Pajé: An Extensible Environment for Visualizing Multi-threaded Programs Executions

Jacques Chassin de Kergommeaux[1] and Benhur de Oliveira Stein[2]

[1] ID-IMAG, ENSIMAG - antenne de Montbonnot, ZIRST, 51, avenue Jean Kuntzmann, 38330 MONTBONNOT SAINT MARTIN, France.
`Jacques.Chassin-de-Kergommeaux@imag.fr`
`http://www-apache.imag.fr/~chassin`
[2] Departamento de Eletrônica e Computação, Universidade Federal de Santa Maria, Brazil.
`benhur@inf.UFSM.br`
`http://www.inf.ufsm.br/~benhur`

Abstract. Pajé is an interactive visualization tool for displaying the execution of parallel applications where a (potentially) large number of communicating threads of various life-times execute on each node of a distributed memory parallel system. The main novelty of Pajé is an original combination of three of the most desirable properties of visualization tools for parallel programs: extensibility, interactivity and scalability. This article mainly focuses on the extensibility property of Pajé, ability to easily add new functionalities to the tool. Pajé was designed as a data-flow graph of modular components to ease the replacement of existing modules or the implementation of new ones. In addition the genericity of Pajé allows application programmers to tailor the visualization to their needs, by simply adding tracing orders to the programs being traced.

Keywords: performance debugging, visualization, MPI, pthread, parallel programming.

1 Introduction

The Pajé visualization tool was designed to allow programmers to visualize the executions of parallel programs using a potentially large number of communicating threads (lightweight processes) evolving dynamically. The visualization of the executions is an essential tool to help tuning applications using such a parallel programming model.

Visualizing a large number of threads raises a number of problems such as coping with the lack of space available on the screen to visualize them and understanding such a complex display. The graphical displays of most existing visualization tools for parallel programs [8, 9, 10, 11, 14, 15, 16] show the activity of a fixed number of nodes and inter-nodes communications; it is only possible to represent the activity of a single thread of control on each of the nodes. Some tools were designed to display multi-threaded programs [7, 17]. However, they support a programming model involving a single level of parallelism within a node, this node being in general a shared-memory multiprocessor. Our programs execute on several nodes: within the same node, threads communicate using synchronization primitives; however, threads executing on different nodes communicate by message passing.

The most innovative feature of Pajé is to combine the characteristics of interactivity and scalability with extensibility. In contrast with passive visualization tools [8, 14] where parallel program entities — communications, changes in processor states, etc. — are displayed as soon as produced and cannot be interrogated, it is possible to inspect all the objects displayed in the current screen and to move back in time, displaying past objects again. Scalability is the ability to cope with a large number of threads. Extensibility is an important characteristic of visualization tools to cope with the evolution of parallel programming interfaces and visualization techniques. Extensibility gives the possibility to extend the environment with new functionalities: processing of new types of traces, adding new graphical displays, visualizing new programming models, etc.

The interactivity and scalability characteristics of Pajé were described elsewhere [2, 4]. This article focuses on the extensibility characteristics: modular design easing the addition of new modules, semantics independent modules which allow them to be used in a large variety of contexts and especially genericity of the simulator component of Pajé which gives to application programmers the ability to define what they want to visualize and how it must be done.

The main functionalities of Pajé are summarized in the next section. The following section describes the extensibility of Pajé before the conclusion.

2 Outline of Pajé

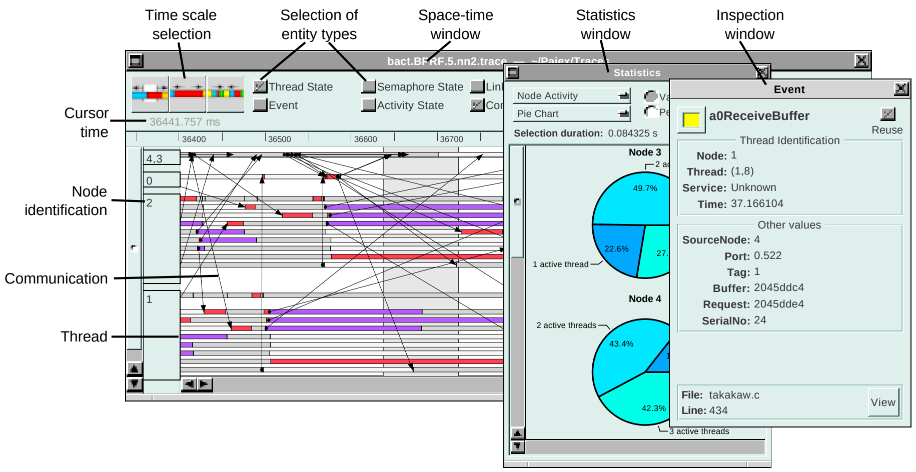
Pajé was originally designed to ease performance debugging of ATHAPASCAN programs by visualizing their executions and because no existing visualization tool could be used to visualize such multi-threaded programs.

2.1 ATHAPASCAN: A Thread-Based Parallel Programming Model

Combining threads and communications is increasingly used to program irregular applications, mask communications or I/O latencies, avoid communication deadlocks, exploit shared-memory parallelism and implement remote memory accesses [5, 6]. The ATHAPASCAN [1] programming model was designed for parallel hardware systems composed of shared-memory multi-processor nodes connected by a communication network. Inter-nodes parallelism is exploited by a fixed number of system-level processes while inner parallelism, within each of the nodes, is implemented by a network of communicating threads evolving dynamically. The main functionalities of ATHAPASCAN are dynamic local or remote thread creation and termination, sharing of memory space between the threads of the same node which can synchronize using locks or semaphores, and blocking or non-blocking message-passing communications between non local threads, using ports. Combining the main functionalities of MPI [13] with those of `pthread` compliant libraries, ATHAPASCAN can be seen as a “thread aware” implementation of MPI.

2.2 Tracing of Parallel Programs

Execution traces are collected during an execution of the observed application, using an instrumented version of the ATHAPASCAN library. A non-intrusive, statistical method



Selection duration: 0.084325 s

a0ReceiveBuffer

Thread Identification
Node: 1
Thread: (1,8)
Service: Unknown
Time: 37.166104

Other values
SourceNode: 4
Port: 0.522
Tag: 1
Buffer: 2045ddc4
Request: 2045dde4
SerialNo: 24

File: takakaw.c
Line: 434

Fig. 1. Visualization of an ATHAPASCAN program execution
Blocked thread states are represented in clear color; runnable states in a dark color. The smaller window shows the inspection of an event.

is used to estimate a precise global time reference [12]. The events are stored in local event buffers, which are flushed when full to local event files. Recorded events may contain source code information in order to implement source code click-back — from visualization to source code — and click-forward — from source code to visualization — in Pajé.

2.3 Visualization of Threads in Pajé

The visualization of the activity of multi-threaded nodes is mainly performed in a diagram combining in a single representation the states and communications of each thread (see figure 1) . The horizontal axis represents time while threads are displayed along the vertical axis, grouped by node. The space allocated to each node of the parallel system is dynamically adjusted to the number of threads being executed on this node. Communications are represented by arrows while the states of threads are displayed by rectangles. Colors are used to indicate either the type of a communication, or the activity of a thread.

The states of semaphores and locks are represented like the states of threads: each possible state is associated with a color, and a rectangle of this color is shown in a position corresponding to the period of time when the semaphore was in this state. Each lock is associated with a color, and a rectangle of this color is drawn close to the thread that holds it. Moving the mouse pointer over the representation of a blocked thread state highlights the corresponding semaphore state, allowing an immediate recognition. Similarly, all threads blocked in a semaphore are highlighted when the pointer is moved over the corresponding state of the semaphore.

In addition, Pajé offers many possible interactions to programmers: displayed objects can be inspected to obtain all the information available for them (see inspection window in figure 1), identify related objects or check the corresponding source code. Selecting a line in the source code browser highlights the events that have been generated by this line.

Progress of the simulation is entirely driven by user-controlled time displacements: at any time during a simulation, it is possible to move forward or backward in time. Memory usage is kept to acceptable levels by a mechanism of checkpointing the internal state of the simulator and re-simulating when needed.

It is not possible to represent simultaneously all the information that can be deduced from the execution traces. Pajé offers several filtering and zooming functionalities to help programmers cope with this large amount of information to give users a simplified, abstract view of the data. Figure 1 exemplifies one of the filtering facilities provided by Pajé where the top most line represents the number of active threads of a group of two nodes (nodes 3 and 4) and a pie graph the CPU activity in the time slice selected in the space-time diagram (see [2, 3] for more details).

3 Extensibility

Extensibility is a key property of a visualization tool. The main reason is that a visualization tool being a very complex piece of software, costly to implement, its lifetime ought to be as long as possible. This will be possible only if the tool can cope with the evolutions of parallel programming models and of the visualization techniques, since both domains are evolving rapidly. Several characteristics of Pajé were designed to provide a high degree of extensibility: modular architecture, flexibility of the visualization modules and genericity of the simulation module.

3.1 Modular Architecture

To favor extensibility, the architecture of Pajé is a data flow graph of software modules or components. It is therefore possible to add a new visualization component or adapt to a change of trace format by changing the trace reader component without changing the remaining of the environment. This architectural choice was inspired by Pablo [14], although the graph of Pajé is not purely data-flow for interactivity reasons: it also includes control-flow information, generated by the visualization modules to process user interactions and trigger the flow of data in the graph (see [2, 3] for more details).

3.2 Flexibility of Visualization Modules

The Pajé visualization components do not depend on specific parallel programming models. Prior to any visualization they receive as input the description of the types of the objects to be visualized as well as the relations between these objects and the way these objects ought to be visualized (see figure 2). The only constraints are the hierarchical nature of the type relations between the visualized objects and the ability to place each of these objects on the time-scale of the visualization. The hierarchical type description is used by the visualization components to query objects from the preceding components in the graph.

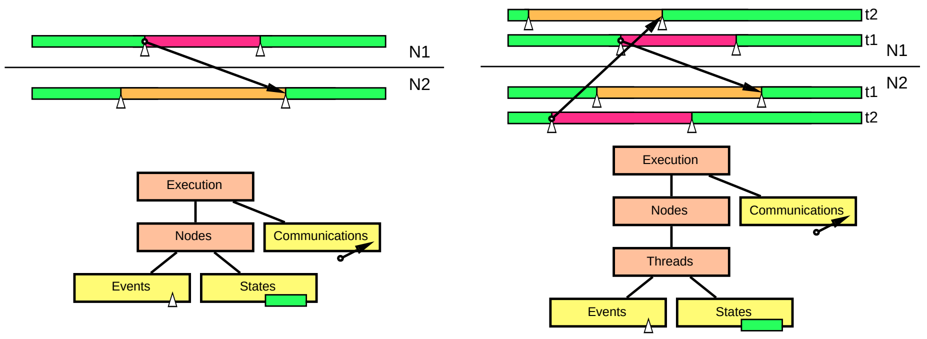


Fig. 2. Use of a simple type hierarchy
The type hierarchy on the left-hand side of the figure defines the type hierarchical relations between the objects to be visualized and how these objects should be represented: communications as arrows, thread events as triangles and thread states as rectangles. The right-hand side shows the changes necessary to the hierarchy in order to represent threads.

This type description can be changed to adapt to a new programming model (see section 3.3) or during a visualization, to change the visual representation of an object upon request from the user. This feature is also used by the filtering components when they are dynamically inserted in the data-flow graph of Pajé — for example between the simulation and visualization components to zoom from a detailed visualization and obtain a more global view of the program execution (see [2, 3] for more details).

The type hierarchies used in Pajé are trees whose leaves are called *entities* and intermediate nodes *containers*. Entities are elementary objects such as events, thread states or communications. Containers are higher level objects, including entities or lower level containers (see figure 2). For example: all events occurring in thread 1 of node 0 belong to the container “thread-1-of-node-0”.

3.3 Genericity of Pajé

The modular structure of Pajé as well as the fact that filter and visualization components are independent of any programming model makes it “easy” for tool developers to add a new component or extend an existing one. These characteristics alone would not be sufficient to use Pajé to visualize various programming models if the simulation component were dependent on the programming model: visualizing a new programming model would then require to develop a new simulation component, which is still an important programming effort, reserved to experienced tool developers.

On the contrary, the generic property of Pajé allows application programmers to define *what* they would like to visualize and *how* the visualized objects should be represented by Pajé. Instead of being computed by a simulation component, designed for a specific programming model such as ATHAPASCAN, the type hierarchy of the visualized objects (see section 3.2) can be defined by the application programmer, by inserting several definitions and commands *in the application program to be traced and visualized*. These definitions and commands are collected by the tracer (see section 2.2) so that they can be passed to the Pajé simulation component. The simulator uses these

Table 1. Containers and entities types definitions and creation

Type definition	Creation of entities
pajeDefineUserContainerType	pajeCreateUserContainer pajeDestroyUserContainer
pajeDefineUserEventType	pajeUserEvent
pajeDefineUserStateType	pajeSetUserState pajePushUserState pajePopUserState
pajeDefineUserLinkType	pajeStartUserLink pajeEndUserLink
pajeDefineUserVariableType	pajeSetUserVariable pajeAddUserVariable

definitions to build a new data type tree relating the objects to be displayed, this tree being passed to the following modules of the data flow graph: filters and visualization components.

New Data Types Definition. One function call is available to create new types of containers while four can be used to create new types of entities which can be events, states, links and variables. An "event" is an entity representing an instantaneous action. "States" of interest are those of containers. A "link" represents some form of connection between a source and a destination container. A "variable" stores the temporal evolution of the successive values of a data associated with a container. Table 1 contains the function calls that can be used to define new types of containers and entities. The right-hand part of figure 2 shows the effect of adding the "threads" container to the left-hand part.

Data Generation. Several functions can be used to create containers and entities whose types are defined using the primitives of the left column of table 1. Functions of the right column of table 1 are used to create events, states (and embedded states using *Push* and *Pop*), links — each link being created by one source and one destination calls — and change the values of variables.

In the example of figure 3, a new event is generated for each change of computation phase. This event is interpreted by the Pajé simulator component to generate the corresponding container state. For example the following call indicates that the computation is entering in a "Local computation" phase:

```
  pajeSetUserState ( phase_state, node, local_phase, "" );
```

The second parameter indicates the container of the state (the "node" whose computation has just been changed). The last parameter is a comment that can be visualized by Pajé. The example program of figure 3 includes the definitions and creations of entities "Computation phase", allowing the visual representation of an ATHAPASCAN program execution to be extended to represent the phases of the computation. Figure 4 shows a space-time diagram visualizing the execution of this example program with the definition of the new entities.

```
unsigned phase_state, init_phase, local_phase, global_phase;
phase_state  = pajeDefineUserStateType( A0_NODE, "Computation phase");
init_phase   = pajeNewUserEntityValue( phase_state, "Initialization");
local_phase  = pajeNewUserEntityValue( phase_state, "Local computation");
global_phase = pajeNewUserEntityValue( phase_state,"Global computation");

pajeSetUserState ( phase_state, node, init_phase, "" );
initialization();
while (!converge) {
    pajeSetUserState ( phase_state, node, local_phase, "" );
    local_computation();
    send (local_data);
    receive (remote_data);
    pajeSetUserState ( phase_state, node, global_phase, "" );
    global_computation();
}
```

Fig. 3. Simplified algorithm of the example program
Added tracing primitives are shown in bold face.

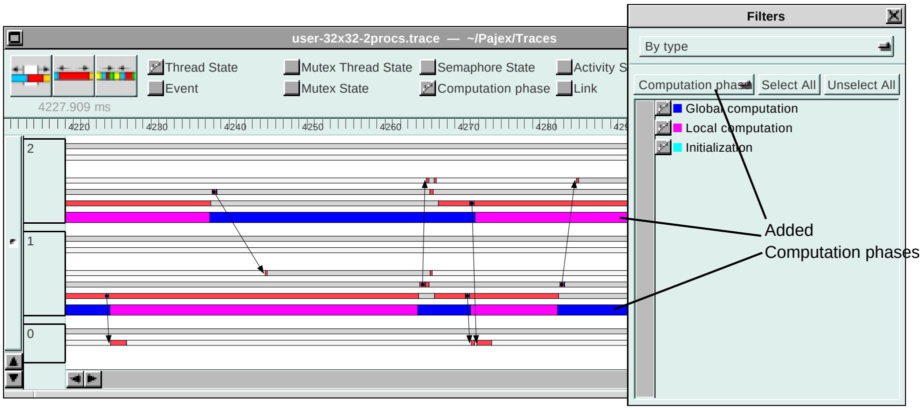


Fig. 4. Visualization of the example program

4 Conclusion

Pajé provides solutions to interactively visualize the execution of parallel applications using a varying number of threads communicating by shared memory within each node and by message passing between different nodes. The most original feature of the tool is its unique combination of extensibility, interactivity and scalability properties. Extensibility means that the tool was defined to allow tool developers to add new functionalities or extend existing ones without having to change the rest of the tool. In addition, it is possible to application programmers using the tool to define what they wish to visualize and how this should be represented. To our knowledge such a generic feature was not present in any previous visualization tool for parallel programs executions.

References

[1] J. Briat, I. Ginzburg, M. Pasin, and B. Plateau. Athapascan runtime: efficiency for irregular problems. In C. Lengauer et al., editors, *EURO-PAR'97 Parallel Processing*, volume 1300 of *LNCS*, pages 591–600. Springer, Aug. 1997.

[2] J. Chassin de Kergommeaux and B. d. O. Stein. Pajé, an extensible and interactive and scalable environment for visualizing parallel program executions. Rapport de Recherche RR-3919, INRIA Rhone-Alpes, april 2000. `http://www.inria.fr/RRRT/publications-fra.html`.

[3] B. de Oliveira Stein. *Visualisation interactive et extensible de programmes parallèles à base de processus légers*. PhD thesis, Université Joseph Fourier, Grenoble, 1999. In French. http://www-mediatheque.imag.fr.

[4] B. de Oliveira Stein and J. Chassin de Kergommeaux. Interactive visualisation environment of multi-threaded parallel programs. In *Parallel Computing: Fundamentals, Applications and New Directions*, pages 311–318. Elsevier, 1998.

[5] T. Fahringer, M. Haines, and P. Mehrotra. On the utility of threads for data parallel programming. In *Conf. proc. of the 9th Int. Conference on Supercomputing, Barcelona, Spain, 1995*, pages 51–59. ACM Press, New York, NY 10036, USA, 1995.

[6] I. Foster, C. Kesselman, and S. Tuecke. The nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, Aug. 1996.

[7] K. Hammond, H. Loidl, and A. Partridge. Visualising granularity in parallel programs: A graphical winnowing system for haskell. In A. P. W. Bohm and J. T. Feo, editors, *High Performance Functional Computing*, pages 208–221, Apr. 1995.

[8] M. T. Heath. Visualizing the performance of parallel programs. *IEEE Software*, 8(4):29–39, 1991.

[9] V. Herrarte and E. Lusk. Studying parallel program behavior with upshot, 1992. http://www.mcs.anl.gov/home/lusk/upshot/upshotman/upshot.html.

[10] D. Kranzlmueller, R. Koppler, S. Grabner, and C. Holzner. Parallel program visualization with MUCH. In L. Boeszoermenyi, editor, *Third International ACPC Conference*, volume 1127 of *Lecture Notes in Computer Science*, pages 148–160. Springer Verlag, Sept. 1996.

[11] W. Krotz-Vogel and H.-C. Hoppe. The PALLAS portable parallel programming environment. In *Sec. Int. Euro-Par Conference*, volume 1124 of *Lecture Notes in Computer Science*, pages 899–906, Lyon, France, 1996. Springer Verlag.

[12] É. Maillet and C. Tron. On Efficiently Implementing Global Time for Performance Evaluation on Multiprocessor Systems. *Journal of Parallel and Distributed Computing*, 28:84–93, July 1995.

[13] MPI Forum. MPI: a message-passing interface standard. Technical report, University of Tennessee, Knoxville, USA, 1995.

[14] D. A. Reed et al. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In A. Skjellum, editor, *Proceedings of the Scalable Parallel Libraries Conference*, pages 104–113. IEEE Computer Society, 1993.

[15] B. Topol, J. T. Stasko, and V. Sunderam. The dual timestamping methodology for visualizing distributed applications. Technical Report GIT-CC-95-21, Georgia Institute of Technology. College of Computing, May 1995.

[16] C. E. Wu and H. Franke. *UTE User's Guide for IBM SP Systems*, 1995. http://www.research.ibm.com/people/w/wu/uteug.ps.Z.

[17] Q. A. Zhao and J. T. Stasko. Visualizing the execution of threads-based parallel programs. Technical Report GIT-GVU-95-01, Georgia Institute of Technology, 1995.