment different values for these parameters on your system (check the *clockstats* file for the obtained precision)

SEE ALSO

tapecfg

BUGS

The *tapepp* parsers may not recognize certain calls to the PVM library (bugs in regular expression stuff). Some of these bugs have been fixed. Missing to instrument PVM calls can cause the instrumented application to stall, or to generate incoherent traces (in case a message reception is recorded without the corresponding send).

The *tapeppf* parser (written in perl) tries to find automatically where to insert the call to the *tapefstart* function. It locates your PROGRAM statement, skipping any following declarations and comments and then inserts the call to *tapefstart* (as the first statement of the instrumented source code). If you get compilation errors, when compiling instrumented Fortran sources, *tapefstart* was probably not properly inserted.

tapeppf expects to find perl in */usr/local/bin*. If your perl is elsewhere, edit the first line of *tapeppf* (the one starting with "# !").

Please report bugs to **Eric.Maillet@imag.fr**

NAME

tapecfg – runtime parameters for instrumented executables

The *tape.cfg* file allows you to override the default parameters specified with *tapepp(f)* and to set some additional parameters. The *tape.cfg* file is searched for in the directory where you launch the instrumented application. It can contain comments (i.e. empty lines or lines where the first character is a '#'). Non comment lines are parameter definitions and have the following format:

```
parameter_name parameter_value
```

Parameter names have just one letter. Possible values depend on the parameter. The following list describes all the parameters.

T *buffer_size*

Specifies the size in bytes of the trace buffers. Default is defined by the *tapepp(f)* preprocessor's command line switches.

E *nb_exchanges*

Specifies the number of message exchanges for clock synchronization. Setting *nb_exchanges* to 0, suppresses clock synchronization : use this on machines which have a precise global time reference. Note that periodic system clock synchronizations (enabled on some distributed OS's) are not precise enough to date events. In case clock synchronization is active, TAPE/PVM writes the results of statistical analysis to the clockstats file.

Default set by *tapepp(f)*.

A *delay*

Specifies the delays in seconds between 2 successive clock synchronization messages.

Default set by *tapepp(f)*.

W *win_size*

The samples collected by the clock synchronization procedure are smoothed using the running (moving) median method. The *win_size* parameter is used to set the width of the running window. Default is *win_size* =5.

You should experience different values of the **E**,**A** and **W** parameters in order to get the best precision on your system (see *clockstats* file). The defaults should be OK for communications over a non-loaded Ethernet.

M *mask*

Specifies which type of events in which source modules are traced. This allows you to do very selective tracing. The form of the mask is specified as follows (we use a regexp to specify the form):

```
[A-Z]*(c|m|p|s|g|i)+
```

The sequence of capital letters A-Z are used to specify in which source module groups to trace events. If no source module letter is given, tracing is active in all modules. See the tapepp manpage for more information on souce module groups.

The lower case letters c,m,p,s,g and i are used to *disable* tracing for one or more particular event classes. If no classes are specified, tracing is active in all classes. Some events, important for the integrity of the traces, cannot be disabled: *pvm_mytid*, *pvm_spawn* and *pvm_exit*.

A given event is traced if it occurs in one of the specified module groups and if its class is not disabled.

TAPE/PVM subdivides the PVM library functions into 7 classes defined as follows:

class c: control type events, generated by pvm_mytid, spawn, kill, exit, notify, sendsig

class m: message communication events, generated by pvm_recv, trecv, precv, send, psend, mcast, nrecv, serror, recvf, probe

class p: message packing related events, generated by all pvm_pk*, pvm_upk* functions

class s: system events, generated by pvm_addhosts, advise, delhosts, tidtohost

class g: group events, generated by pvm_barrier, getinst, bcast, gettid, gzise, joingroup, lvgroup, reduce, scatter, gather

class i: information events, generated by pvm_parent, config, bufinfo, getopt, tasks, pstat, perror

class u: events which could not be classified in any of the above classes, generated by pvm_getfds, insert, delete, halt, lookup, mstat, setdebug, start_pvmd, tickle, version

Default is to trace all events but the packing events *p*. Pk* and Upk* functions are used very frequently in most of the PVM applications. Tracing these generates great volumes of events causing multiple event buffer flushes: this perturbs the instrumented applications. Size of the resulting trace files are huge (tens of megabytes).

Examples of trace masks:

p : trace all events but the packing events in all module groups (this is the default mask)

Ap : trace all events but the packing events in module group A

si : trace all events but the system and info events in all module groups

AZcmsgi : trace only the packing events in module groups A and Z

NOTE: in order the generated traces be representative, a minimum amount of event information is needed. To use *t2p* (*t2np*) on the generated traces, you have to activate tracing in all module groups on at least the following classes: *c* ,*m* (*g* if you use groups). You can also trace the *p* events: then Paragraph models the delay of packing and unpacking by the overhead state. Be conscious about the amount of pk's and upk's in your application !

P *nbphases*

Specifies the number of simultaneously open phases per task. Default is defined by the *tapepp(f)* preprocessor

V *verbose_mode*

Specifies if TAPE/PVM is verbose (*verbode_mode=1*) or not (*verbose_mode=0*). In case verbose is on, TAPE/PVM writes a series of information to your */tmp/pvml...* logfile. Visualizing the logfile with *tail -f* provides real-time information about TAPE/PVM's progress (clock synchro, event collection...). Default is non-verbose mode.

D *dyna_conf*

Specifies if your pvm config is dynamic (*dyna_conf=1*) or not (*dyna_conf=0*). If *dyna_conf* is set, TAPE/PVM checks at the end of your application's execution which hosts have been added, and repeats clock synchronization for these hosts. Note that clock synchronization is much more accurate if your pvm config is stable.

FILES

`./tape.cfg`	configuration file with operating parameter settings
`/tmp/pvml...`	your log file with TAPE/PVM info (see V parameter)
`./tape...`	directory with the trace file (events) and clock statistics (clockstats)

NAME

tapefunc – functional interface to the TAPE/PVM kernel

Some of these functions are automatically inserted into your source files by *tapepp* and *tapeppf*. Others can be inserted manually to define *phases* or to change the *event mask* dynamically. These functions are prototyped in *tape.h*.

User defined phases

```
phase_id = tape_open_phase(my_phase)
call tapefopenphase(my_phase,phase_id)

ret = tape_close_phase(phase_id)
call tapefclosephase(phase_id,ret)
```

Phases can be used to delimit particular regions of your applications. Events in such a phase are generally of particular interest to the user. The *tape_open_phase* function opens a new phase of name *my_phase* and returns a phase descriptor (int) in *phase_id*. This function generates a special event in the trace, which identifies the instant at which the phase was opened. *tape_close_phase* closes a given phase. Both functions return -1 in case of error. *t2np* takes account of these phase events and generates user event marks in the PICL trace. The visualization can then be stopped at the start and at the end of phases using "Pause on tracemark/msg" in Paragraph's option menu.

Changing the event selection mask

```
ret=tape_set_mask(event_mask)
call tapefsetmask(event_mask,ret)
```

You can change the activity of TAPE/PVM's probes dynamically using the *tape_set_mask* function. *event_mask* is a string containing module group letters and event types. See the description of the **M** parameter in *tapecfg* for the details about how to specify *event_mask*.

The function returns -1 if you provide an erroneous *event_mask*. If *event_mask* is correct, and TAPE/PVM is set for the new mask, the function returns 1.

Note: the *tape_open_phase*, *tape_close_phase* and *tape_set_mask* functions actually have 3 additional arguments (before those documented here). These arguments are the file identifier, the line number and the binary event mask, and are automatically provided when your source code is pre-processed by *tapepp* or *tapeppf*.

Initializing TAPE/PVM

```
void tape_start(argc,argv,size,nbphase,delay,nbxchg)
call tapefstart(size,nbphase,delay,nbxchg)
```

Initializes TAPE/PVM with its default operating parameters (set with *tapepp*, *tapeppf*), which are the buffer size (*size*), the maximum number of simultaneously open phases (*nbphase*), the delay in seconds between two successive clock synchronization message exchanges (*delay*), and the number of message exchanges (*nbxchg*). *argc* and *argv* are the command line arguments which are not used currently (can be set NULL).

If the caller is the master task (i.e. it has no parent), the default parameters (given as arguments to *tape_start*) are overridden by the parameters defined in *tape.cfg* (if any), the tape control task is created, and the caller is blocked until the control task has finished the first clock synchronization phase.

If the caller is a slave task (i.e. it is spawned by another task), it receives the operating parameters from its parent (parameters are sent by the parent during intercept of the *pvm_spawn* function). None of the arguments of *tape_start* are used.

Finally, master or slave, initialize local data structures.

tape_start is automatically inserted by *tapepp*, *tapeppf* and has to be the first statement in your task's main program.

Intercepting functions

```
Tpvm_*(file,line,binmask,pvm_args)
tpvmf*(file,line,binmask,pvm_args)
```

Intercepting functions are automatically substituded to the corresponding *pvm_** (*pvmf**) functions by *tapepp* (*tapeppf*). *pvm_args* are the usual arguments to the PVM function, while *file*, *line* and *binmask* are additional arguments provided by the pre-processors.

NAME

t2p, t2np – generate PICL traces from TAPE/PVM traces

SYNOPSIS

```
t2[n]p events > picl
```

t2p and *t2np* are used to transform TAPE/PVM traces into the PICL format for visualization with Paragraph [4]. *t2p* generates PICL traces for the old version of Paragraph, while *t2np* generates traces for the new version. It is recommended to use *t2np* with a recent version of Paragraph, because it generates much more accurate traces than *t2p*. *t2np* fully supports PVM group operations including *scatter, reduce* and *gather* operations. The generated PICL traces are directed to standard output. *t2np* is a wrapper for *tape2npicl* which does the actual conversion from the TAPE/PVM format to the PICL format.

Paragraph uses 3 states to model activity of nodes: *busy, idle* and *overhead*. Refer to [4] for a description of the semantics of these states. *t2np* generates idle states when a PVM task blocks on a request. A call to *pvm_recv*, for example, which blocks in case the specified message has not arrived yet, generates an idle state. Other causes for idle states are barriers, receives with time-out, scatter, reduce and gather operations. *t2np* generates idle states when a PVM task spends some non-zero time in a PVM system call, but without blocking. A call to *pvm_recv* for example, does not block the calling task, but takes some delay to execute. Similarly, a non-blocking call to *pvm_nrecv* causes some overhead to check if the specified message has arrived, and if so, to copy it into the active receive buffer. Other causes of idle states are mcasts and bcasts, scatter, reduce and gather operations, message packing and TAPE/PVM event buffer flushes. States which are not idle and not overhead are busy.

To identify its nodes (tasks), Paragraph uses numbers from 0 to $n-1$, where n is the number of nodes. By default, *t2np* assigns PICL nodes to PVM tids chronologically. A master task, for example, which spawns a number of slaves is assigned node number 0. The slave which is spawned first is assigned node number 1 and so forth. *t2np* generates a file named *nodetable.out*, with one line per task, which shows the node number (field 1), the corresponding decimal tid (field 2), hexadecimal tid (field 3) and some information about the architecture on which the task was running (field 4). You can change the way in which *t2np* assigns PICL node numbers to PVM task tids by creating a file named *nodetable* in the directory where *t2np* is run. *nodetable* should be a copy of *nodetable.out* in which you can swap some of the node numbers in the first column to assign them to different tids.

The following is an example of a *nodetable.out* file, generated by *t2np*, where nodes have been assigned chronologically (default):

```
0     786452    c0014   nsw2/RS6K/1000
1    1048596   100014   nsw3/RS6K/1000
2    1310740   140014   nsw4/RS6K/1000
```

Node 0 is a master task which spawns two slaves, nodes 1 and 2. Due to the assignment of nodes to tids, Paragraph will display the master task at the bottom of the Spacetime or Gantt Diagrams. If you prefer the master task to be visualized in the center, between the two slaves, you can create the following *nodetable* file (only the first column is changed):

```
1     786452    c0014   nsw2/RS6K/1000
0    1048596   100014   nsw3/RS6K/1000
2    1310740   140014   nsw4/RS6K/1000
```

FILES

events, nodetable.out, nodetable

Acknowledge: I owe much to L. Azema and M. Collet, two students at Ensimag, Grenoble, France, who did much of the implementation of TAPE/PVM and also made interesting suggestions as to TAPE/PVM's overall design. I also express my thanks to all the members of the *Alpes* project, in particular to J.P. Kitajima, whose suggestions helped in improving this document, to Y. Arrouye, who wrote the Transputer version of TAPE, and to Pr. B. Plateau who supervises the performance evaluation team at LMC-IMAG.

References

[1] Y. Arrouye. *The* TAPE *Reference Manual.* LMC-IMAG, 46, av. Félix Viallet, 38031 Grenoble CEDEX 1, France, September 1992.

[2] Y. Arrouye. *Performance Evaluation of Parallel Systems: the Scope Extensible Interactive Environment.* IMAG–Équipe APACHE, 46, av. Félix Viallet, 38031 Grenoble CEDEX 1, France, December 1994. Rapport APACHE 15.

[3] G.A. Geist, M.T. Heath, B.W. Peyton, and P.H. Worley. *PICL A portable instrumented communications library*, 1992. Available on netlib.

[4] M.T. Heath and J.E. Finger. ParaGraph: A Tool for Visualizing Performance of Parallel Programs. Technical report, ORNL, June 1994.

[5] E. Maillet and C. Tron. On Effciently Implementing Global Time for Performance Evaluation on Multiprocessor Systems. *To appear in Journal of Parallel and Distributed Computing*, 1995.

[6] P.H. Worley. A new PICL trace file format. Technical report, ORNL, September 1992. Available on netlib.

[7] J.C. Yan. Performance tuning with AIMS – an automated instrumentation and monitoring system for multicomputers. In *Proceedings of the 27th Annual Hawaii International Conference on System Sciences*, 1994.