VAMPIR: Visualization and Analysis of MPI Resources

W.E. Nagel, A. Arnold, M. Weber
Central Institute for Applied Mathematics
Research Centre Jülich (KFA)
D-52425 Jülich, Germany
({w.nagel,a.arnold,m.weber}@kfa-juelich.de)

H.-Ch. Hoppe, K. Solchenbach
PALLAS GmbH
Hermülheimer Str. 10
D-50321 Brühl, Germany
({hch,karls}@pallas.de)

Abstract

Performance analysis most often is based on the detailed knowledge of program behavior. One option to get this information is tracing. Based on the research tool *PARvis*, the visualization environment *VAMPIR* was developed at KFA which now supports the new message passing standard *MPI*. *VAMPIR* translates a given trace file into a variety of graphical views, e.g., state diagrams, activity charts, time-line displays, and statistics. Moreover, it supports an animation mode that can help to locate performance bottlenecks, and it provides flexible filter operations to reduce the amount of information displayed. The most interesting part of *VAMPIR* is the powerful zooming feature that allows to identify problems at any level of detail.

1 Introduction

On massively parallel computer systems, performance analysis and debugging can become an extremely complicated process. Over the years, experience has shown that user-friendly tools supporting this process are extremely helpful and can drastically shorten time-to-solution for a given problem. The complications arise because of the fact that traditional methods used on sequential computers like profiling or debugging step-for-step execution either deliver not enough information or present too much intrusion. A method that has proven usability to a certain degree is *tracing*. The structure of a typical tracing system is shown file=color/tracing.eps, width=10cm

Figure 1: Principle of tracing

in Fig. 1.
Tracing is based on *instrumenting* a program before it is executed. Instrumentation extends parts of the program specified by the user in a way that data records are written into a protocol whenever these parts are executed. The records usually contain a time stamp, the number of the processor that has generated the record, an event type identifier, and a list of additional parameters depending on the event's type. Events that can be instrumented could be subprogram entries and exits or the sending or receiving of a message. Intrusion is reduced by writing the data records into a buffer located on the processor's local memory. I/O activity only takes place when this buffer overflows and has to be flushed to disk. After program end, the individual record streams are merged into a single stream that is sorted chronologically. Analysis can then be done off-line.
The problem of tracing is the large amount of data usually generated. Especially, when a program is traced for the first time, it is not known which parts of the program will be of interest; most people will enable all tracing options which quite often result in very huge trace files. Therefore, there is a need for a flexible and powerful tool that enables the programmer to quickly get an overview of the program run without disabling analyzation on the level of single events.
This paper describes the X Window based visualization environment *VAMPIR* which has been developed at KFA Jülich to support performance analysis of parallel programs. Like most of the other performance analysis tools available for parallel computers (*Paragraph* [Int93] or *Pablo* [Ree92]), *VAMPIR* is used on a post-mortem basis, and it translates a given trace file into a variety of graphical system views which provide a reasonable basis for system understanding and program optimization. *VAMPIR* is based on the visualization environment *PARvis* [Arn93, NaAr93, NaAr94, Mue95] running on a large variety of workstation platforms. It has been extended to support additional panels and filter functions for the new message passing standard *MPI*.

2 The Message Passing Interface (MPI)

The growing interest in parallel computing, and notably in the message–passing programming model, pushes the demand for a standardized application programming interface supported by all major parallel system vendors. Starting in 1993, a group of computer vendors, library writers and application programmers from the US and Europe collaborated to design a standard portable message–passing interface called MPI. The final specification of this interface was published in May 1994 and updated in June 1995 ([MPI95]); [GLS94] gives a good

introduction from the application programmer's point of view.
A number of portable and vendor–specific MPI implementations have since been developed, showing that MPI can indeed be implemented efficiently on the currently available parallel computer platforms. There are three public–domain implementations of MPI, and most parallel system vendors have announced MPI implemetations of their own.
MPI draws from a number of other message–passing interfaces, including IBM's EUI, PVM, Intel's NX, and PARMACS, adding some advanced features:

- Communication modes: MPI supports a multitude of point–to–point communication modes, some allowing to overlap communication and computation.
- Data types: in a message, MPI transmits objects of a specified datatype ranging from predefined elementary types to complicated, non–contiguous user–defined datatypes. Thus, programs do not need to know about datatype sizes, and automatic type conversions can be done on heterogeneous systems.
- Communicators: to isolate different communication spaces, the concept of a *communicator* was introduced into MPI. Messages must be sent and received within the same communicator, thus allowing to encapsulate the communication done by a parallel library from the application.
- Collective communication: MPI contains a complete selection of global communication operations including broadcast, reduction, gather and scatter on arbitrary datatypes.

Therefore, MPI enables portable programs and libraries to be written. Of course, mere functional portability is not sufficient in practice: efficiency of an application or library must be the second focus of interest. In spite of using a standardized interface, parallel programs will not show equal performance on different hardware platforms — same as with sequential programs. Thus, careful adjustments – performance tuning – are necessary to optimize a parallel application for a given parallel system.
For parallel programs on massively parallel systems, performance tuning is much more complicated than in the sequential case, because additional system parameters like the ratio of computational power to communication speed come into play, and currently no automatic tools analogous to optimzing compilers are available. To reap the maximum benefit from MPI, powerful and easy to handle performance analysis and visualization tools are of increasing importance.
Users working with different message-passing libraries on several parallel systems will not have the time to fully understand and tune their message-passing codes for every platform. With the dissemination of *MPI* this will probably change. The powerful features of *MPI* will offer a range of flexibility that allows to get the maximum performance from any kind of parallel hardware supporting message passing. To achieve this in a convenient way users will ask for tools

able to display the communication structure of their programs at almost every time scale. *VAMPIR* will have the functionality to satisfy these demands.

3 The VAMPIR Environment

Performance analysis and program optimization are often based on different categories of system views (Fig. 2, [Mue95]):

- single time system snapshots: panels that show system activities at a particular point of time;
- animation: option to look at a sequence of system snapshots to investigate the dynamic behavior;
- statistics: the component that summarizes system behavior for the time under investigation;
- time-line system view: detailed view of system activities, which are visualized on a time axis.

Each category is supported by the VAMPIR environment; the current prototype can generate traces on the Intel iPSC/860, Intel Paragon and CRAY T3D systems, whereas the product version will work for any standard–compliant MPI implementation.
For user convenience, VAMPIR provides a configuration file where user preferences (color, layout, fonts etc.) are stored between runs. This file enables the tool to come up with the exact same settings of a previous session, and different configurations may be saved and loaded at will. A detailed description of all VAMPIR features can be found in [Arn93, ArRo95, Mue95].
VAMPIR is implemented in ANSI C and uses the OSF/Motif libraries. The current implementation already supports a variety of different hardware platforms (IBM RS/6000, Sun Sparc, DEC MIPS computers (Ultrix), DEC Alpha, HP, and Silicon Graphics).

4 Program Instrumentation

The MPI standard ([MPI95]) specifies a profiling interface that every standard–compliant MPI implementation must provide. Using this interface, wrapper routines can be registered that trace the execution of every MPI routine.

file=color/systemview.eps, width=10cm

Figure 2: The VAMPIR visualization options

VAMPIR provides a TraceGenerator library on top of this profiling interface to generate traces of MPI communicator, point–to–point and collective communication routines. This part of VAMPIR can work with each standard–compliant MPI implementation, and supports both C and Fortran 77 applications.
To trace additional information like subroutine entry/exit, the *PARvis.inst* instrumentation tool for Fortran 77 has been developed at KFA Jülich based on the *Paff* [Ber89] preprocessor. The command

```
PARvis.inst [options] file_name [file_name]
```

automatically instruments the Fortran 77 programs specified on the command line. Flexible options are provided to generate wrapper routines for system and application routines, and tracing of a particular routine can be switched off by just marking that routine as non–traceable. Control directives are supported to start and stop the trace gathering, and in addition an upper bound on the tracefile length can be specified by the user. All directives start with the prefix `CKFA$ TRACE`.
For C applications, a library interface to the TraceGenerator will be supplied that allows to insert instrumentation instructions either manually or with the help of the C preprocessor.

5 Visualization of System Activities

The system activities can be visualized in the *Global_Display/Node Style* display (Fig. 3). Here, every processor is displayed as a box. The size and arrangement of the boxes depend on the number of processors and the geometry of the window and are automatically calculated by *VAMPIR*. Each box is partitioned into a lower and an upper part. The lower part describes the current activity on the nodes, whereas the upper part (called *statistics field*) shows the time portion (in percent) spent on a particular activity for the period under investigation (here: *Calculation*). For monitoring reasons, the background color reflects the current value printed out, and the corresponding percentual values are listed on the right.

file=color/sysact.eps, width=15cm

Figure 3: System activity profile

Fig. 3 represents one example of an actual system snapshot at a special point of time. The *Step*-button in the *VAMPIR Move Control* area can be used to show the system activity changes. Typically, the number of events to be displayed is rather large, so the animation mode can be used to animate the sequence of system snapshots. The step width for the animation mode can be either an event or a given time period; the time difference between two movements (i.e., the animation speed) and the number of movements after which the animation should stop can be adjusted in the panel *Settings/Steppings*. This animation feature can be used to analyze the program behavior in time, to identify critical program sections, and to find the hot spots of the run.

6 Statistics

The Node display mode already contains a small statistics field, but due to its limited size only the time portion of *one* state can be monitored. Quite often, one would like to get a more detailed idea of how the time is spent on each of the nodes. To analyze the complete state distribution, it is possible to switch the display mode to the *statistics display*. Press F6 or select the menu option *Global_Display/Chart Style*, and another window will come up (Fig. 4), which shows a statistics of the complete trace file in a pie chart style. The colors chosen for the individual states are just the same as those which are used as the background color for the state field in the CPU display. The most important activities can be identified for all nodes, and differences in the node behavior will be clear immediately. As can be seen from that panel, most time is spent in *Calculation* on all nodes, and significant portions of time are also spent at a barrier.

When a lot of CPUs are involved in a parallel system, the individual statistics in this display can become very small and uninformative. To relieve this unfortunate situation, *VAMPIR* can open additional windows containing statistics for only *one* CPU. To select the CPUs you want statistics for, simply click at them with the left mouse button, and their frame color will be inverted. You can also drag over a couple of CPUs to select several CPUs with one action. In the example shown in Fig. 4, the actual time distribution spent in user subroutines (*Calculation*, pie chart in the left sub-window: most time is spent in subroutine VELO) as well as for node communication (*Communication*, histogram in the upper right sub-window), and Paragon emulation (*Paragon*, histogram in the lower right sub-window) is shown for node 2. The user can toggle between table, pie chart, and histogram in all chart windows. The histograms may be linear or logarithmic, and zooming is supported.

file=color/stat123.eps, width=13cm

Figure 4: Time distribution statistics for the program run

7 System Activity Profile

The statistics windows only show the accumulated breakdown of system activities over the whole program run. Nevertheless, the program may have different phases where the timing behavior of system activities is quite different. These phases can be identified either by looking for differences during an animation run or by viewing the window *system activity profile* (Fig. 5). This window shows the actual time distribution for all system activities in each time step accumulated over all the nodes; and problem areas where, for example, *calculation* has low values, can be seen immediately.

file=color/actprof.eps, width=10cm

Figure 5: System activity profile

8 Time-line Displays

Based on the data visualization options presented above, we now concentrate on the interaction of parallel activities and possible bottlenecks. At this point, the user is interested in seeing a sequence of activities on all nodes, and the interdependences between these different program parts.
The problem with most other visualization tools like *Paragraph* [Int93] or *Pablo* [Ree92] is that these tools are based on the *Replay Technique*: Whenever the user wants to have just another information about a special part of the program, the whole trace file is analyzed once again, even if the file contains several hundreds of Mbytes (see Fig. 6). The magnification glass has to scan the whole trace file several times whenever the user would like to see a different information or just another time frame.
This is different in the *VAMPIR*-environment: here, the user can specify the size of the magnification glass, and all details within the magnification glass can be seen without any further I/O-activity (Fig. 7). For example, statistics for all activities inside the chosen time window can be generated within milliseconds. Moreover, the user can use a powerful zooming feature to analyze the program behavior on any level of detail; each zoom-operation also takes only a few milliseconds, even if several Mbytes of tracing information are under investigation. Of course, a hierarchical *unzoom*-operation is provided for user convenience.

file=color/zoom1.eps, width=9cm

Figure 6: Zooming and the *Replay Technique*

file=color/zoom3.eps, width=9cm

Figure 7: VAMPIR realization: Make zooming as easy as possible

In *VAMPIR*, the *Global_Display/Timeline* panel is used to display this type of information. As can be seen from the upper part of Fig. 8, colors are used to represent different kinds of activities, and it is possible to show system activities over time on each of the nodes. In this example, the program is running in phases where the subroutine VELO is executed several times. The black parts are hundreds of messages (represented by one line each) which are sent between the nodes. Based on the information displayed in this window, it is quite easy to identify critical program sections where problems may have occurred.
The zooming feature can now be used to go into detail. As shown in the middle part of Fig. 8, the period of interest[1] (400 — 560 ms) was zoomed-in by just specifying the time frame with the mouse. Here, one of the time–step iterations can be seen, and the load imbalance causes long synchronization times at the barrier called GSYNC.
The zooming feature also can be used to get deeper and deeper into the analysis process, to understand program behavior, and finally to identify problems. The lower part of Fig. 8 shows a data communication exchange part of the program (at about 525 ms) where different communication patterns inform the user about his communication activities. In the message passing programming model, communication and data exchange are solely based upon the sending and receiving of messages. Regardless of the network's topology (which is hidden to the application programmer in most cases), it is obvious that the visualization of message transfers and patterns plays an important role in the performance analysis and debugging of parallel programs. Therefore, *VAMPIR* includes means to display and inquire information about message-passing transfers. These tools are not isolated from the other part of *VAMPIR*: message events are read through the same trace file interface into *VAMPIR*, and the message visualization tools work hand-in-hand with the features described so far. It is possible to mouse-click a message that pops up another panel showing all information related to that message, including the transfer rate in MByte/s (i.e. about 20 MByte/s). The information for this message is coming out of the wrapper of the *MPI_SEND/MPI_RECV* communication routine, and the overhead involved is quite low.
Moreover, detailed information about the activities on one node or a selection of nodes can be obtained. The lower left part of Fig. 8 documents that even calls to *gdhigh* (a few microseconds inside the communication library routine) easily can be identified. A case study on Intel Paragon [WiNa94] describes a situation where the *VAMPIR* environment was extremely helpful in identifying performance bottlenecks in the communication library; based on the optimiza-

[1] The time offset is specified in the lower left corner of the panel.

tion process, the output performance (*hippi-output*) was increased by a factor of more than five within a few hours.
In addition, the zooming operation can be used to identify typical communication patterns. It is obvious that the visualization of such communication patterns gives knowledge about implementation aspects of the system and of your own program, and it is very helpful to understand synchronization delays and related side effects which sometimes significantly influence the performance of real applications.
For example, Fig. 9 shows a zoom-up of a program that uses *MPI* calls for a reduction and a broadcast operation (some of the 64 processors have been filtered for a clearer display). One can easily see the communication patterns the *MPI* libary uses internally for a broadcast. Moreover, by clicking at a message, one can identify the communication IDs and types which the *MPI* implementation uses internally. The ability to look into the implementation in this way is a key feature to understand why programs that use a standardized message-passing library like *MPI* behave differently on different machines.
To clarify the display, *VAMPIR* allows to assign different colors to messages of a specific type or communicator range. Alternatively, messages of certain type or communicator ID can be removed completely from the display if only a certain subset of messages is of interest. The length of messages can be visualized by a different line width for shorter and longer messages.
To evaluate the overall message traffic that took place over a period of time, a matrix of communication can be opened (Fig. 8) that shows different statistic values for the messages that were passed between each pair of sender and receiver. Specifically, the following parameters can be shown:

- The total number of messages passed between the processors
- The total number of bytes passed between the processors
- The maximum, minimum and average length of messages
- The maximum, minimum and average data rate that was reached

This display simplifies the detection of unbalanced communication and performance reductions because of too many short messages, what usually results in a low average data rate.

9 Additional Features

VAMPIR accesses several external tools to perform some of its tasks. These tools must be located in a directory included in your *PATH* environment variable:

- *lpr* or *lp*, the standard UNIX printing facilities, to print lists and window snapshots.

- *import*, a screen snapshot utility from the *ImageMagick* package to export or print window contents. *ImageMagick* is delivered as part of VAMPIR; it can also be downloaded from *ftp.zam.kfa-juelich.de*, directory *pub/graphics/ImageMagick*.

- If you have trace files compressed with *gzip* or *compress*, *VAMPIR* can extract them automatically if their counterparts *gunzip* or *uncompress*, respectively, are available.

There are quite a few other enhanced features that cannot be described in detail in this paper; the most important ones are mentioned below:

- filter functions: *VAMPIR* allows to simultaneously display up to 512 nodes. Typically, this number is much too large to be handled meaningfully; therefore, powerful filter functions are available to reduce the number of nodes, either automatically or manually by the user.

- movie support: after each animation step, control is optionally given back to a user-command (i.e., a shell script). This allows to generate movies unattended by the user, just by specifying a single command in a subpanel.

10 Summary and Conclusions

This paper describes the *VAMPIR*-environment which provides some powerful features to discover parallel program behavior on several parallel systems like Intel Paragon and CRAY T3D. Experience has shown, that for debugging, as well as for performance optimization purposes, the supported time-line displays in combination with the statistics features are the strength of the system. With the extremely flexible zooming function in the time-line displays, analysis operations are supported which can drastically improve the understanding of observed performance problems.
VAMPIR is available as a commercial product from PALLAS GmbH; for further information, see the WWW page `http://www.pallas.de` or send mail to `info@pallas.de`.

References

[Arn93] A. Arnold, *PARvis: Eine X-basierte Umgebung zur Visualisierung von parallelen Programmen in Multiprozessorsystemen*, Jül-2848, Forschungszentrum Jülich (KFA), 1993.

[ADN95] A. Arnold, U. Detert, and W.E. Nagel, *Performance optimization of parallel programs: tracing, zooming, understanding*, In: Proc. Spring 1995 Cray Users Group Meeting, pp. 252–258.

[ArRo95] A. Arnold and M. Röth, *PARvis - an X-based visualization environment for parallel programs (User's guide)*, Forschungszentrum Jülich (KFA), to be published.

[Ber89] R. Berrendorf, *Der FORTRAN-Parser PAFF als wiederverwendbares Modul für Programmier-Tools*, Jül-Spez-537, Forschungszentrum Jülich (KFA), 1989.

[Int93] *Paragon application tools user's guide*, Intel Corporation, 1993.

[MPI95] Message-Passing Interface Forum, *MPI: A Message–Passing Interface Standard*, Version 1.1, University of Tennessee, Knoxville, Tennessee, 1995; Available as `ftp://ftp.mcs.anl.gov/pub/mpi/mpi-1.jun95/mpi-report.ps`.

[GLS94] W. Gropp, E. Lusk and A. Skjellum, *Using MPI.* MIT Press, Cambridge, Massachusetts, 1994.

[Mue95] Ch. Müllender, *Visualisierung der Speicheraktivitäten von parallelen Programmen in Systemen mit virtuell gemeinsamem Speicher*, Jül-2911, Forschungszentrum Jülich (KFA), 1994.

[NaAr93] W.E. Nagel und A. Arnold, *PARvis: Ein Werkzeug zur Visualisierung von parallelen Prozessen auf Mehrprozessorsystemen*, Proc. 7. ITG/GI Fachtagung MMB'93 (Kurzberichte und Werkzeugvorstellung) pp. 178–187, 1993.

[NaAr94] W.E. Nagel und A. Arnold, *Performance visualization of parallel programs: The PARvis environment*, In: Proc. 1994 Intel Supercomputer Users Group Conference (ISUG'94), pp. 24–31.

[Ree92] D.A. Reed, R.A. Aydt, T.M. Madhyastha, R.J. Noe, K.A. Shields, and B.W. Schwartz, *An overview of the Pablo performance analysis environment*, Technical Report, Dept. of Computer Science, University of Illinois, Urbana-Champaign, 1992.

[WiNa94] R. Williams and W. E. Nagel, *Optimization of output bandwidth from a Paragon*, Technical Report CCSF-44, Caltech Concurrent Supercomputing Facilities, Pasadena, CA, 1994 (also on WWW: http://www.ccsf.caltech.edu/ roy/hippipap/paper.html).

file=color/timelall.eps, height=20cm

Figure 8: Time-line zooming and message identification

file=color/id64.eps, width=13cm

Figure 9: Identification of MPI messages

file=color/mp$_s tat.eps, width = 10cm$

Figure 10: Statistics of the message passing communication rate