|epcc|

THE UNIVERSITY OF EDINBURGH

Portable Parallel Programming Profiling Tools: Vampir

Chris Johnson

January 27, 2003

1 Introduction

1.1 VAMPIR

Vampir (Visualisation and Analysis of MPI Programs) is a portable tool for profiling MPI codes written in Fortran (77 or 90) and C. It is used extensively on MPI systems at HPC centres all around the world, and is available on most on most of the UK's HPC resources. For example, Vampir is available on HPCx, the UK's new 1280-node IBM system.

The aim of this document is to provide an overview of VAMPIR and to describing the more interesting features of the system. A simple user guide is provided and the ease-of-use of the Vampir system described from the point of view of a new-user.

1.2 Downloading and Installation of Software

Vampir and Vampirtrace are both available from the Pallas website (http://www.pallas.com/). A free evaluation licence is available on both products and lasts one month. Downloading the appropriate binaries is straightforward and the software is presently available for most common platforms.

After downloading, the installation itself is easy due to the fact that for most platforms the software is downloaded as binaries.

1.3 How it works

In order to view a profile of MPI code using the Vampir software, both components, Vampir and Vampirtrace, are needed. A separate licence for both of these components

is required, but these licences can just be concatenated into the same file.

To compile the code, the VT (Vampirtrace) libraries must be linked in and an appropriate licence location given as an environment variable. If the code is compiled and run successfully, a tracefile is produced. This tracefile can then be read in by the Vampir tool, again with the appropriate licence set.

In order to get basic profiling information from a code, all that needs to be done is to link the code with the VT libraries at run time. Provided that careful note is taken to link the libraries in, in the correct order, etc. this should produce a tracefile simply by running the code. This tracefile will contain a profile of all MPI calls, without needing to add any extra calls to the code. This is the most basic use of the Vampir functionality but can still be very useful as it shows all MPI calls and messages.

If a more sophisticated analysis is required, then several more advanced features exist. All of which require some additions to the code. For example, it may be that it is not necessary for the entire code to be profiled. The tracefiles produced from Vampir are often fairly large and so if filespace is limited, it can be useful to produce a smaller tracefile. It is also true that the tracefile produces a lot of information including every MPI call and every message passed. This can cause the visual display to become cluttered and Vampir to run extremely slowly. Thus producing a smaller trace can make things clearer. In order to facilitate the tracing of subsections of code, VT provides call which switch on and off the profiling.

2 A basic user guide

2.1 Introduction

This user-guide is intended to introduce new users to some of the most useful features of Vampir. Our Sunfire 6800 SMP system is used as an example.

2.2 The Licence file

In order to make any use of Vampir or Vampirtrace, the path to the licence file needs to be set, this can be done by setting the PAL_ROOT environment variable to the place where vampir lives, *e.g.*:

```
chrisj@lomond$ export PAL_ROOT=/home/vampir/vampir/
```

This also has the advantage that $PAL_ROOT can be used as a short cut whenever reference to library or include paths are needed. It is a good idea to take account of this when installing vampir by placing links to the relevant executables in convenient places such that, for example, $PAL_ROOT/lib points to the Vampirtrace libraries, etc.

On some other systems the PAL_LICENSEFILE environment variable needs to be set explicity as well as the PAL_ROOT environment variable.

Any such environment variables will need to be set in any batch scripts used to run jobs.

2.3 Creating a tracefile

Creating a basic tracefile is straightforward. You just have to link in the VT library at link time. The exact format of the compile/link command depends on the system and the version of mpi. For example, for Sun MPI on a typical Sun system you also need to link in the nsl library. The order of the compile and link options is important. For example, for a 'hello world' Fortran file 'hello.f':

```
mpf90 -o hello hello.f -L$PAL_ROOT/lib -lVT -lmpi -lnsl
```

and for C 'hello.c':

```
mpcc -o hello hello.c -L$PAL_ROOT/lib -lVT -lmpi -lnsl -lm
```

Pallas also provide scripts when using MPICH (`mpicc, mpif77` and mpif90) which, after adjusting for the local environment, enable the user to add simply a `-vt` flag which automatically inserts the appropriate libraries and the library and include path names.

Assuming this compiles successfully, a tracefile should be produced when it is run:

```
chrisj@lomond$ mprun -np 4 ./hello
 hello world from process 3
 hello world from process 1
 hello world from process 2
 hello world from process 0
[0] Vampirtrace INFO: Writing tracefile hello.bvt
chrisj@lomond$
```

2.4 Viewing a tracefile

We can view the tracefile you have just produced using vampir. In order to view the 'hello world' tracefile, you just have to type:

```
chrisj@lomond$ vampir hello.bvt &
```

This should bring up a window containing the name of the tracefile (Fig. (1)) and another containing a summary chart showing the amount of time spent in MPI calls compared with the amount of time spent in the rest of the code (Fig. (2)).

There are several other useful features which are easy to use. For example clicking on the 'Timeline' from the 'Global Displays' menu gives a visual display of all the

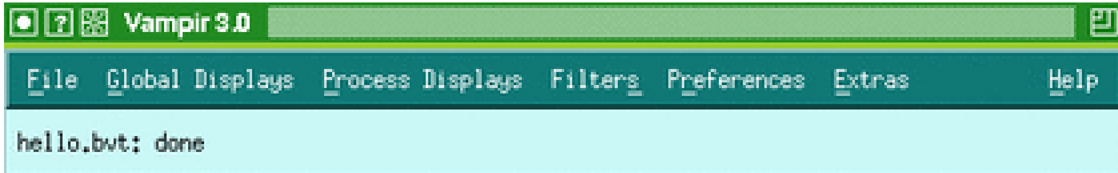


Figure 1: The main vampir window.

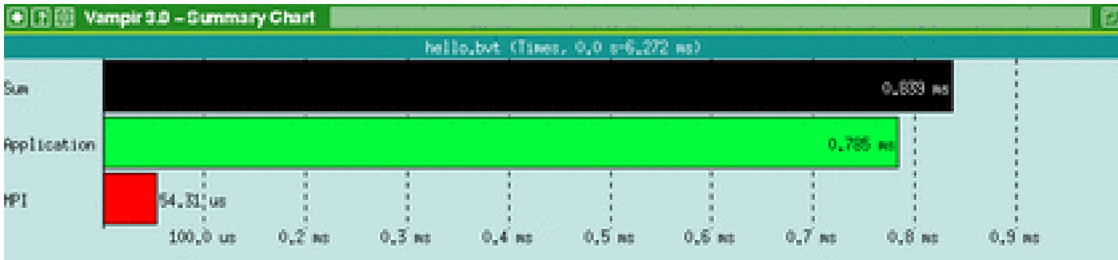
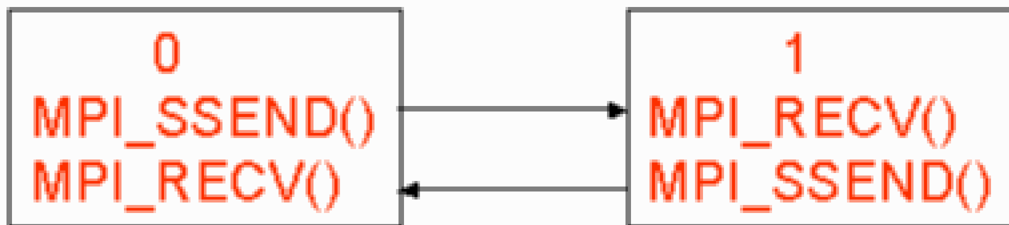


Figure 2: The Summary Chart.

processes involved which were executed when the code was run (Fig. (3)). Normal user code is given in green, and the MPI calls are shown in red. Communications are shown as black lines passing between the processes.

Fig. (4) displays the kind of trace you would see for typical ‘ping-pong’ message passing code:

```
CALL MPI_RECV(r_number, MAX, MPI_INTEGER, 0, 1,  MPI_COMM_WORLD,
      status, ierr)
CALL MPI_SSEND(r_number, MAX, MPI_INTEGER, 0, 2, MPI_COMM_WORLD,
      ierr)
```



and Fig. (5) shows what would happen for so-called ‘ping-ping’ code:

```
CALL MPI_SEND(r_number, MAX, MPI_INTEGER, 0, 2, MPI_COMM_WORLD,
      ierr)
CALL MPI_RECV(r_number, MAX, MPI_INTEGER, 0, 1, MPI_COMM_WORLD,
      status, ierr)
```

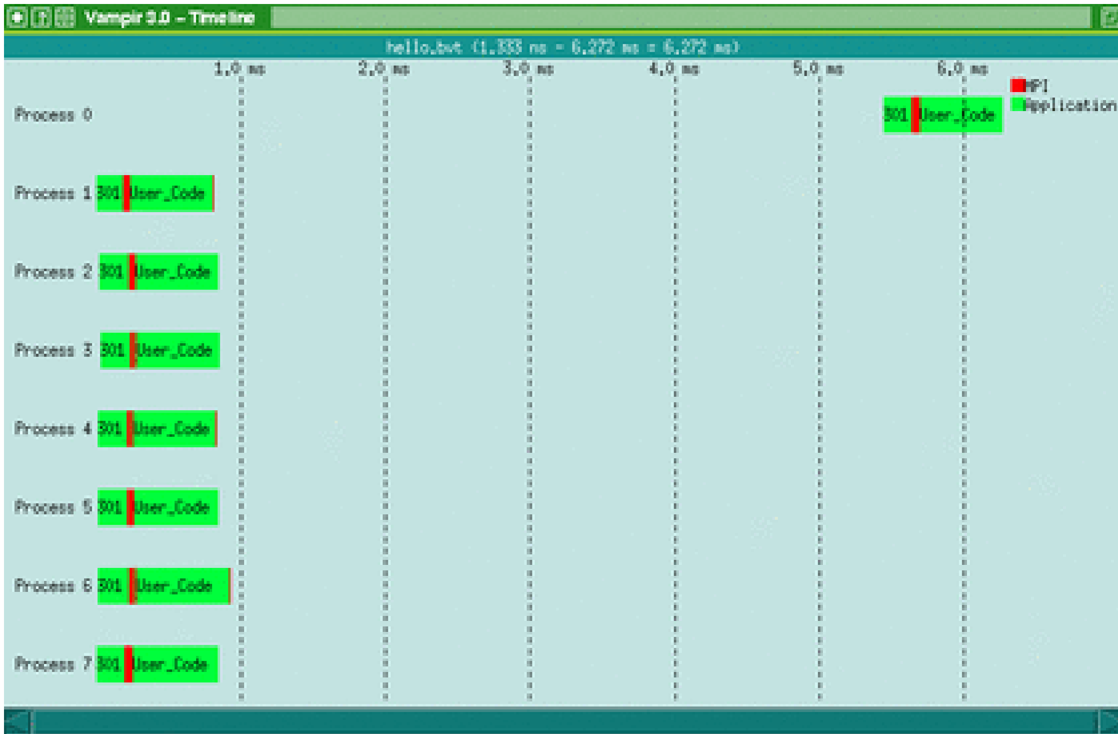
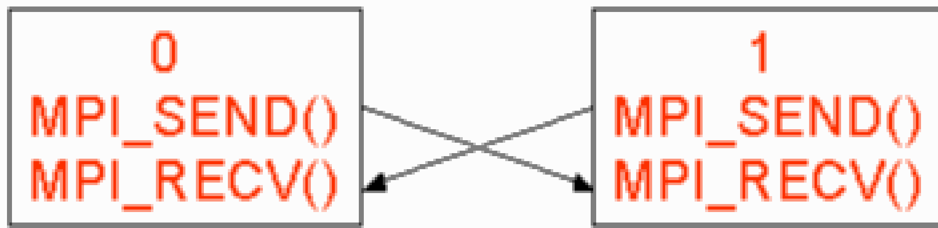


Figure 3: The Vampir Timeline showing all the processes.



Both were run on 4 processors and the figures show zoomed-in sections of the send and receive sections for each example.

3 Useful features

3.1 Timeline

The timelines which are shown in Figs. (4,5) are useful as an overview of what happens during the running of the code. The two figures demonstrate the facility to zoom in on certain sections of the trace. Without this facility it would be almost impossible to see what was going on in all but the simplest of codes. To zoom in on a particular section of the trace, all you need to do is select the region you wish to zoom in on. To do this, click on the left mouse button at the start of the section you wish to zoom in on and

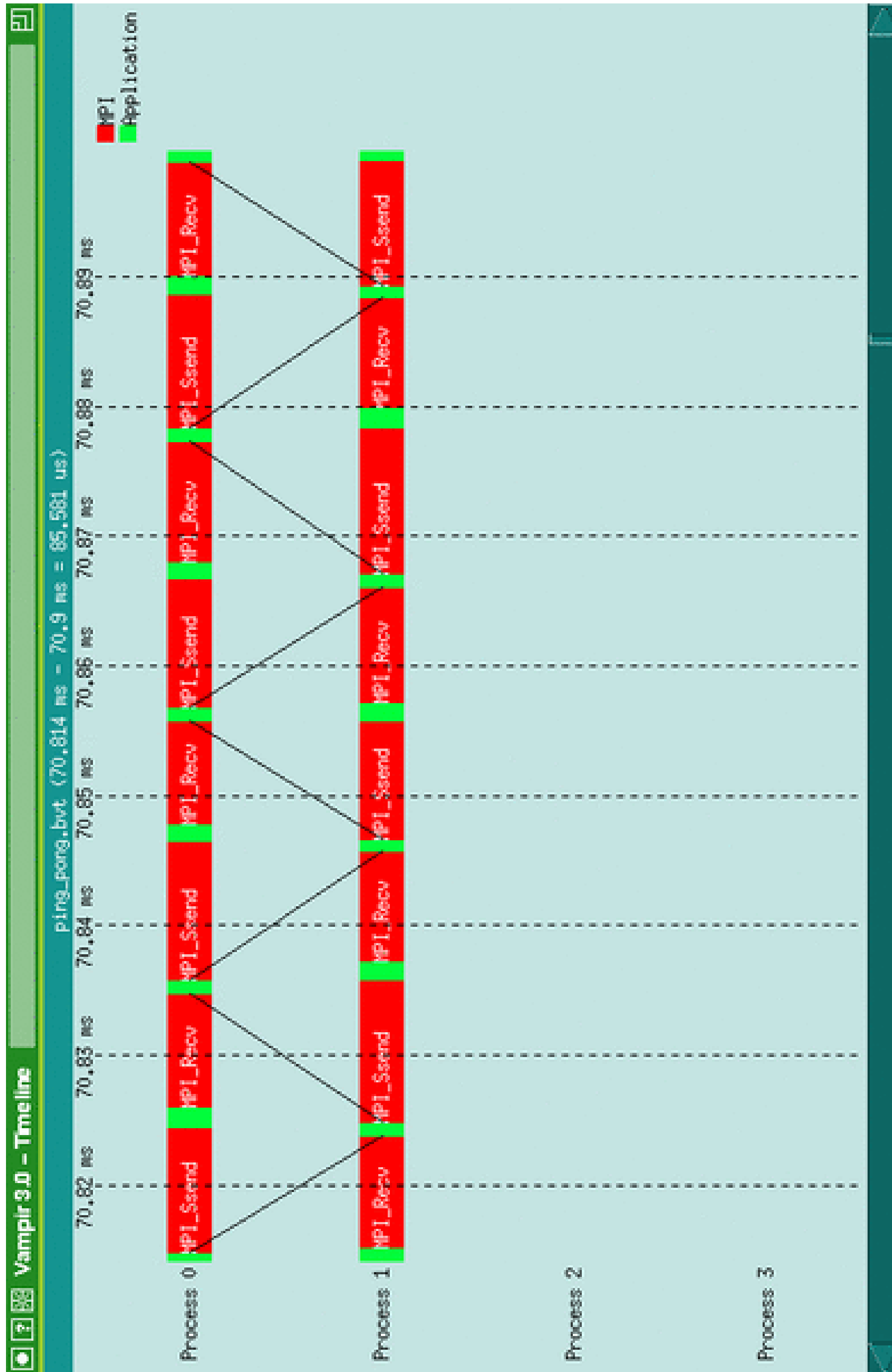


Figure 4: ping-pong

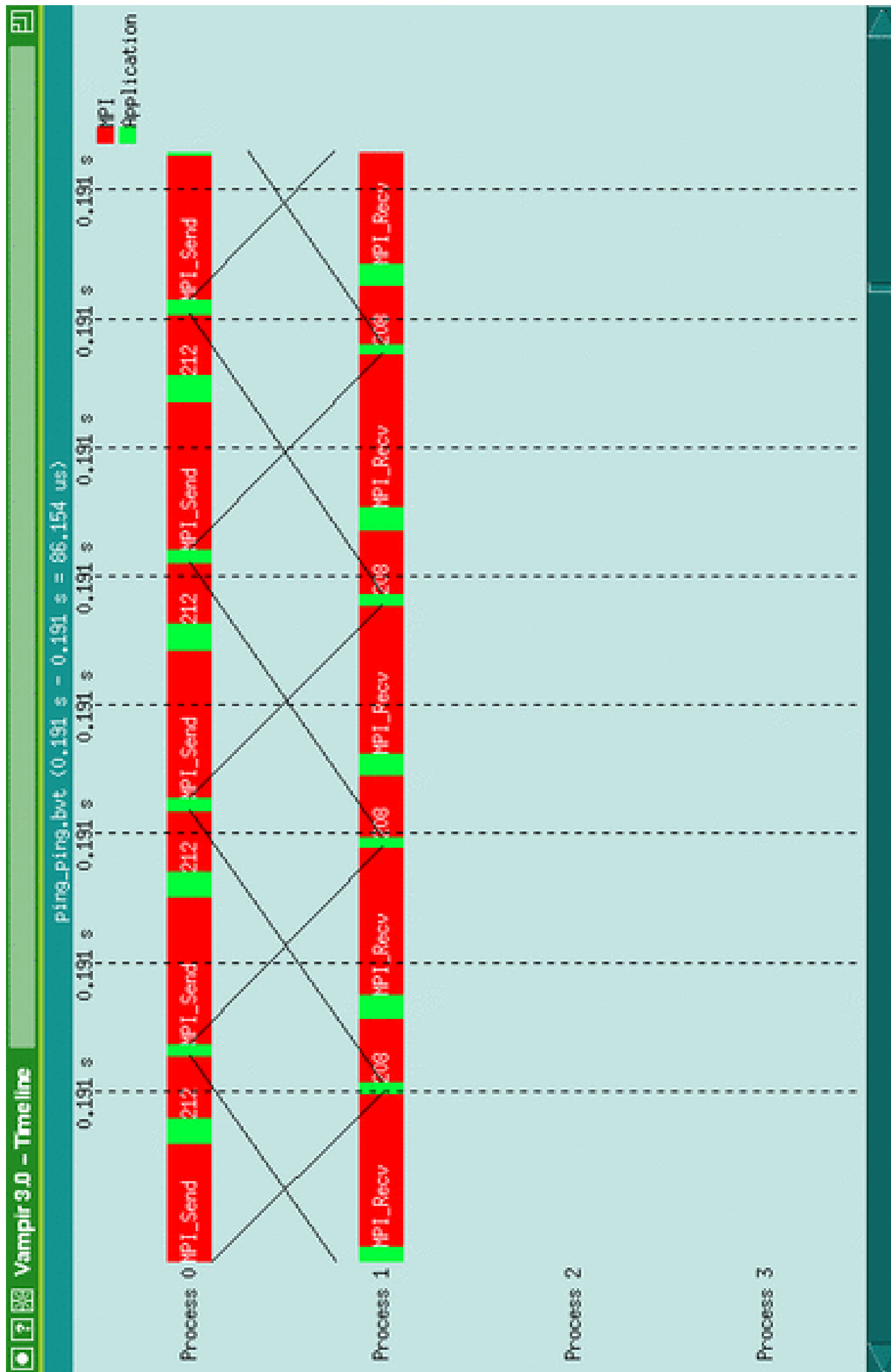


Figure 5: ping-ping

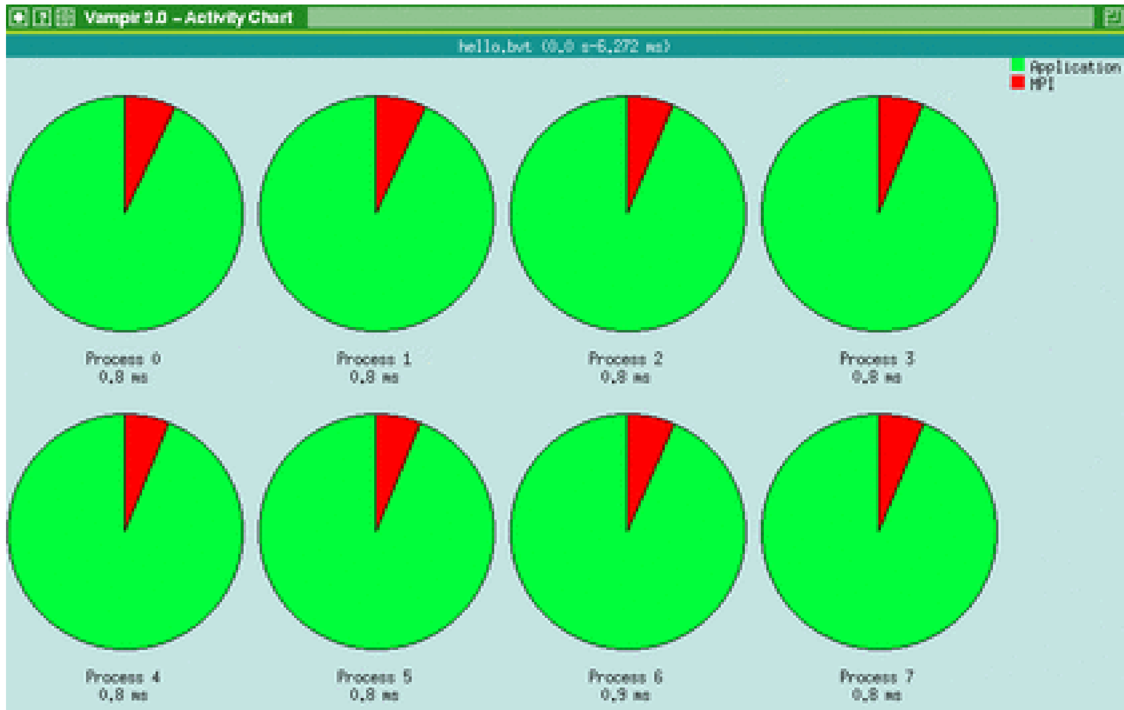


Figure 6: The Activity Chart showing all the processes.

then move to the end of the chosen section and let go of the mouse button. This should cause the timeline immediately to zoom in on the selected section. This zooming can be continued to an arbitrary depth. In order to ‘undo’ a zoom, you just have to type ‘u’. Left-clicking on individual sections of the timeline will bring up a box with some information about the appropriate call/communicator.

3.2 Activity Chart

Clicking on the ‘Activity Chart’ from the ‘Global Displays’ menu brings up a pie-chart of each of the processes (Fig. (6)) showing the relative amounts of time each has spent in MPI calls (red) compared with the rest of the code (green). After becoming more familiar with Vampir it is also possible to add other sections which will be shown in different colours.

4 Advanced Features

4.1 Labelling your code

Vampir offers many more advanced features, some of which require some additions to the code. For example, by default only MPI calls are given meaningful labels,

and all other code is just labelled 'User_Code'. In order to add labels to sections of your code, you need to add VT calls to define your labels (`VTSYMDEF()/int VT_symdef()` Fortran/C) and also to begin (`VTBEGIN/int VT_begin` Fortran/C) and end (`VTEND/int VT_end` Fortran/C) the sections you are labelling.

If you decide to add VT calls into your code, you will also need to include VT.inc (Fortran) or VT.h (C). You will need to add the include path in the compilation

```
-I$PAL_ROOT/include
```

For example - a section of code may look like this (Fortran):

```
      include 'VT.inc'

              ⋮

      CALL VTSYMDEF( 98, 'Init', 'Initialisation',ierr)
      CALL VTSYMDEF( 99, 'write', 'IO',ierr )

              ⋮

      CALL VTBEGIN (99, ierr)
      WRITE(*,*) 'starting run...'
      CALL VTEND (99, ierr)

              ⋮

      CALL VTBEGIN (98, ierr)
      a(0) = 0.0
      CALL VTEND (98, ierr)

              ⋮

      CALL VTBEGIN (99, ierr)
      WRITE(*,*) 'Final value of a = ', a(max)
      CALL VTEND (99, ierr)
```

or in C:

```
#include "VT.h"

              ⋮

      VT_symdef(98, "Init", "Initialisation");
      VT_symdef( 99, "write", "IO");

              ⋮

      VT_begin(99);
      printf("starting run...\n");
      VT_end (99);
```

```
        ⋮
VT_begin(98);
a[0] = 0.0;
VT_end (99);
        ⋮
VT_begin (99);
printf("Final value of a = %f", a[max]);
VT_end (99);
```

4.2 Activity Chart

Clicking on the 'Activity Chart' from the 'Global Displays' menu brings up a pie-chart of each of the processes showing the relative amounts of time each has spent in MPI calls (red) compared with the rest of the code (green). Of course this feature is more useful (and colourful!) if you have add extra labelling to your code (see §(4.1)). It is possible that the timing of your code is dominated by sections you are not interested in. In this case one can 'hide' various sections. Right-clicking one of the charts will bring up a menu. From this select 'hide' → 'Max' and then left-click on one of the charts. This will cause the biggest section to be removed from the statistics from *all* the charts. 'Reset' or 'Undo' will soon restore you to the previous display settings.

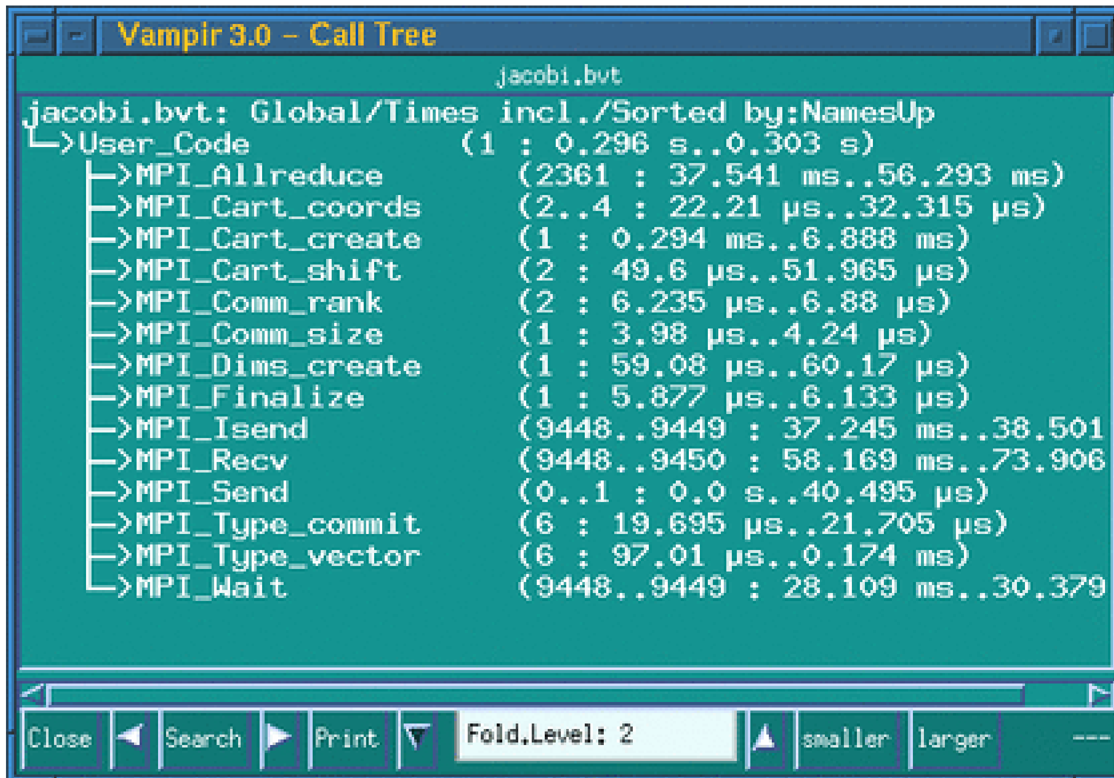
Alternatively, you may wish to display more information on the MPI calls themselves. In which case you can right-click 'Select' → 'MPI'. Now the charts show detailed information about the MPI calls. Again it is possible to 'hide' the biggest sections in the same manner as before.

It is easy to change the type of chart to a 'histogram' or a 'table' using the 'mode' option. This option is found by right-clicking anywhere on the activity chart

By default, the timeline displays information for the whole tracefile, but by selecting 'Use Timeline. Portion', you can ensure that only the section of time shown on the timeline is displayed on the Activity Chart. The option can also be set as a Global option where it affects all global and process specific displays. This is done by selecting 'Preferences' → 'General' → 'Use Timel. Portion'.

4.3 Call Tree

It can be very useful to view a call tree of all the MPI functions and, in particular, where they were called from. Bringing-up a call tree from the main Vampir window produces a diagram similar to the one below:



Again this window is more useful if labeling has been added to the code.

4.4 Communication Statistics

Clicking on either 'Message Statistics' or 'Collective Op. Statistics' from the 'Global Displays' menu produces windows giving cumlative information about the messages passed between each processor. Figs. (7,8) show the 'Message Statistics' and 'Collective Op. Statistics' windows respectively.

4.5 Filtering

We have already seen how to restrict the amount of information displayed on the timeline in a timewise sense, simply by zooming in on sections. However, it is also possible to restrict the number of processes displayed on the screen. This is done by selecting and de-selecting processes from the 'Processes' window found on the 'Filters' menu. Similarly the information can be filtered by message type.

4.6 Useful Extras

Several other features are mention here:

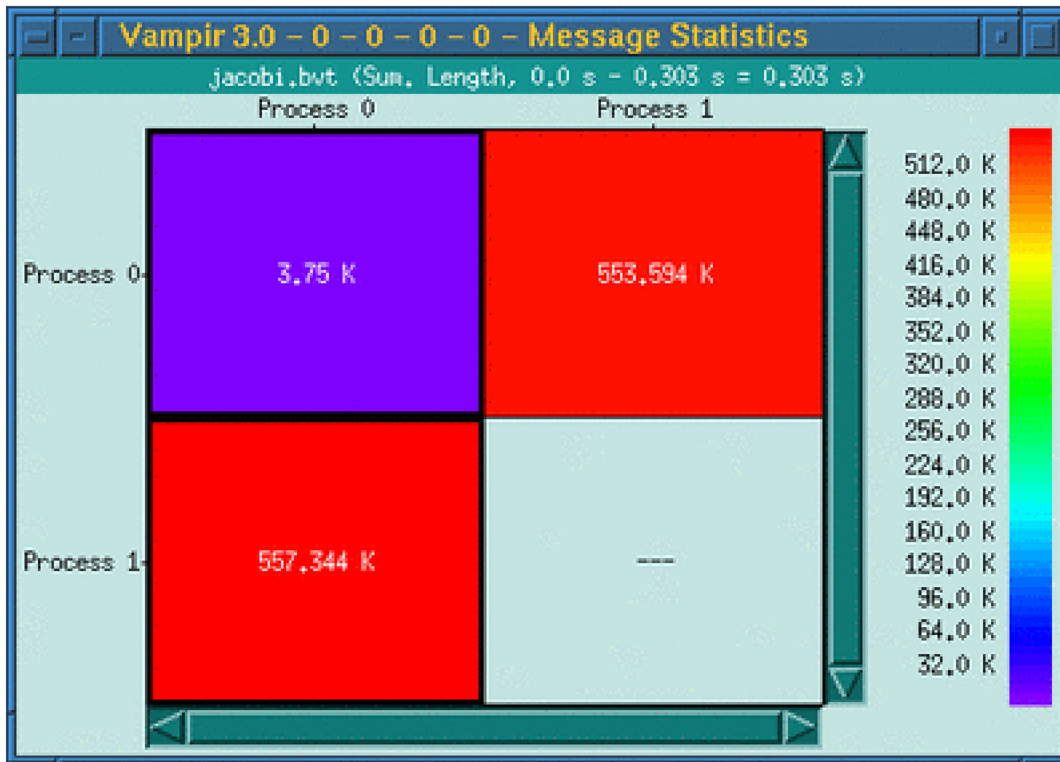


Figure 7: The 'Message Statistics' window

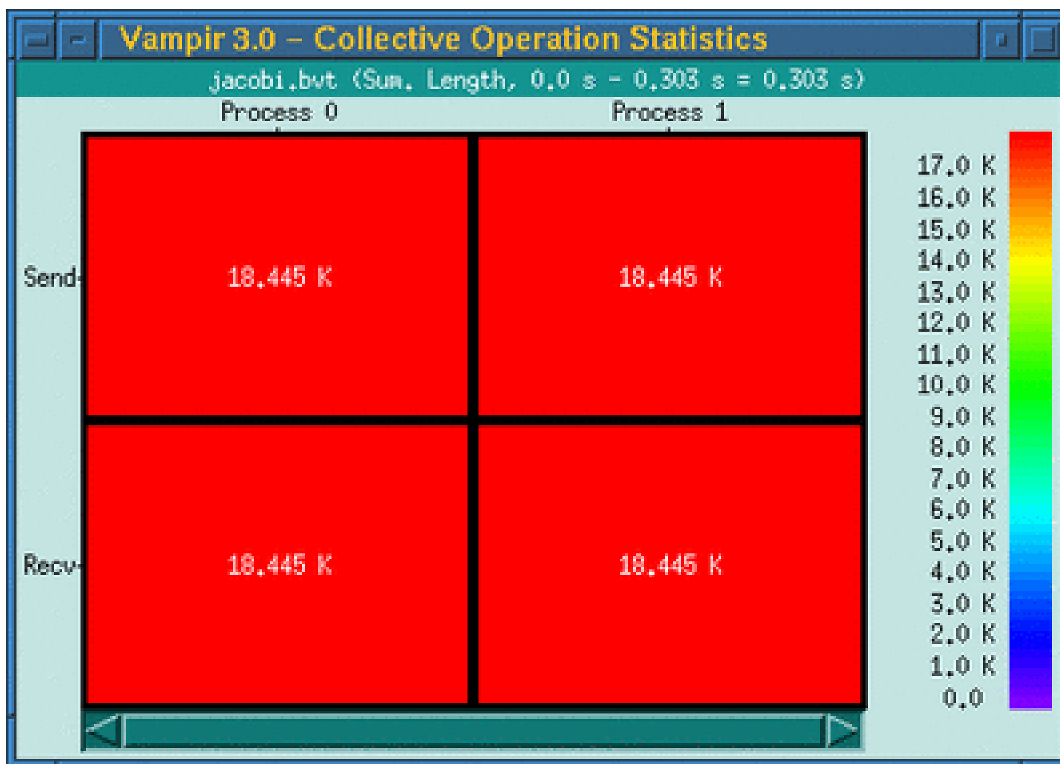


Figure 8: The 'Collective Op. Statistics'

- You can also read in gzipped '.gz' files - Vampir will automatically unzip them.
- You can instead link to VTnull (`-lVTnull`) - this will only compile if the VT syntax is correct, but will not produce any tracefile when the code is run.
- If you find that running with vampir causes the memory constraints to be exceeded (during execution, all VT information is kept locally in each processor's memory), you can 'flush' the tracefile to file using the command `'int VT_flush'` (C) and `VTFLUSH` (Fortran).
- You can arbitrarily switch the trace on and off using
`VTTRACEON()` (Fortran), `VT_traceon (void)` (C) and
`VTTRACEOFF()` (Fortran), `void VT_traceoff (void)` (C)
  - this must be called on each process on which you wish to switch the profiling off.
- When Vampir is run, a '.cnf' file is placed in your a .VAMPIR_defaults directory inside your home directory. This contains your local settings and is where any setting changes made while the code is executing are placed. It is possible to edit this file directly as well.

5 Useability

This section describes the 'ease-of-use' of vampir as found by the author who was a 'new user' to the Vampir and Vampirtrace software.

There are two documents which come with the Vampir installation. An 'Installation and User's Guide' and a 'User's Manual', hereafter refered to as 'document 1' and 'document 2', respectively.

Vampir and Vampirtrace were first installed on our Sun system, using a temporary 30-day evaluation licence. After a few problems were quickly overcome by the PALLAS company, which were specific to Sun MPI (MPICH worked imediately), the full 128-processor licence was purchased (our system consists of 52 processors).

The installations come as tar files for both Vampir and Vampirtrace, where the user can easily choose the most appropriate file for the target system (*e.g.* IBM-AIX, Sun-Sparc-Solaris, Intel Linux, etc.). It is worth noting that Vampir and Vampirtrace are completely separate products from the point of view of installing and licensing, and may even be run on completely separate systems with differing architectures.

After untarring these files, in both cases a simple directory structure is created with an install script which essentially does some checks and sets the permissions according to who will be allowed to use the software.

Vampir, the 'front-end' tool used to view the tracefiles created by vampirtrace, comes with at least one example tracefile so one can become familiar with the functionality

and capabilities of the system before even attempting to creating a tracefile for oneself. The author found this to be a useful facility as it also gives one a better idea of what a successful tracefile is supposed to do and what it should look like.

Document 2 gives a good description of most of the features of Vampir, along with adequate illustration consisting of screen-dumps of the software in action. In some cases the translation to English (from the original German) is a little confusing, but the illustrations help one to see what is going on.

Actually creating a tracefile is fairly straightforward as well. However, it is important to ensure that the correct linking flags are used, as these are very much system dependent. However, examples are given in the document 1 for compiling/linking with each of the supported systems.

As described earlier, in order to produce a trace file, it is not necessary to alter the code in any way. However, if a more detailed analysis of the code is required, it is possible to annotate the tracefile as described in the above user-guide (§(2)). This is where the Vampir 'API' comes in. The documentation refers to this as 'User-level Instrumentation', and whilst the documentation does not give and actual definition of the API, it is fairly easy to see that it just refers to any of the VT (Vampirtrace) functions or subroutine calls that the user adds to the code (see §(4.1)). It is also clear that in order for this to work, a header file (C: VT.h, Fortran: VT.inc), has to be added along with the appropriate include path.

The provided example had many such VT calls in and it is a good idea to follow the example and put the labelling definitions in a separate header file. In practice, Vampir really is by default only concerned with the profiling of message passing (MPI) calls. This means that it can be quite hard to work to perform a more general profiling of a code.

Vampirtrace does produce a lot of information for codes which have many MPI calls. For example, as a case study, the author attempted to produce and view a tracefile for a $\sim 10,000$ line code simulating the excited states of materials such as semi-condutors. This code has recently been parrallelised and is known not to scale well. It contains some 6 Gbytes of message passing in terms of collective communication calls, but no point-to-point communications. It typically takes about 10 minutes to run on 1 processor of our HPCx system and about 5 minutes to run on 32 processors of the same machine.

Without any modification, the size of the tracefile produced was proportional to the number of processors used and was approximately 6.4 Mbytes per processor which means 200 Mbytes for just a 32 processor run! It appears to be impractical to view a file of this size. As it stands it can take up to half an hour for vampir to process the tracefile of the 32-processor run and then any further graphical manipulation is also unwieldy. Pallas are presently developing a version of Vampir that allows for the user to load in only certain time segments of the code.

It is clear that presently some form of filtering needs to be done in order to produce

a useful tracefile in such a case. In the above case, for example, resticting the run to just 2 processors produces a perfectly managable tracefile. It would probably not be necessary to profile the entire tracefile of such a code and this is where Vampir has a very useful API function which can switch on and off the tracefile at specific points (see the user-guide section above §(2)). By making use of this 'switching-on and -off' of the profiler, the tracefile produced is then perfectly manageable.

Whenever any profiling tools is run on a piece of code, it is inevitable that some kind of overhead will be incurred. This is not usually be a problem as profiling tools would not normally be used on a production run. In the case of vampir, it appears that add a few VT calls to sections of the code, add simply a few percent to the overall run time. However, in the case of the above code, profiling the whole code added 20% to the overall runtime.

6 Conclusion

Overall, the author found the Vampir and Vampirtrace system to be a very instructive tool. It's use as a tool to visualise MPI, particularly for students learning MPI, is probably not stressed enough in the existing literature. During the last three months of usage, no stability problems were encountered with the Vampir (front-end) software and the addition of calls to the Vampirtrace libraries, during execution of the code, never caused the code to crash unexpectedly. As a profiling tool, Vampir is very useful, but the intensely visual display often results in it being necessary to scale down or 'sectionalize' the amount of code to be profiled as with most profiling procedures.

The documentation was on the whole good. Some updating of the manuals, including a revised translation, would be useful. It would probably be better for the two manuals to combined into one single manual as it is not always clear what their separate purposes are.

Whilst it is possible to insert VT calls to dump trace information out at any stage in the running of the code, this data is not collected together unless MPI_Finalize is called. It is a shame that the data for each section could not be collected together by the Vampir program. This way it would be possible to run Vampir on code which did not complete for whatever reason.

During the last three months of usage, no stability problems were encountered with the Vampir (front-end) software and the adition of calls to the Vampirtrace libraries, during execution of the code, never caused the code to crash unexpectedly.

Another document [Smi00] exists which compares Vampir with other similar profiling tools such as the Sun tool Prism [Pri]. Debugging tools are also compared.

References

[Pri] Prism. Part of the sun hpc 3.0 software, sun microsystems, inc palo atlo, ca.

[Smi00] L. Smith. Comparison of code development tools on clusters. Technical report, Edinburgh Parallel Computing Centre, The University of Edinburgh, 2000.