A Test Suite for PVM

Henri Casanova (casanova@cs.utk.edu)
Jack Dongarra (dongarra@cs.utk.edu)
Philip J. Mucci (mucci@cs.utk.edu)

March, 1995

Abstract

Although PVM is well established in the field of distributed computing, the need has been shown for a standard set of tests to give its users further confidence in the correctness of their installation. This report introduces **pvm_test** and its X interface **pvm_test_gui**. **pvm_test** was designed to exercise some of PVM's more important functions and to provide some primitive measures of it's performance.

1 Notice

PVM 3.3 was funded in part by the U.S. Department of Energy, the National Science Foundation and the State of Tennessee.

2 Installation

- Verify that PVM version 3.3 or higher including the group server is installed correctly on every machine in the host pool. Also, make sure that the environment variable `PVM_ROOT` is set in the user's login scripts.
- Install Tcl/Tk. *Version 3.x of Tk is expected. Under Tk 4.0, the GUI will not function properly without modifications!*
- Build the pvm_test distribution. Please see `Makefile.notes` for the arguments to given to make.

3 The tester

The test suite consists of two parts, the test engine itself and a GUI.

3.1 The tester engine

The tester engine itself is the `pvm_test` executable, and takes the name of a file as its first argument. This file specifies the configuration for the test run. See the file `sample.input` in the distribution for a sample file. To run the test engine, after creating a configuration file, one must :

1. Run pvm on the master host
2. Run the `pvm_test` executable on this host

We will here describe briefly how to build a configuration file for the test engine.

3.1.1 General features in the configuration file

From now, we will speak of a *line* as a line in the configuration file. First, we must say that each line beginning with a # is a comment line. We strongly suggest the use of comments when building a configuration file, because of its somewhat cryptic format.

The first thing to do is to tell the test engine which are the hosts used in the configuration. This is done by lines like this one :

```
config :rudolph
```

which means that the host `rudolph` will be in the configuration. It is not required to put the *master* host in the configuration file.

Once given a list of hosts, we can specify where the output of the test engine is to be redirected. We can choose to have it sent to the stdout :

```
outfile :stdout
```

or to a specific file :

```
outfile :whatever
```

Then, we can specify if we want the test engine to try various combinations of hosts or not with a line :

```
all_combinations :1
```

or

```
all_combinations :0
```

See 5 for more explanation about this feature.

3.1.2 Test selection

Of course, the PVM test suite allows the user to specify which tests he wants to run on his configuration. The execution of all the tests, especially on an heterogeneous network and with the `all_combination` option set, may take a pretty long time.

One can decide for each test whether it is to be performed or not. In the configuration file, a line :

```
[]test23
```

means that the test #23 will not be performed, and a line

```
[x]test23
```

means that this test will be performed. If there is no line concerning a particular test, the takes won't be executed, by default.

Besides, some options allow the user to select a whole class of tests. If the option for a class of test is set to 0, then the test engine will examine the individual selection for all tests of this class (as above). If this option is set to 1, then all the tests of the class will be executed, whatever their individual settings are.

The different classes of tests are the following :

- `all_tests` : All tests of the test suite
- `all_messaging` : tests #48 to #107
- `all_routine` : tests #1 to #44 and test #47
- `all_perf` : tests #45 and #46
- `all_head_nodata` : tests #48 to #53
- `all_head_data` : tests #54 to #61

- `all_triangle_nodata` : tests #62 to #69
- `all_triangle_data` : tests #70 to #77
- `all_funnel_nodata` : tests #78 to #91
- `all_funnel_data` : tests #92 to #107

Then for instance, if there are two lines :

```
all_triangle_data :1
[]test72
```

then the test #72 will be executed, and so will be the tests #70 to #77. And if the lines are :

```
all_triangle_data :0
[x]test72
```

then, the test #72 will also be executed and perhaps some of the tests #70 to #77.

Notice that if there is no line concerning some class of tests, the the setting for this class is taken to be 0.

3.1.3 More options

There are some other options directly related to some particular tests. These options will be described in section 4. A GUI has been designed which automates the creation of a configuration file.

3.2 The interface

You must be running X and have the *Tcl/Tk* toolkit installed to use `pvm_test_gui`. *Tcl/Tk* are available from `ftp.cs.berkeley.edu` in the directory `/pub/tcl`.

The main window consists of four panels and a menu bar. The leftmost panel contains ten check-buttons and two regular buttons. The check-buttons indicate which common subsets of the one-hundred and seven ests, the user wishes to perform. The button on the bottom of the panel brings up a listbox in which can select a single test from the complete list. The button on top is provided to start the selected tests.

In the top right panel are sliders for setting the parameters for the three types of messaging tests. (funnelling, head-to-head and triangle). These sliders can be "dragged" via the LMB. For more precise control, the values can be increased or decreased by one unit using the right and middle mouse buttons. The slider called 'Number of processes', concerns only the funnelling tests.

The next panel comtains one check-button, "All Combination". When checked, the test engine will try "all" the architecture combinations for all the selected tests (see 5 for more details).

The next panel down also refers to these messaging tests. These check buttons allow the user to select timings to be displayed upon completion of each test. "Total time" will display the total amount of time that the master process took to send all of its messages and "Average Time" will display the average time per message.

The last panel on the right side contains more sliders for setting the parameters for the bandwidth test. A message which is "Start" bytes long is sent and then increased by "Increment" bytes and sent again. This continues until the length "End" is reached. Slider control is identical to that in the upper right panel (LMB for rapid sliding, MMB and RMB for precise sliding).

On the bottom of the window there is a text widget containing the current PVM host file. Due to constraints imposed by the test engine, any options *must be specified on the same line as the host to which it is referring.* If no filename is given on the command line, **pvm_test_gui** will open up a temporary host file and fill it with the name of the host on which it is executing.

At the top of the window, exists a menu-bar and a small help box. The help box provides a small description about what the user is pointing to with the pointing device. The Menu options consist of standard file operations that can be performed with reference to the host file in the text box described above. The second menu allows the user to generate and save a configuration file for use when the GUI is inaccessible.

When the user has finished selecting tests and setting options, the "Run selection" button in the upper right hand corner should be clicked upon. The hostfile widget at the bottom will then be replaced by a status bar and a button. The status bar displays the most recent write of **pvm_test** to standard output. The "ABORT" button allows the user to terminate **pvm_test** prematurely. When the tests complete or are aborted, these widgets change into two buttons. These buttons give the user the option of displaying and saving the results or to discard them and return to the test selection phase. The user should note that any results *are not recoverable* if "done" is chosen. If the user chooses to view or save the results, a new window will open. This window contains three buttons and two text boxes. The buttons allow the user to dismiss the window entirely or save the results to a file. The rightmost text box contains a test number and a value corresponding to the success of failure of that test. If the user clicks on an entry, the leftmost box will display the results of the corresponding test. This includes a description of the test, optional timings and possible reasons for failure. There is also an entry called "Report Header" which will display the miscelleanous infromation provided by the test engine at the beginning of its execution (and possible failures).

4 Description of the tests

There are currently 107 tests contained in the `pvm_test` program. Tests numbers 1 though 47 are intended to exercise the PVM library and to test the stability of the daemon. In the short description of each test, the word *incorrect* means that this PVM functionnality is tested with incorrect parameters, or in incorrect situations, so that its behaviour in failure cases can be checked. Most of the time, there is a single test for each PVM function testing both the *correct* and the *incorrect* case. See 4.1 for more details.

Tests 45 and 46 perform latency and bandwidth measurements.

Tests 48 through 107 perform a benchmark of PVM's message passing functions using various configurations of hosts and options to PVM. See 4.3 for more details.

Test number	Description
test1	pvm_mstat() correct
test2	pvm_mstat() incorrect
test3	pvm_addhosts() correct
test4	pvm_addhosts() incorrect
test5	pvm_delhosts() correct
test6	pvm_delhosts() incorrect
test7	Race pvm_addhosts()-pvm_config()
test8	Race pvm_delhosts()-pvm_config()
test9	pvm_hostsync()
test10	pvm_pstat() correct
test11	pvm_pstat() incorrect
test12	pvm_spawn() sequential
test13	pvm_spawn() simultaneous
test14	pvm_kill() correct
test15	pvm_kill() incorrect
test16	pvm_tidtohost()
test17	pvm_parent()
test18	pvm_sendsig()
test19	pvm_tasks()
test20	pvm_exit()
test21	Are several group server started ?
test22	pvm_joingroup() incorrect
test23	pvm_lvgroup() incorrect
test24	pvm_joingroup(),pvm_lvgroup(),pvm_gsize()
test25	Group completion
test26	pvm_getinst(),pvm_gettid()
test27	pvm_initsend(),pvm_getsbuf(),pvm_freebuf()
test28	pvm_pk*xxx*(),pvm_pk*xxx*()
test29	pvm_getrbuf()
test30	pvm_bufinfo()
test31	pvm_mytid()
test32	pvm_mkbuf()
test33	pvm_send()-pvm_recv() and pvm_psend()-pvm_precv() coherency
test34	pvm_setsbuf()
test35	pvm_setrbuf()
test36	pvm_set*x*buf() incorrect
test37	pvm_bcast()
test38	pvm_barrier()
test39	pvm_reduce()
test40	pvm_gather()
test41	pvm_scatter()
test42	pvm_mcast()
test43	pvm_trecv()
test44	pvm_nrecv()
test45	Latency-Bandwidth pvm_send()-pvm_recv()
test46	Latency-Bandwidth pvm_psend()-pvm_precv()
test47	pvm_notify()

TEST	PATTERN	CODING	ROUTING	ROUTINES	CONTENT
test48 :	head-head	PvmDataDefault	daemon routing	send-recv	empty
test49 :	head-head	PvmDataRaw	daemon routing	send-recv	empty
test50 :	head-head	PvmDataInPlace	daemon routing	send-recv	empty
test51 :	head-head	PvmDataDefault	direct routing	send-recv	empty
test52 :	head-head	PvmDataRaw	direct routing	send-recv	empty
test53 :	head-head	PvmDataInPlace	direct routing	send-recv	empty
test54 :	head-head	-	daemon routing	psend-precv	empty
test55 :	head-head	-	direct routing	psend-precv	empty
test56 :	head-head	PvmDataDefault	daemon routing	send-recv	data
test57 :	head-head	PvmDataRaw	daemon routing	send-recv	data
test58 :	head-head	PvmDataInPlace	daemon routing	send-recv	data
test59 :	head-head	PvmDataDefault	direct routing	send-recv	data
test60 :	head-head	PvmDataRaw	direct routing	send-recv	data
test61 :	head-head	PvmDataInPlace	direct routing	send-recv	data
test62 :	head-head	-	daemon routing	psend-precv	data
test63 :	head-head	-	direct routing	psend-precv	data
test64 :	triangle	PvmDataDefault	daemon routing	send-recv	empty
test65 :	triangle	PvmDataRaw	daemon routing	send-recv	empty
test66 :	triangle	PvmDataInPlace	daemon routing	send-recv	empty
test67 :	triangle	PvmDataDefault	direct routing	send-recv	empty
test68 :	triangle	PvmDataRaw	direct routing	send-recv	empty
test69 :	triangle	PvmDataInPlace	direct routing	send-recv	empty
test70 :	triangle	-	daemon routing	psend-precv	empty
test71 :	triangle	-	direct routing	psend-precv	empty
test72 :	triangle	PvmDataDefault	daemon routing	send-recv	data
test73 :	triangle	PvmDataRaw	daemon routing	send-recv	data
test74 :	triangle	PvmDataInPlace	daemon routing	send-recv	data
test75 :	triangle	PvmDataDefault	direct routing	send-recv	data
test76 :	triangle	PvmDataRaw	direct routing	send-recv	data
test77 :	triangle	PvmDataInPlace	direct routing	send-recv	data
test78 :	triangle	-	daemon routing	psend-precv	data
test79 :	triangle	-	direct routing	psend-precv	data

TEST	PATTERN	CODING	ROUTING	ROUTINES	CONTENT
test80 :	funneling	PvmDataDefault	daemon routing	send-recv	empty
test81 :	funneling	PvmDataRaw	daemon routing	send-recv	empty
test82 :	funneling	PvmDataInPlace	daemon routing	send-recv	empty
test83 :	funneling	PvmDataDefault	direct routing	send-recv	empty
test84 :	funneling	PvmDataRaw	direct routing	send-recv	empty
test85 :	funneling	PvmDataInPlace	direct routing	send-recv	empty
test86 :	funneling	PvmDataDefault	daemon routing	mcast-recv	empty
test87 :	funneling	PvmDataRaw	daemon routing	mcast-recv	empty
test88 :	funneling	PvmDataInPlace	daemon routing	mcast-recv	empty
test89 :	funneling	PvmDataDefault	direct routing	mcast-recv	empty
test90 :	funneling	PvmDataRaw	direct routing	mcast-recv	empty
test91 :	funneling	PvmDataInPlace	direct routing	mcast-recv	empty
test92 :	funneling	-	daemon routing	psend-precv	empty
test93 :	funneling	-	direct routing	psend-precv	empty
test94 :	funneling	PvmDataDefault	daemon routing	send-recv	data
test95 :	funneling	PvmDataRaw	daemon routing	send-recv	data
test96 :	funneling	PvmDataInPlace	daemon routing	send-recv	data
test97 :	funneling	PvmDataDefault	direct routing	send-recv	data
test98 :	funneling	PvmDataRaw	direct routing	send-recv	data
test99 :	funneling	PvmDataInPlace	direct routing	send-recv	data
test100 :	funneling	PvmDataDefault	daemon routing	mcast-recv	data
test101:	funneling	PvmDataRaw	daemon routing	mcast-recv	data
test102:	funneling	PvmDataInPlace	daemon routing	mcast-recv	data
test103:	funneling	PvmDataDefault	direct routing	mcast-recv	data
test104:	funneling	PvmDataRaw	direct routing	mcast-recv	data
test105:	funneling	PvmDataInPlace	direct routing	mcast-recv	data
test106:	funneling	-	daemon routing	psend-precv	data
test107:	funneling	-	direct routing	psend-precv	data

4.1 Tests #1 through #44 and #47

These tests are general-purpose and cover the basic functionality of PVM. Notice that the tests #7 and #8 may indicate a failure without implying a PVM fault. In fact, such a failure means only that the host configuration is not *fast* enough to add or delete hosts.

4.2 Tests #45 and #46

These two tests do some Latency-Bandwidth measurement between the hosts of the configuration. These measurements are done according to the parameters set by the user for these two tests. However, the default parameters are :

- minimum length : 100

- maximum length : 1000
- progression : 100

The file `sample.input` shows how to set these parameters. In this file they are respectively set to 200, 2000 and 200.

These two tests generate an output. If the output file of the test engine is a file, then their output goes directly into this file, otherwise they go in two seperate files, `/tmp/output_45.dat.pid` and `/tmp/output_46.dat.pid`, where `pid` is the process-id of the test engine.

4.3 Tests #48 through #107

Each test in this section performs only one of three different patterns of communication:

- *Funneling* tests consist of many slave processes, each sending messages back to the master.
- *Head-to-head* tests consist of the master and only one slave, both sending messages to the other.
- *Triangle* tests consist of a master and two slave processes that communicate in a circular pattern.

These tests have different sets of the following options:

1. Use of XDR encoding, No encoding or Data in place
2. Use of direct or indirect routing
3. Use of `pvm_mcast()`, `pvm_send()` or `pvm_psend()`
4. Use of Not-empty or empty message

Besides, the default values for the parameters are :

- Number of messages : 10
- Length of messages : 100
- Number of processes : 5

In the file `sample.input` theses parameters are set to 3, 300 and 3.

The user can also get some timing results with these tests. Two results are available :

- Total time : Time to send and receive all the messages
- Average time : Average round-trip time per message

In the file `sample.input` these two flags are set to 1, which means that the total time and the average will both be printed when theses tests will be executed.

5 Various architecture combinations

This option is only interesting in the case of an heterogeneous network,. Each test requires some number of hosts, this implies that sometimes the test engine has to make a *choice* and pick some hosts in the host configuration. Basically, when a choice has to be made, if the option is turned off, the test engine picks the first hosts in the configuration file. If the option is turned on, it runs the test for some different configurations.

We could have chosen to try all the architecture combinations, but this would have been too much, leading to prohibitive execution times. The policy we have chosen is the following. The test engine runs a subset of the set of all combinations, provided that each pair of architecture is at least tried once in this subset. For instance, let's assume that we have 5 different architectures in our host configuration, let's say A, B, C, D and E. And suppose that the current test requires 3 hosts. Then the test engine will try the following combinations : ABC,ACD,ADE,BCD,BDE and CDE. This way, every pair of architecture is tried and the number of tries is only polynomial in the number of architectures and the number of hosts required.

All the tests requiring either one single host (the master host) or a number of hosts greater or equal to the number of different architectures in the configuration are not affected by this option.

The tester allows the user to turn this option on or off (see 3.1).

6 Functions not tested

Not all the PVM function are tested by the `pvm_test` program. The following PVM functions are used without being explicitly tested.

```
pvm_halt()
pvm_setopt()
```

The following PVM functions are not used at all :

```
pvm_catchout()
pvm_perror()
pvm_reg_hoster()
pvm_reg_rm()
pvm_reg_tasker()
```

7 MPPs

The PVM test suite has been installed and successfully run on :

- Connection Machine 5.

- Intel Paragon
- IBM SP2

There are three makefiles, `Makefile.cm5`, `Makefile.pgon` and `Makefile.SP2`, which allow you to compile and run the test suite on these three machines.

We have to be aware of something on this kind of MPP's. The master process of each test will run on the *host* and each slave program will run on one `node`. Thus, the tests #45 and #46 won't give the latency and bandwidth results for a node-node communication, but for a host-node communication. For an effective performance measurement, the user is advised to use the `nntime` program provided with PVM. Of course, the test engine tests the node-to-node copmmunication, in the *triangle* tests.

8 Customizing the test suite

It is rather easy to add new tests to the PVM test suite. Here follows a description, step by step, of what has to be done to add one test.

8.1 Creating a test descriptor

The file `module_list.h` contains the array of test descriptors. A descriptor is a structure containing :

- Short textual description of the test
- A structure `Flag` containing :
  - Flag *Do we execute this test ?* (filled at configuration time)
  - Number of hosts expected after the test
- The number of hosts needed before the test
- The status of the test
- A pointer to the master function
- A pointer to the slave function

Generally, the flag *Do we execute the test ?* is set to 1, since it always can be modified in the configuration file (see 3.1.2). The status of the test in this file has to be `NOT_YET_PROCESSED`, and it will be overwritten during the execution of the test, depending on the result. The numbers of hosts can be any integer number, `ALL_HOSTS` or `ALL_ARCHS`. `ALL_HOSTS` means that the test requires all the hosts of the configuration file. `ALL_ARCHS` means that the test requires one host of each architecture. The integer number can be greater than the real number of hosts. In that case the test engine will duplicate some hosts for the execution of the test.

For instance, we could add the test descriptor :

```
{"This is a new test",{1,ALL_HOSTS},0,NOT_YET_PROCESSED,
       (short (*)())new_test, (short (*)())new_slave}
```

This test will begin with all the hosts in the virtual machine and should have none left when it is completed.

8.2 Designing the test

Now, we have to write the master and the slave procedures. These two procedures have to be placed in the file `test_list.c` and to be declared in the file test_list.h.

The basic frame for the master is :

```
short   new_test(array_of_index,length)
int     *array_of_index;
int     length;
{
.....
char    *args[2];
char *test = "108";

args[0]=test;
args[1]=NULL;

......

pvm_spawn(TEST_SLAVE,args,..,...,...,..);

......
}
```

This may seem rather cryptic, without any explanation. The variable `array_of_index` is an array of integers. Each integer is the rank of a host in the virtual machine. This array is computed according to the number of required hosts for the test and one integer can be more than once in the array. `length` is of course the size of this array. The purpose of these two variable is to allow the designers of tests to be aware of the resources they have to use in the tests. The next step is to use the `hostpool` global variable. This is an array of host names. For instance `hostpool[array_of_index[2]]` is the name of the third host in the virtual machine during the test. Usually `hostpool[array_of_index[0]]` is the name of the master host.

The second part concerns the "spawning" of the slave(s). In fact, the master test spawns a *generic* slave, the name of which is `SLAVE_NAME`. This slave, according to the parameter `args` call the corresponding slave procedure (`new_slave`). The parameter passed to `SLAVE_NAME` is the rank of the test, that is the rank of the test descriptor in the file `module_list.h`. For instance here, 108 is the rank of the new test descriptor, since there are 107 tests in the original test suite.

The slave is very simple :

```
void new_slave()
{
....
}
```

The best way to understand how all this works is to have a look at the master and slave procedure of test #12 in `test_list.c`. These procedures are `test12()` and `test12_slave()`.

8.3 Updating the `NUMBER_OF_TESTS` constant

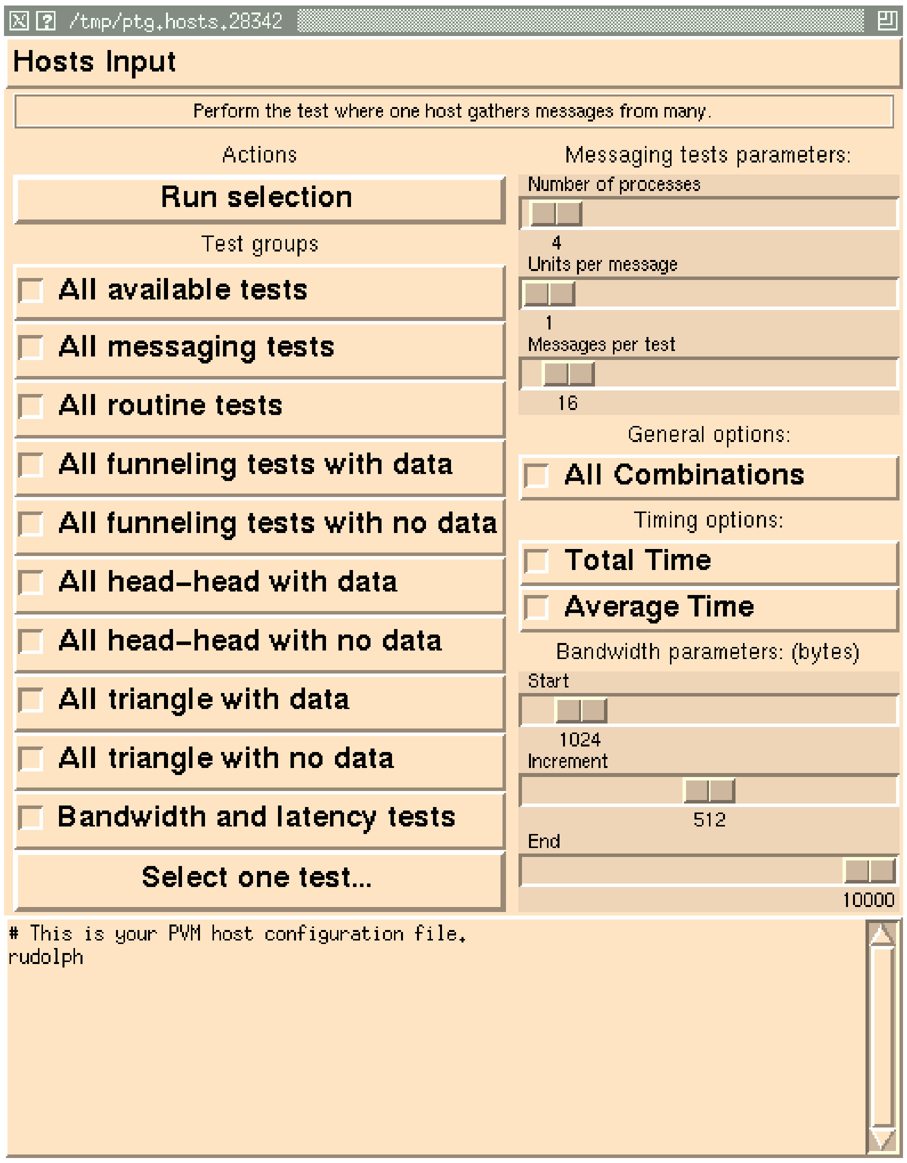
The last step is now to edit the `pvm_test.h` file and to increase by one the `NUMBER_OF_TESTS` constant. You can now run the tester and execute your new test suite, provided that the new test is written correctly according to these specifications !

9 Future work

- Updating of the tester for the next PVM release.
- Porting of the tester on more platforms.

10 Who to contact

Please send any bugs, questions and comments to `casanova@cs.utk.edu` and `mucci@cs.utk.edu`.



Perform the test where one host gathers messages from many.

Messaging tests parameters:
Number of processes 4
Units per message 1
Messages per test 16

Bandwidth parameters: (bytes)
Start 1024
Increment 512
End 10000

```
# This is your PVM host configuration file.
rudolph
```

Figure 1: The main window of pvm_test_gui