
ELSEVIER

Parallel Computing 26 (2000) 1253–1274

PARALLEL COMPUTING

www.elsevier.com/locate/parco

Pajé, an interactive visualization tool for tuning multi-threaded parallel applications

J. Chassin de Kergommeaux [a,*], B. Stein [b], P.E. Bernard [c]

[a] *ID-IMAG, Project APACHE, B.P. 53, F-38041 Grenoble Cedex 9, France* [1]
[b] *Departamento de Eletrônica e Computação, Universidade Federal de Santa Maria, Brazil*
[c] *Numath, INRIA Lorraine, BP 101, F-54600 Villers les Nancy, France*

Received 5 March 1999; received in revised form 19 October 1999; accepted 20 January 2000

Abstract

This paper describes Pajé, 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. Pajé is capable of representing a wide variety of interactions between threads. The main characteristics of Pajé, interactivity and scalability, are exemplified by the performance tuning of a molecular dynamics application. In order to be easily extensible, the architecture of the system was based on components which are connected in a data flow graph to produce a given visualization tool. Innovative components were designed, in addition to "classical" components existing in similar visualization systems, to support scalability and interactivity.

Keywords: Performance and correctness debugging; Parallel program visualization; Threads; Interactivity; Scalability; Modularity

1. Introduction

The Pajé visualization tool described in this article was designed to allow programmers to visualize the executions of parallel programs using a large number of

* Corresponding author. http://www-apache.imag.fr/apache.
E-mail addresses: Jacques.Chassin-de-Kergommeaux@imag.fr (J. Chassin de Kergommeaux), benhur@inf.UFSM.br (B. Stein).

[1] APACHE is a research project supported by CNRS, INPG, INRIA and UJF. This research was done while B. Stein was on leave from U. F. de Santa Maria and supported by a CAPES-COFECUB grant.

PII: S0167-8191(00)00010-7

communicating threads (lightweight processes) evolving dynamically. Combining threads and communications is increasingly used to program irregular applications, mask communication or I/O latencies, avoid communication deadlocks, exploit shared-memory parallelism and implement remote memory accesses [6,7,9]. Achieving the same results using (heavy) processes, communicating through a message-passing library such as PVM [21] or MPI [19], involves considerable programming efforts. All possible cases of unbalance must be predicted by the programmer of an irregular application. Masking communication and I/O latencies requires to manage a communication automaton, on each of the nodes of the parallel system. System (heavy) processes having disjoint address spaces are not suited for exploiting shared memory parallelism. On the contrary, it is fairly simple to spawn several threads to cope with the evolution of an irregular problem or mask communication latencies. In addition, inner parallelism of shared memory multiprocessor nodes can be exploited by several threads sharing the same memory. Remote memory accesses can be serviced by dedicated threads.

The ATHAPASCAN [2,8] programming model was designed for parallel hardware systems composed of shared-memory multi-processor nodes connected by a communication network. It exploits two levels of parallelism: inter-nodes parallelism and inner parallelism within each of the nodes. The first type of parallelism is exploited by a fixed number of system-level processes while the second type 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. The visualization of the executions is an essential tool to help tuning applications implemented using such a 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 parallel programs visualization tools [11,12,14,15,20,22,23] 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. It is of course conceivable to use these systems to visualize the activity of multi-threaded nodes, representing each thread as a node. In this case, the number of threads should be fairly limited and should not vary during the execution of the program. The existing visualization tools are therefore not adapted to visualize threads whose number varies continuously and life-time is often short. The problem raised here is similar to the "scalability" problem arising when these tools are used to visualize the activity of a high number of processors: the execution of ATHAPASCAN programs, even using a small number of nodes, can result in the creation of a high number of threads having a wider variety of interactions. In addition, these tools do not support the visualization of local thread synchronizations using mutexes or semaphores.

Some tools were designed to display multithreaded programs [10,25]. However, they support a programming model involving a single level of parallelism within a

node, this node being in general a shared-memory multiprocessor. ATHAPASCAN programs execute on several nodes: within the same node, threads communicate using synchronization primitives; however, threads executing on different nodes communicate by message passing. Compared to these systems, Pajé ought to represent a much larger number of objects.

Pajé was designed to be interactive, scalable and extensible. In contrast with passive visualization tools [11,20] 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 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. Extensibility is an important characteristic of visualization tools to cope with the evolution of parallel programming interfaces and visualization techniques. Therefore, the environment ought to be extensible enough to ease the inclusion of on-line analysis facilities and dynamic insertion of trace data. Extending one part of Pajé should be transparent to the other parts. Its modules should be reusable in different configurations of the environment.

The organization of this article is the following. Section 2 summarizes the main functionalities of Pajé by exemplifying its use for tuning a large application in molecular dynamics. Section 3 discusses the design of Pajé. The two last sections present related work and conclude.

2. Main features of Pajé

The functionalities of Pajé are exemplified by the tuning of a very large molecular dynamics application. Decomposing the computation performed by each node in a number of threads enabled to overlap communications with computation in a very natural way. Using Pajé to tune this application proved to be very helpful to improve load balancing as well as overlapping of communicating and computing threads. In order to visualize a program execution, it is first necessary to trace an execution of this program to produce the trace that will be used as input data by Pajé.

2.1. 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 is used to estimate a precise global time reference [17]. Dated events are causally coherent, the estimated global time being available at the end of the instrumented application which prohibits on-line dating. This is not considered as a drawback since traces are intended for post-mortem analysis and visualization only. The events are stored in local event buffers, which are flushed when full to local event

files. The collection of events into a single file is only done after the end of the user's application to avoid interfering with it.

The problem of perturbation of parallel applications due to the presence of a tracing tool is a difficult one. Although intrusion can be reduced by a careful implementation of the tracing tool, it cannot be eliminated. The main causes of intrusion are the flushing of local event buffers, the accumulation of the delays of each individual event generation, as well as the extra synchronizations added to the executing threads. To limit the tracing intrusion, on-line compacting of events is used. This allows a gain of space of about 50% with respect to a non-compacted representation of events. The number of buffer flushes is significantly reduced and so is the perturbation of the application. To further limit the intrusion of the traced threads, the management of event buffers is performed by a specific low priority thread.

Finally, to allow users to quickly find the statement in their source code which generated a particular event, recorded events may contain the line number of that statement and the identifier of the source code file. This feature is used by Pajé to implement source code click-back.

2.2. Visualization of threads in Pajé

The visualization of the activity of multi-threaded nodes is performed by a space-time diagram. This diagram (see Fig. 1) combines in a single representation the states and communications of each thread (among other things, discussed later). The horizontal axis represents time while threads are represented in the vertical axis, grouped by node. The space allocated to each node of the parallel system is

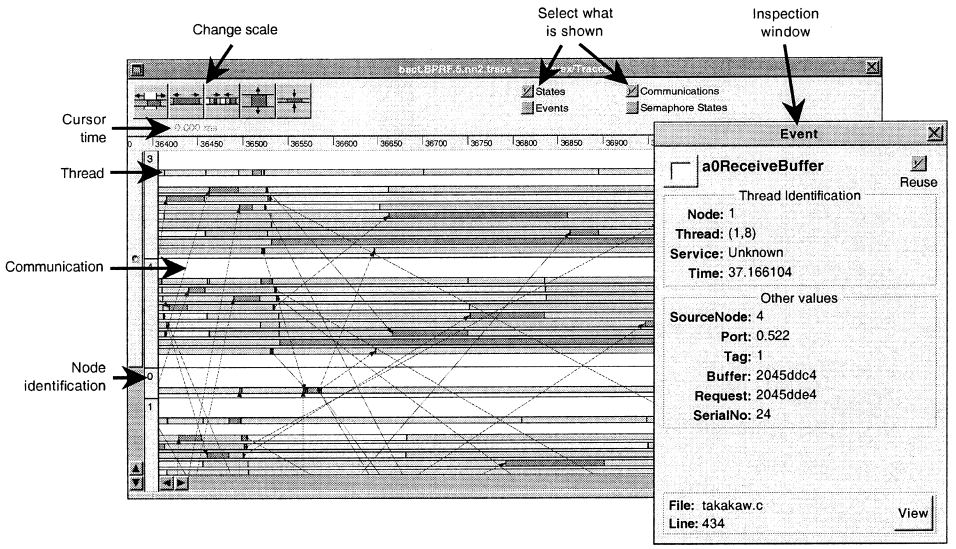

Fig. 1. Visualization of an ATHAPASCAN program execution. Blocked thread states are represented in clear color; runnable states in a dark color. The smaller window shows the inspection of an event.

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. It is not the most compact or scalable representation, but it is very convenient for analyzing detailed threads relationship, load distribution and communication latency masking. Pajé deals with the scalability problem of this visualization by means of filters, discussed later in Sections 2.9 and 3.4.

The user can move the view backward and forward in time, within the boundaries of the time window currently managed by Pajé. When it is needed to move beyond the initial boundary of the current time window, a previously recorded state of the simulator is restored and the trace is simulated again, until the period of interest is reached.

2.3. Molecular dynamics application

The molecular dynamics application simulates the movement of atoms of proteins [1]. It consists of repeatedly computing the successive positions in time of the atoms of a given system, starting from their initial positions and speeds. The positions of the atoms are computed using Newton's motion equation: the forces taken into account are non-bound electrical and Van der Waals forces as well as bound forces modeling the cohesion of molecules. This application is able to cope with large molecular structures of proteins. We have simulated the movement of the largest protein structure found in the Brookhaven Protein Data Bank: β-galactosidase [13]. After immersion of the protein of about 65 240 atoms into a 100 Å radius sphere of water, we obtain a system of 430 000 atoms. The size of this system is more than four times bigger than the size of the systems handled by other current MD program.

The calculation of the forces between each pair of atoms constitutes the main bottleneck of the computation of an iteration of this molecular dynamics application. In order to decrease the volume of computations, only the interactions of each atom with its neighbors were considered, i.e., only with the atoms included in a cut-off sphere of a given radius. This approximation makes the problem irregular from the parallelism point of view: the pairs of atoms for which it is necessary to compute a force depend on the position of the atoms in the system. However, the problem has a good data locality.

2.4. Parallelization of the application

A traditional parallelization for the simulation of a great number of atoms consists in mapping part of the simulated space on each node. Each node deals with the movements of the atoms belonging to its part of the simulated space. To compute the forces exerted on its atoms, it exchanges the positions of the atoms close to the border of its portion of space with the nodes in charge of the neighbor spaces.

Each node computes the forces which are exerted between the atoms of its domain. And, in agreement with the nodes in charge of the neighbor spaces, it computes part of the forces which are exerted between its atoms and the atoms of the

border of its domain. The nodes then exchange the forces which are exerted between the atoms at the border of their domain.

Lightweight processes allow the exploitation of fine grain concurrency and automatic overlap of communication overhead and computation. In our case, a thread deals with the computation of the forces between the atoms of its domain. It requires only the local data of the node. It is then possible to mask the time of the communications of coordinates and forces with other nodes. Only synchronizations between the threads of a node needs to be described. These synchronizations depend on the access to the shared data. Then, during the execution, the scheduling of the threads will automatically overlap the communication overhead with the local computations.

The following section describes more precisely the role of each thread in an iteration of molecular dynamics, and establishes a link with the threads of the trace.

2.5. Parallel solution using threads

On a node p, a simulation is performed by the following threads:

- *Main thread*. It manages the access to the data shared with the other local threads. More precisely, it manages the synchronization of the threads between the various read/write phases of the data of its atoms. It also computes part of the forces of interaction relative to the geometry of the molecules of the system. Finally it integrates the equations of movement for its atoms. On the traces of Figs. 2(a) and (b), it is the uppermost thread of each node.
- *Threads that send coordinates and receive forces*. There exists on node p a copy of this kind of thread for each neighbor node (a node is a neighbor of p if it has a part of the space close to the one of p). These threads send the coordinates of the atoms and receive the forces computed on the corresponding neighbor nodes. The second, third and fourth threads (from the top) of each node on the traces of Figs. 2(a) and (b) are of that type.
- *Threads that compute the inter-nodes forces*. There also exists a copy of this kind of thread for each neighbor node. They are in charge of receiving the coordinates of the atoms, computing the forces between the atoms of two different nodes and sending back the computed forces. The fifth, sixth and seventh threads (from the top) of each node on the traces of Figs. 2(a) and (b) are of that type.
- *Thread that computes the local forces*. It computes the forces between the atoms of node p. This is the thread on top of the lower-most one on each node in Figs. 2(a) and (b).
- *Thread of control*. It communicates control information for the simulation between node p and the main node. This is the lower-most thread on each node in Figs. 2(a) and (b).

Remark. It is always possible to relate a visual object – thread state for example – with the corresponding instruction of the parallel program whose execution is being displayed. However, to ease the identification of the threads of the program, it would be nicer to design them by some user-defined name such as ''Main'', or ''Control'', etc.

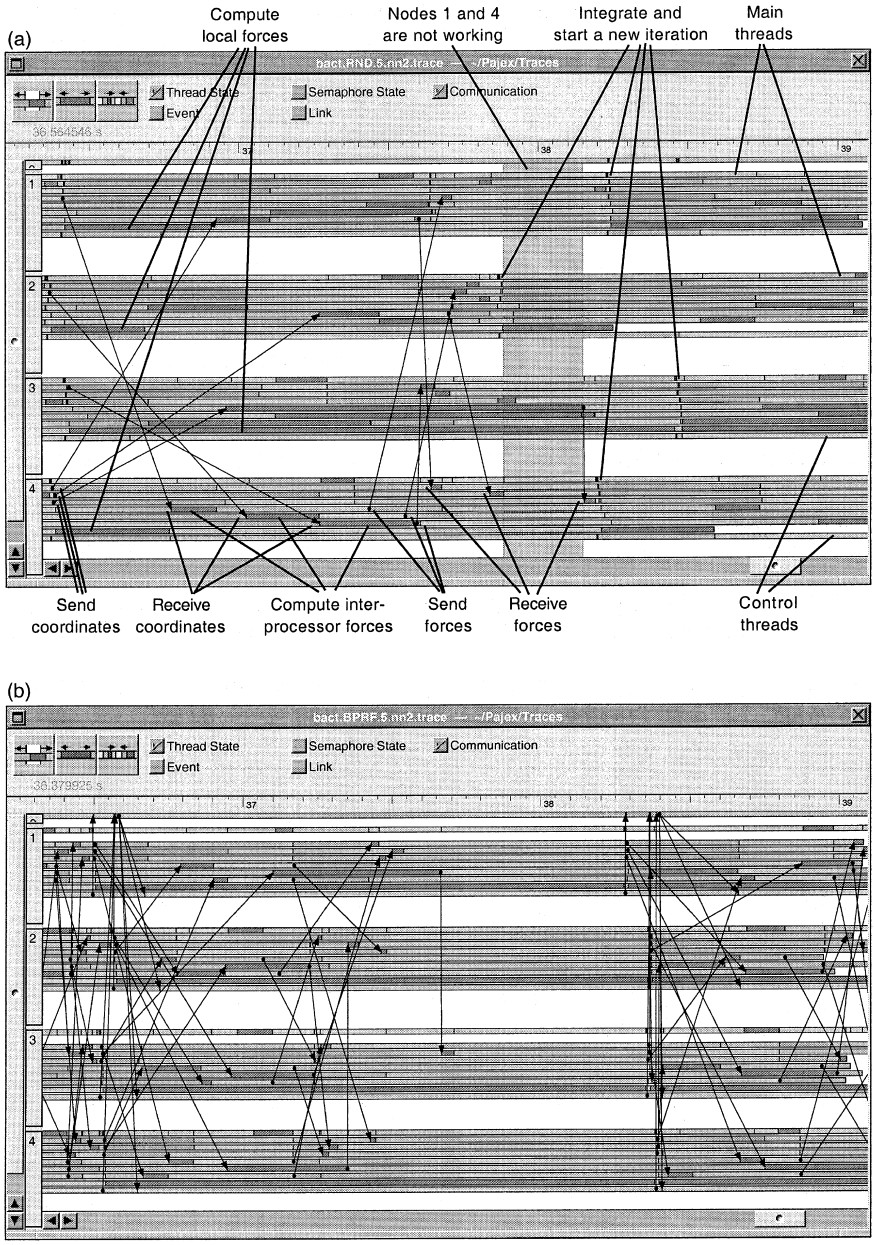

Fig. 2. Visualization of an iteration of the molecular dynamics program: (a) With random placement of domains – nodes 1 and 4 wait a long time for forces computed by node 3 (only communications involving node 4 are shown) and (b) with a better placement of domains, all nodes work all the time.

This is not currently possible with the ATHAPASCAN library whose functionalities are a subset of the *pthread* standard. Such a facility could be easily added to ATHAPASCAN, the symbolic thread identification been stored in the attributes of each thread (*setspecific* function of the *pthread* standard). This possibility is already used by the ATHAPASCAN tracer and Pajé to number threads in order to identify correspondants, since the thread identification provided by the *pthread* standard is an opaque type, unmanageable by the tracer and the visualizer.

2.6. Visualization of the application

A detailed visualization of the lightweight processes gives an invaluable help to the programmer, allowing him:

- to check the coherence of the synchronizations between the threads sharing data on a given node;
- to check the balancing of load between nodes and thus to contribute to the development of good mapping heuristics.

Figs. 2(a) and (b) represent two traces of execution of an iteration of molecular dynamics using two different placements of computational load. On these traces, node 0 is used only to control the simulation. Fig. 2(a) represents a trace of execution with a random initial mapping of the tasks on the nodes. The computational load is unbalanced. One can see on the highlighted portion of the trace that the less loaded nodes (nodes 1 and 4) are waiting for the other, more loaded, node (node 3).

Fig. 2(b) represents a trace of execution that uses an initial mapping heuristic that balances the computation load and minimizes the communication volume. Here, one can see that during an iteration, the nodes work all the time and finish their iteration at the same time. One can also observe how the thread that computes local forces overlaps the communications of the other threads.

2.7. Visualization of local synchronizations

ATHAPASCAN provides primitives that allow the synchronization between threads of a node. These primitives are the classical semaphores and mutual exclusion locks (mutexes). In the space-time diagram of Pajé, the states of semaphores and locks are represented just 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. Pajé recognizes three different states for a semaphore: when there is some thread blocked in it, when a thread will block in it if it issues a "P" operation, and when a thread can issue a "P" operation without blocking. An example of the visualization of some semaphore operations can be seen in Fig. 3.

Unlike semaphores, locks have ownership, that is, at most one thread at a time can hold a lock. Other threads block if they try to hold it, until the lock is released by the thread that holds it. Based on this behavior, there is a second way of representing locks in the space-time diagram. In this representation, each lock is associated with a color, and a rectangle of this color is drawn close to the thread that holds it (as

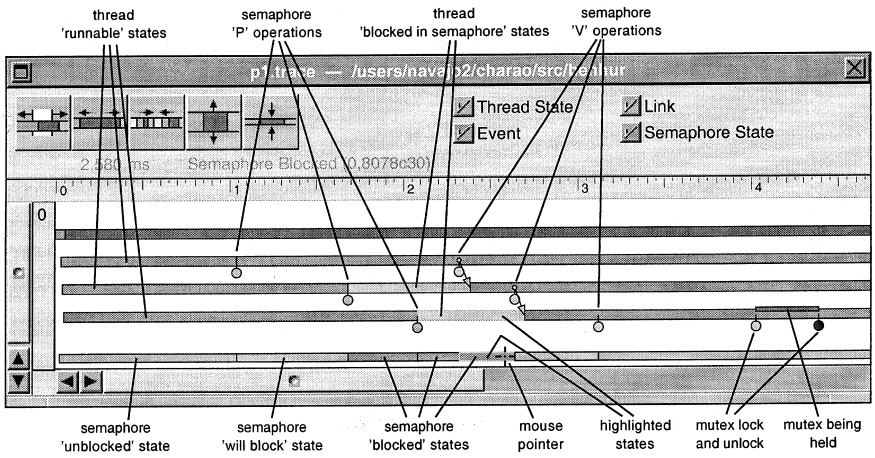

Fig. 3. Visualization of semaphores. Note the highlighting of a thread blocked state because the mouse pointer is over a semaphore blocked state, and the arrows that show the link between a "V" operation in a semaphore and the corresponding unblocking of a thread.

shown on the right of Fig. 3). Also, to ease the identification of the threads that are blocked waiting for a lock, a different rectangle with the color of the lock is displayed close to these threads (not shown in Fig. 3).

2.8. Interactivity

In non-interactive visualization tools, users can only control the simulation speed. In contrast, Pajé gives the possibility to move in time. Progresses of the simulation are entirely driven by user-controlled time displacements. At any time during a simulation, it is possible to move backward in time to a previous state. Moving around the current simulation state is fast, while moving to a remote position can take longer (see Section 3.3.1). In addition, Pajé offers many possible interactions to programmers: displayed objects can be inspected to obtain more detailed information, identify related objects or check the corresponding source code.

Not all the information that can be deduced from a trace file is directly representable (or its simultaneous representation may not be desirable) in the space-time diagram. More information can be obtained upon user request. All the information available for a displayed object can be shown in an inspection window, created by clicking over the representation of the object. Such an inspection of an event in ATHAPASCAN is shown in Fig. 1.

Another very useful information is the interrelation between the entities of the diagram. For example, the color of a thread state indicates that this thread is blocked in a semaphore during some time period, but the identification of the semaphore is not immediate. Representing this information permanently on the screen would clutter the visualization. In Pajé, moving the mouse pointer over the representation of this blocked state highlights the corresponding semaphore state, allowing an

immediate recognition (see Fig. 3). Similarly, all threads blocked in a semaphore are highlighted when the pointer is moved over the corresponding state of the semaphore. The name of the state under the pointer is also displayed on the top of the window, together with the time corresponding to the pointer position. Many similar relations are displayed in this way in Pajé.

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. Likewise, selecting a line in the source code browser highlights the events that have been generated by this line.

2.9. Filtering of information and zooming capabilities

It is not possible to represent simultaneously all the information that can be deduced from the 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, abstract view of the data and easy, intuitive ways for them to access more details from what seems to be the cause of problems, seems to be a good way of helping them find out what these problems are. Accessing more detailed information can amount to exploding a synthetic view into a more detailed view or getting to data that exist but have not been used or are not directly related to the visualization.

Pajé offers several filtering and zooming functionalities to help programmers cope with this large amount of information. Filters in Pajé can be of several types:

- *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. 4).
- *Selection*. Permits the removal of objects from a visualization. This selection can be based on the type of a visual object (not shown events, thread states, communications, etc), on its subtype (of all possible thread states, show only the ones that represent a running thread) or on some specific instance (select which nodes, threads, semaphores, mutexes or groups to show, see Fig. 4).
- *Repositioning*. Allows users to choose the order in which the objects are shown, so that, for example, related nodes can be displayed close together. Pajé has a repositioning filter that reuses the screen space made available by the termination of objects such as short-living threads (see Fig. 5(a) and (b)).
- *Reduction*. Provides more abstract representations of information, for the production of synthetic views. Fig. 6 shows the same execution as Fig. 2(b), with only one line per node, containing states that represent the number of active threads at each instant. It also shows a pie graph of CPU activity in the time slice selected in the space-time diagram.
- *Visual changes*. The correspondence between the type of an entity and the color, shape and size of its graphical representation can be personalized by a user, and is remembered by Pajé across executions.

Being able to switch from detailed to grouped visualizations gives programmers zooming capabilities within a node or between several nodes.

3. Design of the visualization environment

For a visualization environment to be really useful, it needs to be easily adaptable to changes. These changes can be in the paradigms used by the parallel programming environment, in new user needs of different capabilities and different visualizations, or even in the format of trace files. To be able to resist to these changes, Pajé is

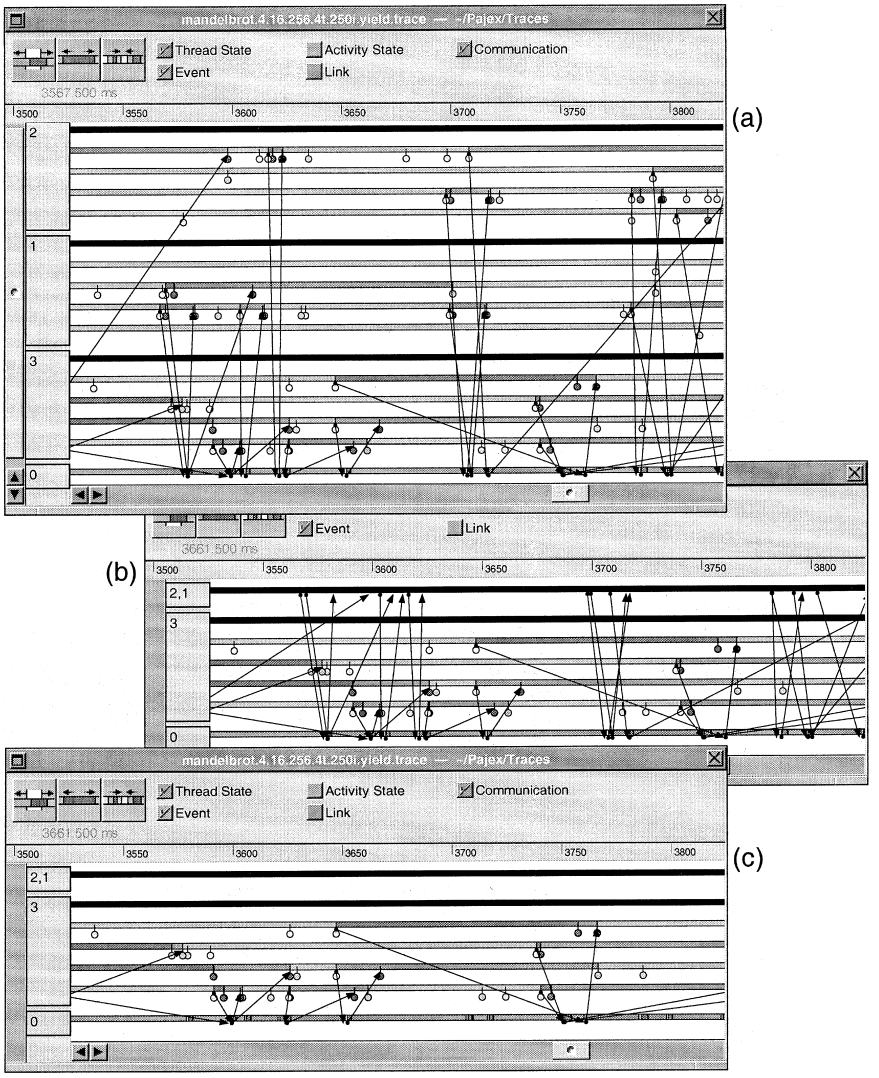

Fig. 4. Use of grouping and selection filters. In figure (a), the events of node 0 are filtered. In figure (b), nodes 1 and 2 are grouped and the events of this group are filtered. In figure (c), the communications of this group are also filtered.

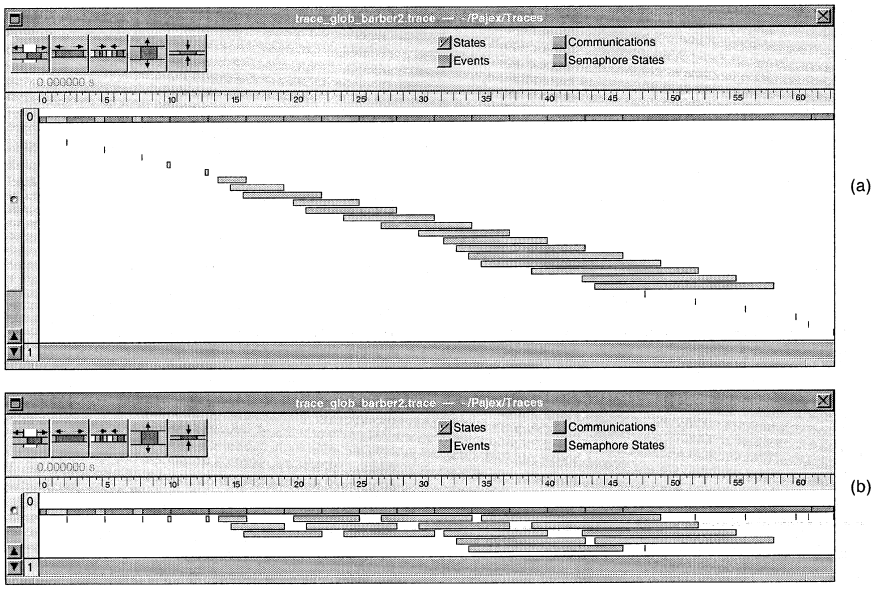

Fig. 5. Reusing the space of short-lived threads: (a) View of a program with short-lived threads and (b) same program, reusing the space of terminated threads.

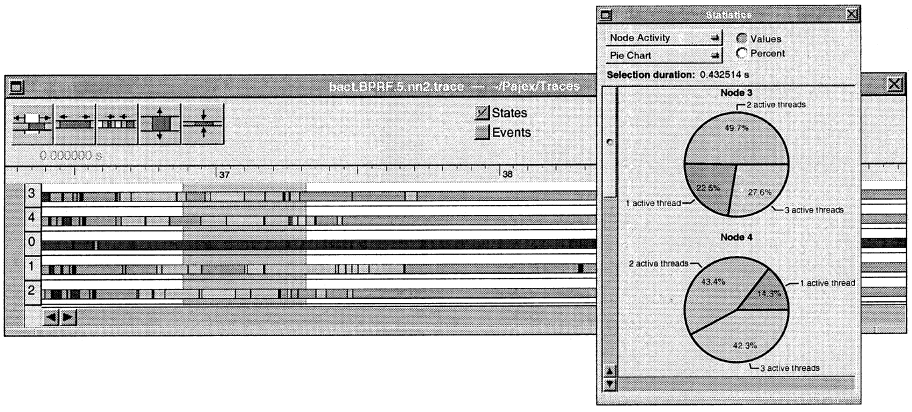

Fig. 6. CPU utilization. Grouping the threads of each node to display the state of the whole system (lighter colors mean more active threads); the pie-chart shows the percentage of the selected time slice spent with each number of active threads in each node.

organized as a graph of independent components, communicating exclusively by well defined and extensible protocols. Besides "classical" components existing in similar visualization tools, original components were developed to support interactivity and zooming capabilities.

3.1. Graph of components

To ease the extensibility of the environment, it is developed in a very modular way, similarly to Pablo [20]. Each component is an independent object, that communicates with others through communication links, building a coarse grain data-flow graph. The vertices of the graph are analysis components while its arcs are communication links. Data traveling on the arcs are objects representing the entities – events, thread states, communications, etc. – of the analyzed program (see Fig. 7). A given visualization environment is made by connecting the available components in a graph representing the way the data will be analyzed.

Contrary to Pablo, the graph of components in Pajé is not a pure data-flow graph: some components are connected by bidirectional links for the exchange of control signals. The control flow graph is mainly required by the implementation of interactivity in Pajé. The control flow and its interaction with the data flow are described in Section 3.3.

Fig. 7 represents an example of a simple data-flow graph, including a trace reader, a simulator, a statistics and a visualization module. The trace is read from the trace file by the trace reader which produces objects representing the events produced by the analyzed program. These events are used by the simulator to simulate the activity of the traced program and produce objects representing more abstract entities such as semaphore states, communications, etc. These objects are eventually used by the statistics module as well as by the visualization module.

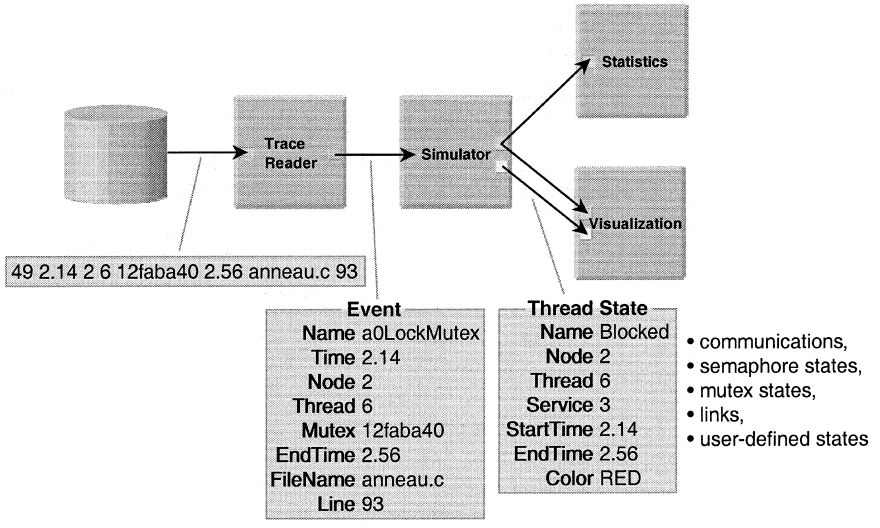

Fig. 7. Example data-flow graph. The trace reader produces event objects from the data read from disk. These events are used by the simulator to produce more abstract objects, like thread states, communications, etc., traveling on the arcs of the data-flow graph to be used by the other components of the environment.

The modules share a common interconnection interface and a common protocol for accessing the data in each entity (see Section 3.3.2). These two characteristics allow the extension of the environment by the addition of new components to the graph. Also, where possible, components were designed with no semantical knowledge of the data they process. This independence with respect to input data, combined with a well-defined protocol for data access, makes the components easily reusable for processing different types of entities produced by the parallel program (even those defined by the user).

3.2. Classical components

Most of the components of our environment can be found in other existing visualization environments [20]: controller, trace readers, simulators of parallel programs, etc.

- *Controller*. This module is always present. It is not inserted in the data-flow graph and therefore is not visible on Fig. 7. It is the first module to be executed. It dynamically loads and connects the other modules according to an environment configuration file. Then it manages the user interface as well as the use of memory.
- *Trace readers*. Readers for two versions of ATHAPASCAN-0 and for the Alog trace format used by the upshot tool [12] and the IBM SP/1 tracer [23] have been implemented.
- *Simulator*. Analyzing the events produced by the trace reader, the simulator produces the thread states, communications, links, semaphore and mutex states. It can register a complete simulation state so that it is possible to move back in time before the limit of the current observation window (see Section 3.3.1). Besides simulating the events defined by ATHAPASCAN-0, the simulator allows the definition of user events, states and communications. This is a powerful mechanism to easily extend the types of objects that can be visualized by Pajé.

3.3. Implementation of interactivity

Structuring the environment as components of a data-flow diagram is well suited for the implementation of modules needing a single access to the information derived from the trace. An example is the module that computes the CPU usage of nodes. This module checks the type of the objects produced by the simulator. If the object is a running state, its duration is added to an accumulator associated to the object's node. Then, the thread state object does not need to be accessed anymore. Another example is a passive visualization module that, for each object, displays a corresponding visual representation that cannot be interrogated nor changed. After being displayed, the object is not accessed anymore. However, interactive modules need to access the data objects several times. In a normal data-flow graph, a module receives each data object independently. In this case, it should either store the objects or fetch them again each time they are needed. The first solution would result in added complexity of such modules, to manage a large volume of data, as

well as data replication if more than one module of this type were used. The second solution would result in added computation costs to read and simulate the trace several times.

3.3.1. Observation window and compounding component

To overcome the contradiction between having Pajé built from a data-flow graph of components for extensibility, and the requirement of being able to access elementary visual data objects several times, the elementary objects produced by the simulator and used for visualization are kept in a complex data structure called *observation window*. Elementary objects are elementary events recorded during the observed execution, states of threads and semaphores between two events, communications, etc. Within this observation window, it is possible to move back and forth in time, without having to re-simulate the observed execution, because the objects are directly accessible in memory. The observation window slides forward in time by including new objects and "forgetting" old ones, when the user attempts to visualize after the end of the current observation window. The state of the simulation is recorded at regular intervals of simulation time, so that later it is possible to move past in time, outside the observation window. In such cases, simulation is restarted from the closest saved state, before the date of interest.

The observation window object is built by the *compounding component*, from elementary objects produced by the simulator. The compounding component breaks the pure data-flow graph aspects of the graph of components: it does not output the data that it generates. Instead, each elementary object input by the compounding component is linked to the observation window. After the compounding component, the flow of information on the graph, instead of being simply triggered by data availability, is explicitly activated by *control messages*; the data flows on demand, only when requested by a component. Control messages can go in both directions (see Fig. 8).

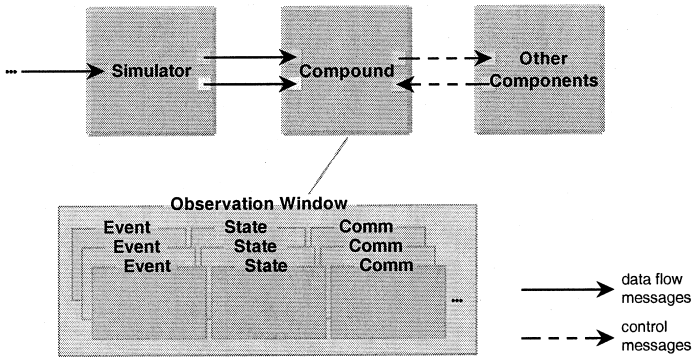

Fig. 8. Compounding component and observation window. The compounding component organizes the access by other components to all elementary entities (events, states, communications) produced by the simulator through the use of an observation window. After the compounding component, components are linked by control links, instead of data links used before it.

3.3.2. Control messages

There are two kinds of control messages in Pajé: messages that go forward in the graph, called *notifications* and messages that go backwards, called *queries*. Notifications inform other components that data has changed, for example that the observation window was slid or that the hierarchical structure of elementary objects was changed, etc. Queries are used by data-consuming components (such as a visualization component) to obtain information about the data encapsulated in the observation window. As there are many types of information in the observation window, there are many kinds of queries to:

- get global information about the execution or about the observation window (number of nodes, maximum number of threads in each node, hierarchical structure of elementary objects, etc.);
- get some of the elementary objects, chosen by time and type: e.g. all events in thread 1 of node 7 between 3.2 and 4.3 s of execution;
- get more information concerning an elementary object, such as the node it belongs to, its timing, shape or color, other objects related to it, etc.;
- ask for the inspection of an elementary object-used by visualization components when the user selects a graphical representation of an elementary object; the visualization component does not need to know all the details of the inspected object.

All the complexity of storing and accessing the large quantity of data generated from the trace is isolated in the observation window. Besides simplifying the construction of data-consuming modules, centralizing access to data has some other advantages for the construction of filters (see Section 3.4) and for the management of memory by the controller component. Every time a component queries for data that is not in the current observation window, the compounding component informs the controller component, that can restart data-flow reading and analysis of the traces until the needed data is linked to the observation window.

3.3.3. Data structure of the observation window

Because it encapsulates such a large number of elementary objects, which need to be searched very frequently during the visualization of a parallel program execution, the structure of the observation window favors efficient search. The number of elementary objects involved during a visualization may be very large. For example in the programs tested so far, up to 10^4 events per second of execution time per node were produced, generating a larger number of elementary objects.

The most frequent accesses to the observation window are done to search the object currently pointed to by the cursor on the screen: moving the mouse of one pixel may involve searching the entire observation window for the new object pointed to by the cursor. Other frequent accesses are performed to identify the objects to be displayed on the screen, given the dates of the events located on the screen boundaries.

Within an observation window, data is structured hierarchically into *containers* such as threads or semaphores. It is very important for this structure to be easily reconfigurable for filtering (see Section 3.4) and extensibility reasons. Within a container, two types of elementary objects exist: ''instantaneous'' objects, such as

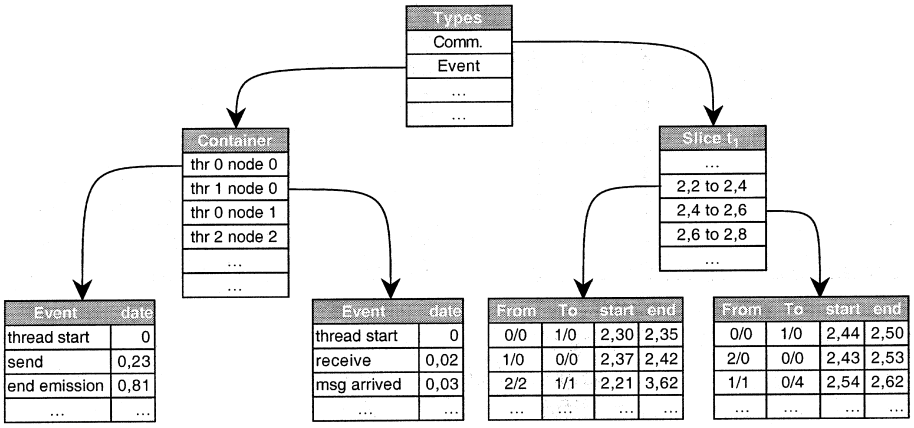

Fig. 9. Structure of an observation window. Types and containers are stored in hash tables. Instantaneous objects are sorted by date. Non-instantaneous objects are grouped in time slices, depending on their initial date. Within each time slice, they are sorted by termination date.

elementary events, and "non-instantaneous" objects such as thread states and communications, which can, in the worst case, last during the entire visualization and be displayed each time a new display is computed. The observation window is organized as a hierarchy of tables (see Fig. 9). The types and containers tables are organized as hash tables, whose access is thus performed in constant time (O(1)). The instantaneous objects of each container are stored in a table sorted by date and whose access is thus performed in logarithmic time.

Providing efficient accesses to non-instantaneous data objects involved considerable design and programming efforts: several data organizations were designed and compared [5]. In the most naive organization, data were sorted by initial date, resulting in a worst case search of all objects. In the data structure selected for Pajé, non-instantaneous data objects of each container are grouped in time slices, an object belonging to the time slice corresponding to its creation date. Within each time slice, data are sorted by termination date. Using such a data structure, it is possible to eliminate rapidly the non-instantaneous objects irrelevant to the current time period being represented on the screen (see Fig. 10). The number of time slices was selected by measuring the search time of an object in the observation window, as a function of the number of slices and of the position of the searched object in the slice, for various numbers of events in the observation window and various durations of the time interval of interest (value of $t_2 - t_1$).

3.4. Filters

Visualization modules fetch data from the observation window each time they compute a new visualization. Filter components select or transform the data to implement the filtering and zooming functionalities described in Section 2.9. Filters

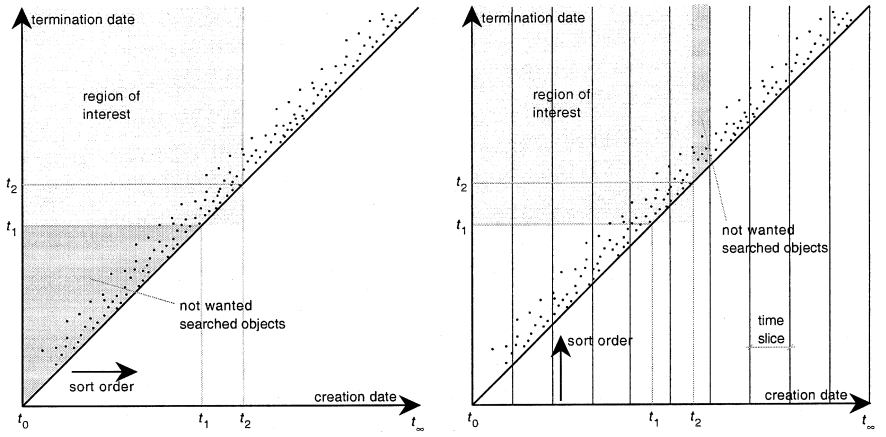

Fig. 10. Organization of non-instantaneous data objects within a container. Each object is represented by a point whose *X*- and *Y*-coordinates are the creation and termination dates. The objects of interest are located in the light grey area. The objects located in the dark grey area are searched while they are not wanted. A lot fewer non wanted objects are searched in the retained data organization (right) than in the naive one (left).

transform the replies from the compounding component to the queries of the visualization components, without modifying the observation window data. A visualization component connected to a filter obtains all information concerning elementary objects indirectly by querying the filter. Filters can be modified or deactivated dynamically: visualization components affected by these changes are notified so that they can update their visualizations; this updating being performed from filtered data, it will therefore take into account the latest filtering modification.

Filters can easily be connected to the graph, anywhere in the path from the compounding component to a data-consuming component. Fig. 11 shows a graph with two visualizations, one accessing the raw data from the observation window and another accessing this data through a filter.

A filter does not generate a new object for each elementary object, nor does it alter the elementary objects. Instead, filtered data is produced only when requested. Other possible implementations of filtering (considering a data-flow graph and the need of access to the data for interactivity) would result either in duplication or modification of the elementary objects of the current observation window, the former resulting in high memory consumption and the latter being unsuitable for the situation in which a module would use filtered data while another would use them non-filtered.

4. Related work

A large number of tools have been developed to visualize the execution of parallel programs. In most tools, the number of processes remains constant during a simulation: they are not adapted to the visualization of parallel programs creating pro-

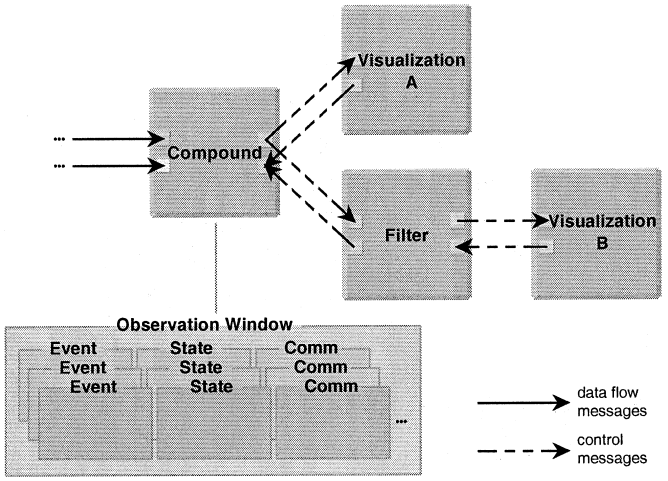

Fig. 11. A filter component. Visualization A has a direct access to the compounding component, obtaining non-filtered data from the observation window. Visualization B queries data from the filter, obtaining a filtered view of the same data.

cesses dynamically, such as multi-threaded programs. Many visualization tools are not interactive, only giving the possibility to adjust the simulation speed, it is not possible to interact with the displayed objects nor to move backwards in time. The most widespread of these tools is Paragraph [11], which provides a large number of possible views. Although the number of processes may be high, Paragraph does not scale well, since most views become hard to understand when the number of processes is high. Pablo [20] is an extremely powerful visualization environment whose architecture inspired Pajé's. The architecture of Pablo is based on an extensible graph of components which can be connected to produce a given visualization tool. A large number of components were developed, providing a wide number of visual or sound representations. Its graph of components, being a pure data-flow graph, is simpler to modify than that of Pajé, at the expense of not supporting interactivity. A form of scalability is provided by switching automatically from trace recording to counting when the volume of traces passes some threshold. The main visualization of the AIMS system [24] is a space-time diagram that represents the execution of a parallel program on a possibly large number of nodes, showing the functions executed at each period and the communications between nodes. From this visualization, source code can be inspected. The Paradyn performance debugging environment [18] was designed to identify performance errors automatically. Performance data is monitored online to adjust the amount of performance data collected: more data is collected where a performance problem is expected. Non-interactive histogram and bar-chart visualizations are provided.

Several visualization tools provide some form of interactivity since it is possible to inspect the displayed objects or to relate a given visualization to the source code. This is the case of the upshot visualization tool [12,23] which also gives the possibility

to move past in time, provided that the entire trace is available in main memory. Atempt [14] can be used to change the order of events in order to test different execution paths in conjunction with other tools. An ATHAPASCAN-0 reader was implemented for Atempt, each thread being represented as a different process, thus giving the possibility to visualize ATHAPASCAN-0 programs executions involving a limited number of threads since there is no support for representing dynamically created processes. VAMPIR [15] is a visualization tool designed for MPI programs. Program executions are represented as a time-line view including processor states and communications. Being designed for MPI, VAMPIR assumes a constant number of processes during a visualization session. The NTV trace visualizer [16] shows processor states and communications in a time-line visualization. Annai [3] is an integrated environment for developing and debugging parallel programs. One of its visualizations is a space-time diagram that can show the states and communications of various processors, possibly combined with some other time-varying quantities, like memory consumption.

To the authors' knowledge, only two visualization tools were conceived with support for multi-threaded programs where the number of threads varies dynamically during the execution of a program. In Gthread [25], the execution of threads and their synchronizations can be visualized. However, threads must be located on a single node and have no other form of communication than local synchronization (no message passing). Gransim [10] is a visualization tool of a parallel functional language (Glasgow Parallel Haskell) simulator. As in ATHAPASCAN, dynamically created threads can be executed by each of the nodes of the simulated system. It is possible to visualize the global activity of all nodes of the system or the activity of the threads of a particular node. Gransim is different from other visualization tools since it produces its visualizations as printable files. In addition, it has no representation for communications and synchronizations.

5. 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 features of the environment are interactivity and scalability. These properties were achieved – without sacrificing other "desirable" properties of a visualization environment, such as extensibility and reusability – by a very careful modular design and the use of sophisticated data structures. The Pajé environment is structured as a graph of independent and reusable software components connected by data- and control-flow relations. Interactivity and scalability are mainly supported by a complex data structure called the observation window and produced by a compounding component. Using the observation window, it is possible to compute the displays requested by users rapidly enough so that the response time of the tool remains good. The compounding component can be combined with various filter components operating on the observation window, in order to offer to users zooming

capabilities between several observation levels. Such zooming capabilities give the environment its scalability since it is possible to observe the execution of a parallel program on a large system at high level of abstraction, without missing details which can be observed by zooming on a particular node or group of nodes.

Future work includes the development of new visual components, both classical ones and higher level representations of the ATHAPASCAN programming model (call graph, data structures, etc.). Another foreseen extension concerns the coupling of Pajé with a distributed symbolic debugger such as DDBG developed at UNL [4] to provide a high level debugging interface for ATHAPASCAN programs.

Acknowledgements

Florin Teodorescu designed a prototype thread visualization tool. Philippe Waille implemented the ATHAPASCAN tracer. The anonymous referees helped improving the paper by their useful comments. All APACHE project research reports are available at http://www-apache.imag.fr.

References

[1] P.-E. Bernard, B. Plateau, D. Trystram, Using threads for developing parallel applications: molecular dynamics as a case study, in: R. Trobec (Ed.), Parallel Numerics, Gozd Martuljek, Slovenia, September 1996, pp. 3–16.

[2] J. Briat, I. Ginzburg, M. Pasin, B. Plateau, Athapascan runtime: efficiency for irregular problems, in: C. Lengauer et al. (Eds.), EURO-PAR'97 Parallel Processing, volume 1300 of LNCS, Springer, Berlin, August 1997, pp. 591–600.

[3] C. Clémençon, A. Endo, J. Fritscher, A. Müller, B.J.N. Wylie, Annai scalable run-time support for interactive debugging and performance analysis of large-scale parallel programs, Technical Report CSCS-TR-96-04, Centro Svizzero di Calcolo Scientifico, CH-6928 Manno, Switzerland, April 1996.

[4] J.C. Cunha, J. Lourenço, An experiment in tool integration: the DDBG parallel and distributed debugger, EUROMICRO Journal of Systems Architecture, Second Special Issue on Tools and Environments for Parallel Processing, 1997.

[5] B. de Oliveira Stein, Visualisation interactive et extensible de programmes paralléles á base de processus légers, Ph.D. Thesis, Université Joseph Fourier, Grenoble, 1999, http://www-mediatheque.imag.f (in French).

[6] T. Fahringer, M. Haines, P. Mehrotra, On the utility of threads for data parallel programming, in: Proceedings of the Nineth International Conference on Supercomputing, Barcelona, Spain, 3–7 July, ACM Press, New York, 1995, pp. 51–59.

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

[8] I. Ginzburg, Athapascan-0b: Intégration efficace et portable de multiprogrammation légère et de communications, Ph.D. Thesis, INPG, September 1997 (in French).

[9] M. Haines, W. Böhm, An initial comparison of implicit and explicit programming styles for distributed memory multiprocessors, in: H. El-Rewini, B.D. Shriver (Eds.), Proceedings of the 28th Annual Hawaii International Conference on System Sciences, volume 2: Software Technology, Los Alamitos, CA, January 1995, IEEE Computer Society Press, Silver Spring, MD, pp. 379–389.

[10] K. Hammond, H. Loidl, A. Partridge, Visualising granularity in parallel programs: a graphical winnowing system for haskell, in: A.P.W. Bohm, J.T. Feo (Eds.), High Performance Functional Computing, April 1995, pp. 208–221.

[11] M.T. Heath, Visualizing the performance of parallel programs, IEEE Software 8 (4) (1991) 29–39.
[12] V. Herrarte, E. Lusk, Studying parallel program behavior with upshot, 1992, http://www.mcs.anl.gov/home/lusk/upshot/upshotman/upshot.html.
[13] R. Jacobson, X.-J. Zhang, R. DuBose, B.W. Matthews, Three-dimensional Structure of β-galactosidase from *E. coli*, Nature 369 (1986) 761–766.
[14] D. Kranzlmueller, R. Koppler, S. Grabner, C. Holzner, Parallel program visualization with MUCH, in: L. Boeszoermenyi (Ed.), Third International ACPC Conference, volume 1127 of Lecture Notes in Computer Science, Springer, Berlin, September 1996, pp. 148–160.
[15] W. Krotz-Vogel, H.-C. Hoppe, The PALLAS portable parallel programming environment, in: L. Bouge, P. Fraigniaud, A. Mignotte, Y. Robert (Eds.), Second International Euro-Par Conference, volume 1124 of Lecture Notes in Computer Science, Lyon, France, August 1996, Springer, Berlin, pp. 899–906.
[16] L. Lopez, The NAS Trace Visualizer (NTV) Rel. 1.2 User's Guide, September 1995, http://science.nas.nasa.gov/Pubs/TechReports/NASreports/NAS-95-018/NAS-95-018.ps.
[17] É. Maillet, C. Tron, On efficiently implementing global time for performance evaluation on multiprocessor systems, Journal of Parallel and Distributed Computing 28 (1995) 84–93.
[18] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, T. Newhall, The Paradyn parallel performance measurement tool, Computer 28 (11) (1995) 37–46.
[19] MPI Forum, MPI: a message-passing interface standard, Technical Report, University of Tennessee, Knoxville, 1995.
[20] D.A. Reed et al., Scalable performance analysis: the Pablo performance analysis environment, in: A. Skjellum (Ed.), Proceedings of the Scalable Parallel Libraries Conference, IEEE Computer Society, Silver Spring, MD, 1993, pp. 104–113.
[21] V. Sunderam, PVM: a framework for parallel distributed computing, Concurrency: Practice and Experience 2 (4) (1990) 315–339.
[22] B. Topol, J.T. Stasko, 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.
[23] C.E. Wu, H. Franke, UTE User's Guide for IBM SP Systems, 1995, http://www.research.ibm.com/people/w/wu/uteug.ps.Z.
[24] J. Yan, S. Sarukkai, P. Mehra, Performance measurement, visualization and modeling of parallel and distributed programs using the AIMS toolkit, Software – Practice and Experience 25 (4) (1995) 429–461.
[25] Q.A. Zhao, J.T. Stasko, Visualizing the execution of threads-based parallel programs, Technical Report GIT-GVU-95-01, Georgia Institute of Technology, 1995.