Available at www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

ELSEVIER

Future Generation Computer Systems 19 (2003) 735–747

FGCS FUTURE GENERATION COMPUTER SYSTEMS

www.elsevier.com/locate/future

Flexible performance visualization of parallel and distributed applications☆

J. Chassin de Kergommeaux [a,*], B. de Oliveira Stein [b]

[a] *Laboratoire Logiciels Systèmes et Réseaux (LSR-IMAG), BP 72, 38402 St. Martin d'Heres Cedex, France*
[b] *Departamento de Eletrônica e Computação, Universidade Federal de Santa Maria, Santa Maria, RS, Brazil*

Abstract

Performance debugging of parallel and distributed applications can benefit from behavioral visualization tools helping to capture the dynamics of the executions of applications. The Pajé generic tool presented in this article provides interactive and scalable behavioral visualizations; because of its genericity, it can be used unchanged in a large variety of contexts.

Keywords: Performance debugging; Behavioral visualization; Interactivity; Scalability; Genericity; Pajé

1. Introduction

In addition to the traditional debugging activity aimed at correctness, parallel and distributed applications require some performance debugging. Performance debugging is necessary since parallel applications should perform as efficient as possible.

Performance debugging requires a large number of performance indices to be measured. The objective of performance debugging is to reduce the completion time of a certain application but the corresponding net-index can be decomposed into several sub-measures due to fractional time spent in various parts of the application, such as procedures, communication protocols, etc. In order to reduce completion times, programmers also need to know what percentage of the CPU-time goes into synchronization code, communication or idling time, and all the mentioned issues may be addressed from performance debugging.

1.1. Performance monitoring

Performance data are collected by monitoring tools which are mainly clock driven or event driven [4]. Clock driven monitoring amounts to having the state of each observed process registered at periodic time intervals by a second process which is independent of the observed process. Tools such as *gprof* [7] belong to this category. Clock driven tools are mainly used to compute resource utilization rates. However, this sort of tool may fail in finding the causes of overheads in parallel and distributed programs: global performance indices are of little help to discover bottlenecks or to evaluate communication- or idling time. Such

☆ This work was partially sponsored by grants from CAPES-COFECUB and CNPq-INRIA while both authors were members of the Apache research project of the ID-IMAG laboratory (ID-IMAG, http://www-id.imag.fr).
* Corresponding author. Present address: Lab. Informatique et Distribution, ENSIMAG, Antenne de Montbonnot ZIRST, 51 Avenue Jean Kuntzmann, Montbonnot-Saint-Martin 38330, France.
E-mail addresses: jacques.chassin-de-kergommeaux@imag.fr (J. Chassin de Kergommeaux), benhur@inf.ufsm.br (B. de Oliveira Stein).
URL: http://www-lsr.imag.fr/

0167-739X/02/$ – see front matter
PII: S0167-739X(02)00181-4

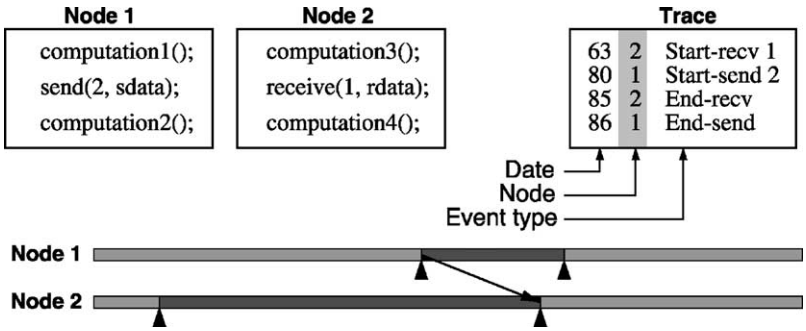

Fig. 1. Visualization from an execution trace. Events Start-send and End-send, are recorded when the send function is executed on node 1 and events Start-receive and End-receive when the receive function is executed on node 2.

information can be better obtained by event driven monitoring approaches which include counting, timing and tracing.

The number of occurrences of certain events, e.g. number of messages sent or received are computed and recorded when counting is used. The time spent in various parts of the programs, bounded by observed events, is measured when timing is used. Tracing is done by recording each of the observed events into a performance trace. Each record includes the type of recorded event as well as its time-stamp and several optional parameters depending on which type of event has occurred; for example a *send message* event is likely to include the destination process (or thread) and the size of the message. Tracing is the most general event driven monitoring technique since it allows the computation of all the performance indices that can be obtained by counting or timing. In addition, performance traces can be used to reconstruct the behavior of parallel applications, which is often necessary to help users to identify performance malfunction (see Fig. 1). For more information on performance monitoring please, see Ref. [4].

1.2. Performance visualization

Monitoring of parallel and distributed applications produces a large amount of data which can be used to compute detailed performance indices. Visualization is the technique most frequently used to present these indices in a comprehensible way. In order to be helpful high-quality visualization needs to meet certain criteria [9]. Among the most important are: scalability, interactivity and extensibility. Scalability is the ability to cope with large systems and long running applications. This becomes important with the growing popularity of large-sized clusters of several hundreds of interconnected off-the-shelf computers. Perfectly scalable views are independent of the duration of the monitored application or of the size of the observed system and are therefore limited to statistical information. Interactivity enables users to decide during the visualization what they want to see next: “inspect” an object to obtain more details, change the level of abstraction or the type of visualized data, etc. It is important since it is not possible to represent within a single display all the information of potential interest for performance debugging. Extensibility gives the possibility to extend the visualization tool with new functionalities: processing of new types of performance data, adding new graphical displays, visualizing new programming models, etc. Extensibility is important to cope with the evolution of visualization techniques and parallel (distributed) programming models.

1.3. Overview of Pajé

The Pajé[1] tool, described in the remainder of this article, aims at visualizing the dynamics of large-scale applications while providing characteristics such as scalability, interactivity and extensibility. Such a combination is very hard to achieve since behavioral visualization tools are inherently non-scalable while

[1] Pajé is a word used by Tupi Indians to name a doctor or a sorcerer.

interactivity often contradicts scalability; in addition, behavioral visualization tools include a simulation module which most often depends on the programming model used by the observed programs and therefore hinders the extensibility of the tool.

Scalability is mainly provided by zooming and filtering functionalities. Interactivity appears in most functionalities: inspection of displayed objects, highlighting of the objects related to the object pointed to by the mouse, moving back and forth in time, zooming and filtering. Several characteristics of Pajé were designed to provide a high degree of extensibility: modular architecture, genericity of the modules, including the visualization and simulation ones. The genericity property has so far allowed the use of Pajé in various contexts such as to visualize the execution of applications using a thread-based hierarchical programming model, the execution of Java distributed applications or the visualization of system data on a large-sized cluster.

The next section gives surveys briefly existing visualization tools. The main functionalities of Pajé are then described and exemplified in the following sections. The flexibility provided by the generic characteristic of Pajé is then illustrated by its use for performance debugging of distributed Java applications and system administration of large-sized clusters before a quick overview of ongoing and future developments is given.

2. Survey of existing visualization tools

Several monitoring and visualization tools were developed to support performance debugging of parallel applications. ParaGraph [8] produced a large number of visualizations from traces collected by the instrumented communication library PICL or by other monitoring tools producing traces in PICL format. Although having been the most widespread parallel visualization tool for years, ParaGraph suffered from the absence of interactivity, the lack of scalability and the absence of extensibility due to a monolithic implementation.

In the Pablo monitoring environment [13], a graphical tool helps users to insert tracing instructions in their applications. To be easily extensible, Pablo is implemented as a data-flow graph of software components, which can be connected using a graphical tool to produce a specific visualization tool. There exist a large number of components to transform data—arithmetic operations, statistical reductions, etc.—or to perform various visualizations. Scalability is provided by switching from tracing to counting when the amount of monitored data becomes too important and by providing only synthetic visualizations. Pablo is not interactive and does not provide any behavioral representation of parallel executions. Because users are requested to build a graph of components to be able to use it, Pablo is sometimes considered as difficult to use.

In SvPablo [6], monitoring is performed by counting and timing techniques and is therefore scalable. Scalability is also enforced by providing only visualization of statistical data. The tool is interactive and performance data are correlated with the source code of monitored applications, allowing users to obtain measurement data of, for example the time spent on executing a given function call in an application. SvPablo is independent of the programming language and of the target architecture of applications. Because event traces are not recorded, it is not possible to visualize the behavior of the monitored applications.

Vampir [11] is a widespread commercially available toolkit[2] for performance visualization of MPI programs. To be scalable, Vampir provides several filtering mechanisms, during monitoring (if VampirTrace is used to trace the application) or during visualization. In addition, zooming functionalities are available along the time-axis of space–time visualizations. Vampir is fairly interactive by allowing users to check the source code having generated a given event (source code click-back facility) and to open and close new displays interactively. Being a commercial tool, it is hard to know if it can be extended easily or not.

The objectives and functionalities of Virtue [14] go far beyond those of the above mentioned tools. Virtue aims at helping on the optimization of large-scale, geographically far distributed applications. In addition to post-mortem analysis and optimization, Virtue provides on-line performance measurement and steering. Virtue also aims at exploiting human sensory capabilities. Data are presented and user interactions are performed using virtual reality techniques: immersive

[2] From Pallas GmbH.

displays, tracking systems and spatialized audio. The counterpart of these outstanding functionalities is that they require a virtual reality studio to be used.

The main objective of Paradyn [10] is to search for performance problems in long running programs executed on a large number of Processing Elements. To prevent users to be overloaded with performance indices, the analysis is done automatically via a "Performance Consultant" inserting instrumentation dynamically where and whenever needed. A graphical interface with fixed-size (scalable) metric visualization is presented to the user.

The initial objective of Pajé was to help performance debugging of parallel applications running on large-sized clusters. Helping means that it is done by application programmers and not automatically. The assumption behind this design choice is that, until fully automatic tools become available, user driven performance debugging tools will most likely remain useful. To help application programmers to discover performance problems in their applications, Pajé provides behavioral representations of the dynamics of the executions of these applications. This choice implies to use tracing as a monitoring technique and to visualize the evolution of performance data along the time-axis. These techniques are known to scale poorly because the size of the monitoring and visualization data increases with the size of the application and complexity of the platform. In contrast to the usual limitations of scalability in tracing, Pajé visualizations provide a high degree of scalability because of interactive grouping and selection functionality. Pajé was also designed to be easily extensible by tool developers while being usable in a large variety of contexts. Another design goal of Pajé was to be functional on "standard" PC platforms and therefore to use "classical" interaction techniques.

In the following sections, the properties of Pajé will be exemplified by visualizing the execution of parallel applications using a thread-based parallel programming model—which can be considered as a thread aware implementation of the standard communication library MPI [2]. In this model, several threads execute on each of the nodes, communicate by shared-memory inside a node and by message passing between nodes. Inner parallelism between the processors of shared-memory symmetric multiprocessor nodes, used in many cluster architectures, can thus be exploited efficiently while multiprogramming of nodes allows the overlapping of communications by computation. To provide a clear representation of the behavior of parallel applications, a time line representation of the activities and communications of threads is mainly used, combining the information contained in the classical "space–time" charts with "Gantt chart" visualizations.

3. Extensibility

Extensibility is a key property of a visualization tool since it is a very complex piece of software, which is expensive to implement and should therefore have a lifetime as long as possible. This will be possible only if the tool can cope with the rapid evolution of parallel programming models and visualization techniques. The modular and generic characters of Pajé were therefore major design goals.

3.1. Modular architecture

To favor extensibility, the architecture of Pajé has been designed as a data-flow graph of software modules or components communicating through well-defined interfaces (see Fig. 2). According to this scheme, it is therefore possible to add a new visualization component or adapt to a change in the trace format by changing the trace reader component without having to change the rest of the environment. This architectural choice was inspired by Pablo [13] although there are important differences between both tools. The behavioral visualizations of Pajé require a simulation component, not present in Pablo which is also not interactive. To implement interactivity, the graph of components of Pajé includes also control-flow information, generated by the visualization modules to process user interactions and trigger the flow of data in the graph (see [3] for more details).

3.2. Genericity

Visualization tools providing behavioral visualizations, such as Pajé, are usually specialized for a given parallel programming model. The reason is that they include a trace driven simulator which usually depends on the semantics of the programming model of the

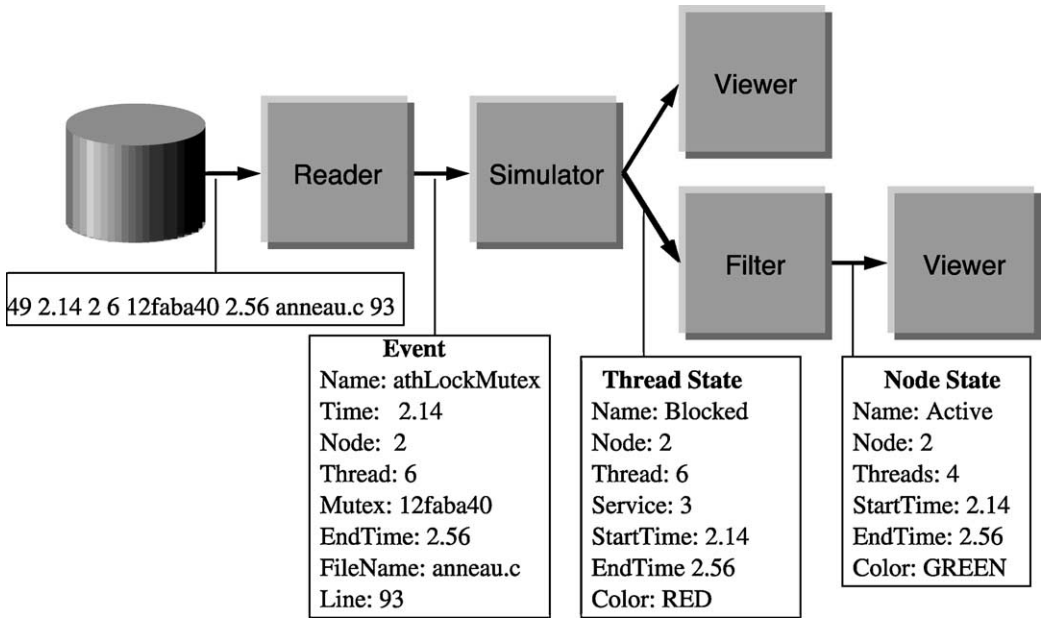

Fig. 2. Example data-flow graph. The trace reader produces event objects from the data read from the disk. These events are used by the simulator to produce more abstract objects, like thread states, communications, etc. The filter component groups (see Section 4) the thread states to build a node state indicating whether some threads of the node are active or not: the event processed in the figure reduces the number of active threads during the period where thread 6 remains blocked in the semaphore.

traced programs. This explains why tools such as Pablo [13], focusing on extensibility, do not provide behavioral representations: they do not include a simulator and their modules perform mere data transformations.

Similar to Pablo, the modular structure of Pajé makes it "easy" for tool developers to add a new component or extend an existing one. However, since Pajé includes a trace driven simulator, it was considered that developing a new simulation component to visualize a new programming model would require considerable programming efforts. This explain why Pajé was designed to allow all of its components, including the simulation and the visualization components, to be generic. The genericity of Pajé allows application programmers to define, in hierarchical data type trees, *what* they would like to visualize and *how* the visualized objects should be represented. The terminology used in the Pajé input data definition [5] is the following: the leaves and the nodes of the type trees are called *entities* and *containers*. Entities are elementary objects types. Containers are higher level object types, composed of lower level nodes and/or leaves. In the example shown in Fig. 3, entities can be events, thread states or communications; all events occurring in thread 1 of node 0 belong to the instance "thread 1 of node 0" of the container type "thread" of the type tree.

In the current implementation of Pajé, type hierarchies are defined in the trace files. These files contain four categories of data. A meta-format is used to describe the generic instructions and their formats as well as the trace events and their formats. The type tree is defined using the previously defined generic instructions. The last category of data is the set of events composing the trace itself. For a tutorial example on the Pajé input data format, see Fig. 4 and for more details, see Ref. [5].

Four kinds of data traverse the graphs of components: type hierarchies, instance hierarchies, entities and data characterizing entities such as colors, names, etc. When needed, these data are requested by visualization components from the preceding components in the data-flow graph. This is the case when a visualization has to be recomputed, due to some user interaction

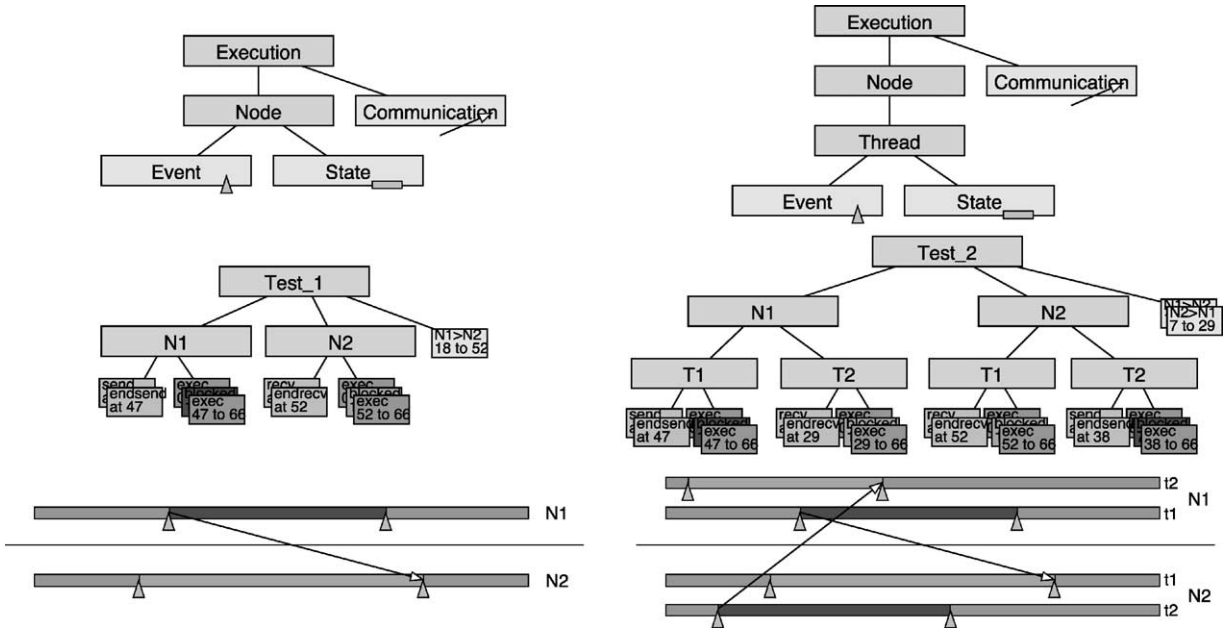

Fig. 3. Use of a simple type hierarchy. Both sides of the figure represent a type hierarchy, an instance of this type and how this instance can be displayed by a visualization component. The type tree also specifies how the entities are to be represented: here, communications as arrows, thread events as triangles and thread states as rectangles.

(see Section 5). These data traverse all filter components inserted in the path from data source to visualization modules. Any of these data can be altered, created or hidden by filter components. Traces collected by an existing tracer, such as an execution of an MPI program traced by VampirTrace [11], can be visualized using Pajé. An appropriate set of definitions, adapted to the programming model of the traced program and defining how executions are to be visualized, needs to be provided.

4. Scalability

It is not possible to represent simultaneously all the information that can be deduced from execution traces. Screen space limitation is not the only reason: part of the information may not be needed all the time or cannot be represented in a graphical way or can have several graphical representations. Giving users a simplified view of the data and intuitive ways to access more details from what appears to be the cause of problems, seems to be a good way of helping them to find out what these problems are. Accessing more detailed information at any later stage can always lead to a more detailed view of a certain situation. As indicated in Section 3.2, filtering activity can happen for any kind of data traversing the component graph. Pajé offers several filtering functionalities to help programmers in controlling this large amount of information. Filters in Pajé can be of several types:

- *Selection*. Permits the removal of entities from a visualization. This selection can be based on the type of visual objects (thread states, communications, etc.), on their values (of all possible thread states, show only the running thread states) or other user-intended actions (select which nodes, threads, semaphores, mutexes or groups are deemed to be more informative, see Fig. 5).
- *Reduction*. Provides more abstract representations of information, for the production of synthetic views. Fig. 5(b) and (c) shows the same execution as (a), with only one line for nodes 1 and 2, whose states represent the number of active threads on

1-*/* Description and format of a generic instruction defining container types */*

PajeDefineContainerType 1,*/* Instruction code of the generic instruction */*
NewType **string**, */* Type name of the defined container */*
ContainerType **string**, */* Parent container type in the type hierarchy */*
NewName **string**,*/* Name of the new type, as it is known by the user of Pajé */* **endef**

2- */*Description and format of a generic instruction defining one type of execution events: the creation of a new container.*/*

PajeCreateContainer 7, */* instruction code */* Time **date**, */* event date */*
NewContainer **string**, */* container identifier */*
NewContainerType **string**, */* type of the new container, already defined by a PajeDefineContainerType */*
Container **string**, NewName **string**, **endef**

3- */* Definition of a small hierarchical type tree, using the previously defined generic container type definition instruction */*

```
1 P 0 Program /* 0:  no parent in the type hierarchy */
1 N P Node
1 T N Thread
```

4-*/* Event records complying with the previously defined formats and type tree. */*

```
7 0.000 TP P 0 ‘‘Test Prog’’ /* no parent in instance hierarchy */
7 0.000 N1 N TP ‘‘Node 1’’ /* Node 1 created at initialization */
7 0.125 T1 T N1 ‘‘Thread 1’’ /* Creation of thread 1 */
7 0.154 T2 T N1 ‘‘Thread 2’’ /* Creation of thread 2 */
```

Fig. 4. Example of Pajé input data format. Comments are written in italics while the keywords of the meta-language are written in bold font. This example does not illustrate which type of representation has been chosen for entities, which is done by selecting one of the elementary graphical objects provided by Pajé: square, arrow, etc.

these nodes. The CPU activity of one or several nodes can also be summarized in a pie chart (see Fig. 6). Reduction is performed by filter components (see Fig. 2); visual objects can be grouped using several reduction operators computing the average, maximum or minimum of some object type parameter or a counter of the number of grouped entities; these operators can be selected by Pajé users when defining the parameters of a filter component.

- *Grouping*. Nodes, threads, semaphores, mutexes can be grouped. An object belonging to any member of a group is shown as belonging to this group (see Fig. 5). Grouping does not reduce the number of objects being visualized.
- *Repositioning*. Allows users to choose the order in which the objects are shown, so that, for example, related nodes or threads can be displayed closer together.

Being able to switch *interactively* from detailed to grouped visualizations gives programmers zooming capabilities within a node or between several nodes. The execution of an application running on a large-sized cluster can be observed at the group of processors level of abstraction, until a problem is noticed in one of the groups: it is then possible to focus on this group and, if necessary, on one of its nodes until the origin of the problem is found.

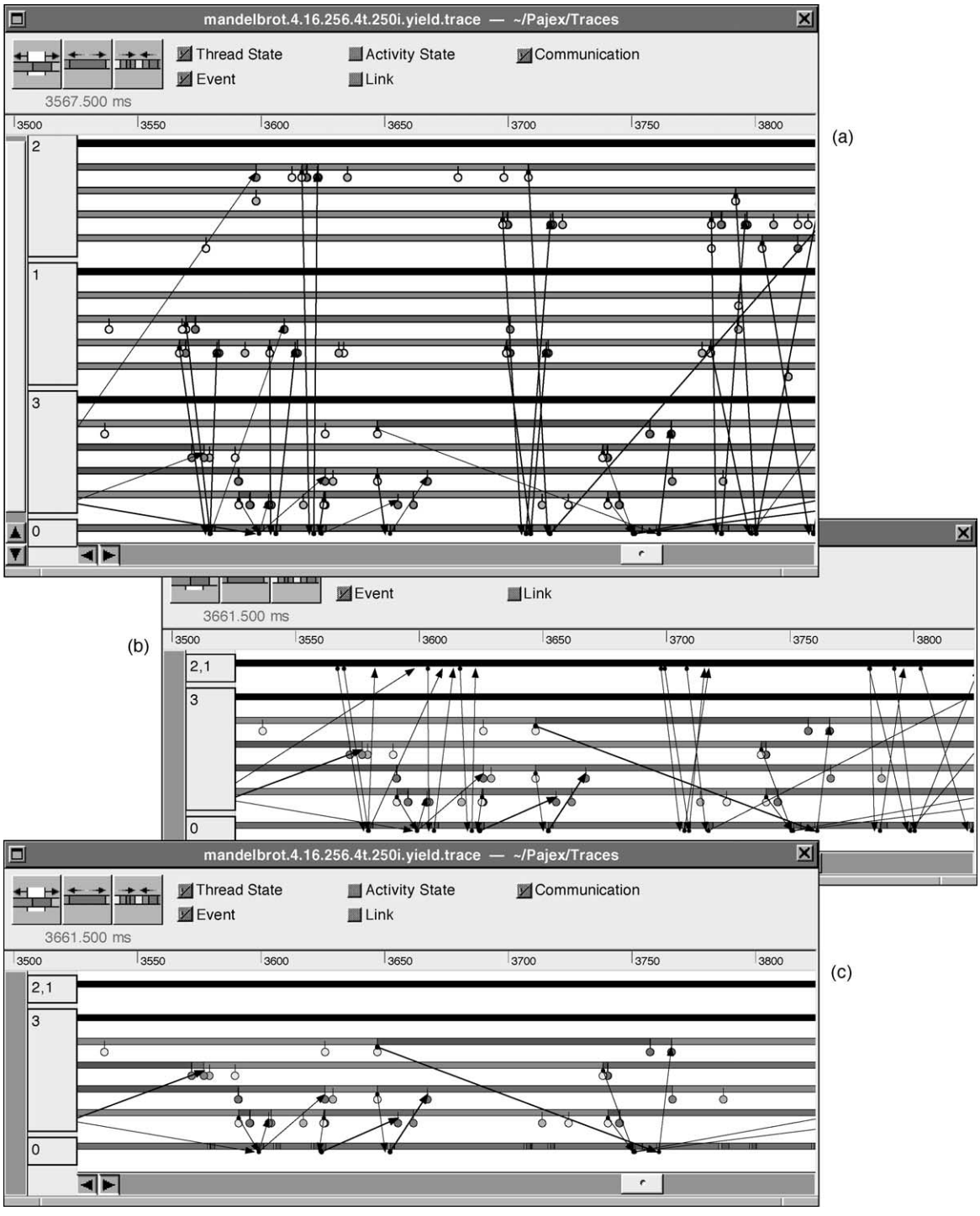

Fig. 5. Use of selection and grouping filters. Blocked thread states are represented in clear color; runnable states in dark color. In (a), the events of node 0 are filtered. In (b), nodes 1 and 2 are grouped and the events of this group are filtered. In (c), the communications of this group are also filtered. Filters are activated by the user who selects, with simple mouse clicks, the entities (thread state, activity, state, etc.) that should appear on the screen (see selection of entity type buttons in Fig. 6).

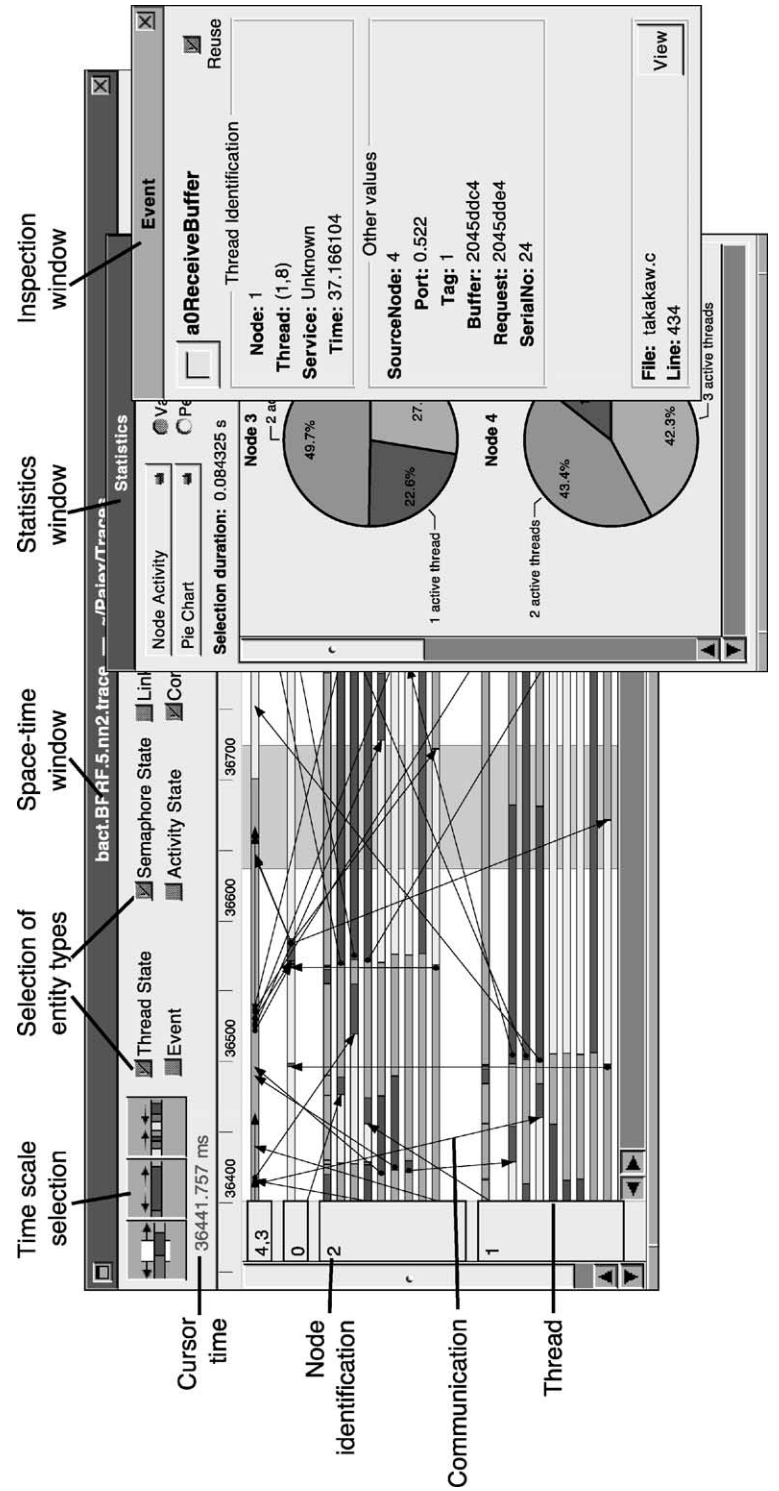

Fig. 6. Inspection of an event.

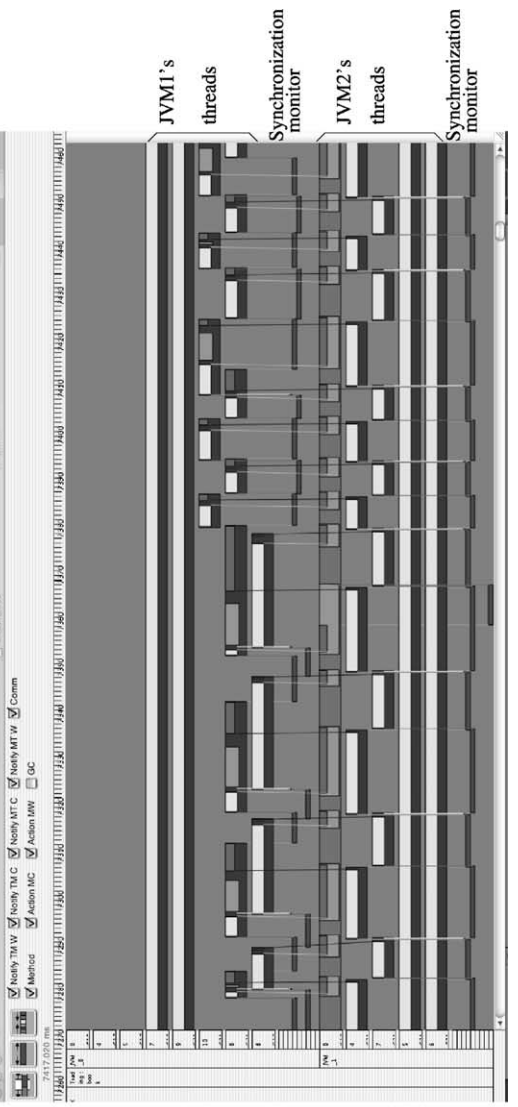

Fig. 7. Visualization of Java distributed applications. The type hierarchy used is represented by the tree on the left. The screen shot of the right represents the execution of two JVMS where threads synchronize using a monitor within a JVM and communicate by message passing between JVMs.

5. Interactivity

As indicated above, the use of all filtering functionalities of Pajé is interactive. In addition, moving forward and backward in time is entirely driven by user-controlled time displacements. All the information available for a displayed object can be shown in an inspection window, created by clicking over the representation of the object (see Fig. 6). The name of the visual object under the pointer is also displayed on top of the window, together with the time corresponding to the pointer position.

Pajé keeps a relation between visual objects and source code: from the visual representation of an event, it is possible to display the corresponding source code line of the parallel application being visualized (click-back). Likewise, selecting a line in the source code browser highlights the events that have been generated by this line (click-forward).

Interactivity implies to keep all the elementary objects in memory—thread or semaphore states, communications, events, etc. Such a constraint hinders scalability. As a compromise for long running applications, only the data allowing the visualization of part of an execution (in time) are kept in memory. These data are organized in an "observation window" which slides forward in time by including new objects derived from reading the trace and "forgetting" old ones, when a user attempts to visualize events occurring outside the time-frame of the current observation window [3]. The state of the simulation is recorded at regular intervals, so that later on it is always possible to move back in time. In such cases, the simulation is restarted at the closest saved state before the date of interest.

Fig. 8. Scheduling of jobs on a large-sized cluster. Each line represents a different processor of a 100 processors cluster. The three columns on the left identify interconnection switches, nodes and processors (there is a single processor per node in the example). Each color represents a different user reservation.

6. Performance debugging of distributed Java applications

Pajé was used unchanged to visualize the execution of distributed applications by several Java virtual machines (JVM) cooperating via object-interaction [12]. Performance traces are recorded at the application as well as at the operating system level of abstraction. The events recorded at the application level represent JVM activities such as method calls. The operating system level record is due to communications between different JVMs. Execution traces are then passed to Pajé together with a description of the type hierarchy to be used for the visualization (see Fig. 7). This technique allows the observation and visualization of the dynamic behavior of distributed Java applications, including communication, without any modification of the application program nor of the JVM. Moreover, it allows a temporal analysis of the hierarchy of method invocations, represented as nested boxes.

7. System administration of large-sized clusters

Pajé could also be used unchanged to help system administration of large-sized clusters, built from off-the-shelf personal computers and standard interconnection networks [1]. System parameters such as CPU load, number of active processes, memory usage and I/O's are available on each of the nodes and should be made available to the system administrators. The filtering and grouping functionalities of Pajé can help combining global and detailed observation of system parameters. The behavior of the system can be analyzed post-mortem over a certain time period: it is possible to concentrate on fractional periods of interest. It is also possible to restrict the visualization to a subset of problematic nodes with too high or too low CPU load, for example. System parameters are collected on each node at regular time intervals and used to build a trace file. In the beginning of this trace file a description of the type hierarchy associated to the visualization is inserted (Fig. 8).

8. Conclusion and future developments

Pajé makes available the most important characteristics of behavioral visualization tools for a large variety of application domains related to parallel or distributed computing. This is enabled by its genericity, allowing users to describe what they wish to visualize and how this should be done. Recent developments are mainly concerned with the development of new filtering capabilities, on-line visualization and evolution of the input format. New filters and new functionalities, allowing users to define their own filtering capabilities, are developed. Another functionality, allowing data to be read on-line is currently being tested. This will allow data generated at a slow rate—such as system parameters used for cluster administration—to be visualized in real time. Another work in progress is to allow the format and the hierarchical type tree definitions to be separated from the actual trace file containing execution events. At the same time, XML will be used to express these definitions. Because of the modular structure of Pajé, these extensions could be done rapidly.

The limits of Pajé were not reached so far. However, the large amount of data corresponding to tracing the execution of parallel or distributed applications running on large-sized platforms does certainly pose a limit. Several solutions to this problem are under consideration. One solution would be to parallelize the execution of Pajé. Trace information would be kept in separate files and processed in parallel upon request from the visualization components running on several distributed workstations. The modular structure of Pajé, with independent components communicating via well-defined protocols, should ease such a parallelization.

Acknowledgements

François Ottogali used Pajé for performance debugging of Java applications. Cyril Guilloud and Philippe Augerat used it for system administration of a large-sized cluster. Pajé was developed using the OpenStep operating system and then ported to MacOS X and GNUStep on top of Linux.

References

[1] P. Augerat, C. Martin, B. de Oliveira Stein, Scalable monitoring tools for grids and clusters, in: Proceedings of

the 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing, IEEE Computer Society Press, Silver Spring, MD, 2002, pp. 147–153.
[2] J. Briat, I. Ginzburg, M. Pasin, B. Plateau, Athapascan runtime: efficiency for irregular problems, in: Proceedings of the EuroPar'97, Lecture Notes in Computer Science, vol. 1300, Springer, Berlin, August 1997, pp. 591–600.
[3] J. Chassin de Kergommeaux, B. de Oliveira Stein, P. Bernard, Pajé, an interactive visualization tool for tuning multi-threaded parallel applications, Parallel Comput. 26 (10) (2000) 1253–1274.
[4] J. Chassin de Kergommeaux, E. Maillet, J.-M. Vincent, Monitoring parallel programs for performance tuning in cluster environments, in: J. Cunha, et al. (Eds.), Parallel Program Development for Cluster Computing: Methodology, Tools and Integrated Environments, vol. 5, Nova Science, 2001, Chapter 6, pp. 131–150.
[5] B. de Oliveira Stein, J. Chassin de Kergommeaux, G. Mounié, Pajé trace file format, Technical Report No. ID-IMAG, Grenoble, France, 2002. http://www-id.imag.fr/Logiciels/paje/publications.
[6] L. DeRose, D. Reed, SvPablo: a multi-language architecture-independent performance analysis system, in: Proceedings of the International Conference on Parallel Processing (ICPP'99), Fukushima, Japan, September 1999, pp. 311–318.
[7] S. Graham, P. Kessler, M. McKusik, gprof: a call graph execution profiler, in: Proceedings of the SIGPLAN'82 Symposium on Compiler Construction, ACM, New York, 1982, pp. 120–126.
[8] M.T. Heath, J.A. Etheridge, Visualizing the performances of parallel programs, IEEE Trans. Softw. Eng. 8 (5) (1991) 29–39.
[9] B. Miller, What to draw? When to draw? An essay on parallel program visualization, J. Parallel Distrib. Comput. 18 (1993) 265–269.
[10] B. Miller, et al., The paradyn parallel performance measurement tool, Computer 28 (11) (November 1995) 37–46. More recent publications at: http://www.paradyn.org.
[11] W. Nagel, et al., VAMPIR: visualization and analysis of MPI resources, Supercomputer 12 (1) (1996) 69–80.
[12] F.-G. Ottogalli, V. Olive, B. de Oliveira Stein, J. Chassin de Kergommeaux, J.-M. Vincent, Visualisation of distributed applications for performance debugging, in: V. Alexandrov, et al. (Eds.), ICCS'01, Lecture Notes in Computer Science, vol. 2074, Springer, Berlin, 2001, pp. 831–840.
[13] D.A. Reed, R.A. Aydt, Madhyastha, et al., An overview of the Pablo performance analysis environment, Technical Report, Department
of Computer Science, University of Illinois, Urbana, IL, 1992. This report and recent publications at: http://www-pablo.cs.uiuc.edu/Publications/Documents/documents.htm.
[14] E. Shaffer, D. Reed, S. Whitmore, Virtue: performance visualization of parallel and distributed applications, IEEE Comput. 32 (12) (1999) 44–51.

J. Chassin de Kergommeaux has been a Professor in Computer Science at the Institut National Polytechnique de Grenoble, France (INPG) since 2001. Prior he was a CNRS (Centre National de la Recherche Scientifique) researcher working at the IMAG institute since 1991. His current interests include correctness and performance debugging and testing of parallel and distributed programs. He obtained a Master's degree in computer science from the Ecole National d'Informatique et de Mathématiques Appliquées de Grenoble (ENSIMAG), France, in 1974, a PhD in computer science from the Université Joseph Fourrier de Grenoble, in 1989 and an Habilitation Thesis from the Institut National Polytechnique de Grenoble (INPG), in 2000.

B. de Oliveira Stein has been teaching computer science at the University of Santa Maria (Rio Grande do Sul, Brazil) since 1991 where he is now a Professor. Prior, he was an Electrical Engineer from 1983 to 1988. His current interest include performance debugging and visualization of parallel and distributed programs, cluster computing and distributed systems. He obtained a Master's degree in computer science from the Universidade Federale do Rio Grande do Sul at Porto Alegre, Brazil, in 1992 and a PhD in computer science from the Université Joseph Fourier at Grenoble, France, in 1999.